Modellgetriebene Entwicklung sicherer Web Service-Anwendungen

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg, 2017

Marian Borek

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Mündliche Prüfung:	27. März 2017

Zusammenfassung

Die Entwicklung von sicheren Web Service-Anwendungen ist eine schwierige Aufgabe und das Entwerfen von sicheren Protokollen sehr fehleranfällig. In dieser Arbeit wird ein modellgetriebener Ansatz für die Entwicklung von sicheren und auf kryptographischen Protokollen basierten Web Service-Anwendungen vorgestellt. Die Modellierung erfolgt anhand der in dieser Arbeit vorgestellten UML-basierten Modellierungssprache SecureML. Da diese Arbeit auf den Ergebnissen von Moebius [62] aufbaut, wird zudem die sichere Interaktion mit Chipkarten unterstützt. Aus einem plattformunabhängigen SecureML-Modell wird automatisch lauffähiger Programmcode generiert, als auch mehrere formale Spezifikationen. Eine formal Spezifikation wird für das Finden von Sicherheitslücken mittels Model-Checking verwendet. Eine andere dient zur interaktiven Verifikation der Sicherheitseigenschaften. Ein großer Unterschied zu anderen Ansätzen ist, dass das interne Verhalten von Services plattformunabhängig modelliert und somit lauffähiger Programmcode generiert wird. Zudem sind die Modelle detailliert genug, um anwendungsspezifische Sicherheitseigenschaften zu überprüfen und zu garantieren. Damit auch reale Anwendungen mit diesem Ansatz entwickelt werden können, wurde zudem die sichere Integration und Austauschbarkeit von existierenden Web Services betrachtet. Der Ansatz wurde anhand mehrere Fallstudien evaluiert.

Danksagung

An dieser Stelle möchte ich mich bei einigen Menschen bedanken, die zum Gelingen dieser Dissertation beigetragen haben. Mein Dank gilt im Besonderen:

meinem Doktorvater **Prof. Dr. Wolfgang Reif**, für das in mich gesetzte Vertrauen und die Möglichkeit in diesem Bereich promovieren zu dürfen.

meinem Kollegen und guten Freund **Kuzman Katkalov** für seine fachliche und private Unterstützung. Er half mir oft die Dinge im richtigen Licht zu sehen.

Dr. Kurt Stenzel, für seine Zeit und seine unendliche Geduld.

Dr. Nina Moebius, für ihre engagierte und fürsorgliche Betreuung.

Ein ganz besonderer Dank gilt jedoch meiner Verlobten, **Iris Kraus**, die immer nur das Beste in mir sieht und mich in allem unterstützt, sowie **meiner Mutter**, die immer an mich geglaubt hat und ohne die ich niemals soweit gekommen wäre.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

In unserer digitalisierten und immer mehr vernetzen Welt verwalten sicherheitskritische Systeme unser Vermögen, unsere privaten Informationen und sogar unsere Identität. Bei diesen Systemen handelt es sich meistens um verteilte Anwendungen in denen Web Services eine wichtige Rolle spielen. Web Services sind Softwareanwendungen, die anhand standardisierter und maschinenlesbarer Schnittstellen die automatische Interaktion zwischen Maschinen unterstützen. Durch Web Services lassen sich Dienste in verteilten Systemen maschinenübergreifend nutzen und sogar zu neuen Diensten kombinieren. Solche Dienste können z.B. das Verwalten von sensiblen Daten oder das Ausführen von sicherheitskritischen Geschäftsprozessen sein. Die Einsatzgebiete für Web Services sind vielfältig. Sie werden z.B. in elektronischen Bezahlsystemen (E-Commerce), elektronischen Melderegistern (E-Government) sowie bei der elektronische Gesundheitskarte (E-Health) eingesetzt. Bei solchen Systemen handelt es sich oft um große und besonders sicherheitskritische verteilte Systeme mit unterschiedlichen Kommunikationsteilnehmern, deren Interaktion auf kryptographischen Protokollen basiert. Jedoch ist die Entwicklung solcher Protokolle bekanntermaßen schwierig und fehleranfällig [3, 68], da es immer einen potentiellen Angreifer gibt, der durch das Mithören und Manipulieren von Nachrichten bestehende Sicherheitslücken ausnutzen könnte. Ein aktuelles Beispiel ist das Poseidon-Protokoll, welches eine spezielle Implementierung des Standards ISO 8583 [45] ist. Hierbei handelt es sich um ein Sicherheitsprotokoll, welches beim Bezahlen an einem Point of Sale (POS) für die sichere Kommunikation zwischen dem POS und dem Serviceprovider (Zahlungsverkehrsdienstleister) verwendet wird. Durch eine Sicherheitslücke konnte ein Angreifer sensible Daten erfahren, wodurch es möglich war sich im Namen der Händler gültige Gutschriften zu erstellen [26]. Ein weiteres aktuelles Beispiel ist das OAuth 2.0-Protokoll [46, 47] mit dem ein Benutzer einer Anwendung den Zugriff auf Daten erlaubt, die von einem Serviceprovider verwaltet werden. Durch eine Sicherheitslücke im Protokoll, war es möglich, dass ein Angreifer unberechtigten Zugriff auf diese Daten

erlangen konnte [32]. Ein weiteres wichtiges Protokoll, welches bei der Kommunikation mit Web Services oft verwendet wird ist TLS (Transport Layer Security). Durch eine Sicherheitslücke bei der Neuverhandlung des Sitzungsschlüssels war es einem Angreifer möglich eine bestehende Sitzung zu übernehmen [76]. Eine Sicherheitslücke in solchen sicherheitskritischen Protokollen kann schwerwiegende Folgen mit hohem finanziellen Schaden haben. Somit ist für die Entwicklung von sicheren Anwendungen, die auf kryptographischen Protokollen basieren, die Integration von formaler Verifikation in den Entwicklungsprozess essentiell. Außerdem müssen die Sicherheitsaspekte der zu entwickelnden Anwendungen in allen Phasen des Entwicklungsprozesses betrachtet werden.

SecureMDD ist ein modellgetriebener Entwicklungsansatz der beide Aspekte realisiert. Aus einem Anwendungsmodell wird eine lauffähige Anwendung sowie mehrere formale Spezifikationen zum Testen und Verifizieren der Sicherheit automatisch generiert. Dadurch wird nicht nur die Sicherheit der Protokolle gewährleistet, sondern auch Implementierungsfehler bei der Realisierung der Protokolle vermieden und die Anwendung kann auf einer abstrakten Ebene, und somit einfacher, schneller und übersichtlicher entwickelt werden.

1.2 Ziele

Ein großes Problem bei der modellgetriebenen Entwicklung und somit auch bei der modellgetriebenen Entwicklung von Web Service-Anwendungen ist, dass viele Anwendungen nicht als geschlossenes System entwickelt werden können, sondern mit schon vorhandenen Systemen interagieren müssen. Eine Verkaufsplattform muss z.B. für die Bezahlung der angebotenen Ware mit einem bestehenden Bezahldienst kommunizieren. Dabei kann es sich um Systeme handeln die von anderen verwaltet werden oder um eigene Systeme (z.B. Altsysteme), deren Neuentwicklung zu aufwändig wäre. Des Weiteren besitzen Web Services vielseitige Aufgaben zu deren Realisierung oft bestehende Bibliotheken notwendig sind. Solche Aufgaben können analytische und numerische Berechnungen (z.B. Aktienprognosen), Datenbearbeitung (z.B Einlesen und Bearbeiten von Excel und Word-Dateien), Grafikbearbeitung (z.B. Iris-Vergleich), u.v.m. sein. Zudem kommt der Fall, dass existierende Web Services z.B. durch billigere oder effizientere ausgetauscht werden sollen.

Somit ist ein wichtiges Ziel dieser Arbeit, dass bei der modellgetriebenen Entwicklung von sicherheitskritischen Web Service-Anwendungen existierende Web Services unter Berücksichtigung der zu zeigenden Sicherheitseigenschaften integriert und mit geringem Verifikationsaufwand ausgetauscht werden können.

Um eine Web Service-Anwendung modellgetrieben zu entwickeln, muss die Behandlung von Web Services berücksichtigt werden. Diese Arbeit baut auf den Ergebnissen von Moebius [62] auf, die sich mit der Entwicklung von sicheren Smart Card-Anwendungen befassen. Web Services haben jedoch grundsätzlich andere Eigenschaften und ein anderes Verhalten als Smart Cards.

Somit ist das Ziel, Web Services und ihr Verhalten adäquat zu modellieren und unter Betrachtung eines Angreifers zu formalisieren, sodass der generierte Code ein Refinement der Formalisierung ist.

Während der Entwicklung einer Anwendung werden Annahmen getroffen, unter denen bestimmte Eigenschaften gelten. Bei diesen Eigenschaften kann es sich um die konkreten Sicherheitseigenschaften handeln, aber auch um Hilfseigenschaften, die die Verifikation der Sicherheitseigenschaften erleichtern, da sie das Problem in kleinere Teilprobleme zerlegen. Da der Entwickler die Anwendung so modelliert, dass diese Annahmen und Eigenschaften gelten, soll auch der Entwickler in der Lage sein diese Annahmen und Eigenschaften im Anwendungsmodell zu definieren. Zudem sind dann diese Informationen für jeden verständlich dokumentiert und können beim Deployen des ausführbaren Codes oder beim Verifizieren verwendet werden.

Deshalb ist das Ziel, die Sicherheitseigenschaften sowie initiale Eigenschaften und Hilfseigenschaften im Anwendungsmodell modellieren zu können, sodass diese beim Verifizieren verwendet werden können.

Des Weiteren besitzt nicht jeder Entwickler die Expertise um eine Anwendung interaktiv zu verifizieren. Zudem erschweren Sicherheitslücken in der modellierten Anwendung die interaktive Verifikation erheblich, da nach der Beseitigung einer Sicherheitslücke die Verifikation von neuem begonnen werden kann.

Aus diesem Grund ist das Ziel einen Model Checker zum Finden von Sicherheitslücken in den SecureMDD-Ansatz zu integrieren.

1.3 Erreichte Ergebnisse

Die in Abschnitt 1.2 erwähnten Ziele wurden alle erreicht. Die Ergebnisse sind in diesem Abschnitt zusammengefasst.

Integration und Austauschbarkeit existierender Web Services

Dieses Ziel wurde erreicht indem existierende Web Services schon im Anwendungsmodell integriert und somit während des gesamten Entwicklungsprozesses (Modellierung, Codegenerierung und Verifikation) mit berücksichtigt werden. Zudem können existierende Web Services trotz unterschiedlicher Nachrichtenstrukturen ausgetauscht werden ohne die gezeigten Sicherheitseigenschaften neu zu verifizieren, indem die Sicherheitseigenschaften in Abhängigkeit von Annahmen für einen existierenden Web Service gezeigt werden und anschließend nur noch ein Refinement zwischen den tatsächlichen Zusicherungen eines existierenden Web Services und den genannten Annahmen gezeigt wird. Die erreichten Ergebnisse wurden in [21, 22] publiziert.

Modellierung, Codegenerierung und Formalisierung von Web Services

Auch dieses Ziel wurde erreicht. Es wurde eine Modellierungssprache für sicherheitskritische Web Service-Anwendungen entwickelt, wodurch lauffähiger Web Services-Code, einschließlich seines internen Verhaltens, generiert wird. Web Services sowie deren Sitzungsverwaltung wurde im generierten Code durch stateful Web Services realisiert und im formalen Modell in ASMs und algebraischen Spezifikationen unter Betrachtung eines Angreifers formalisiert. Besonders herausfordernd war die Formalisierung von Serviceaufrufen über Adressen, sowie von kaskadierenden Serviceaufrufen, der Kommunikation über TLS als auch die adäquate Formalisierung des Angreifers. Die erreichten Ergebnisse wurden in [17, 16] publiziert.

Modellierung von anwendungsspezifischen Sicherheitseigenschaften

Dieses Ziel wurde ebenfalls erreicht. Hierzu wurde die Sprachmächtigkeit von OCL genutzt um sicherheitsrelevante Sprachkonstrukte vorzudefinieren und somit anwendungsspezifische Sicherheitseigenschaften modellieren zu können. Diese Sicherheitseigenschaften werden anschließend zusammen mit der modellierten Anwendung in das formale Modell transformiert. Die erreichten Ergebnisse wurden in [19] publiziert.

Integration eines Model Checkers zum Finden von Sicherheitslücken

Dieses Ziel wurde durch die Integration von AVANTSSAR[4] (Automatische Validierung von Trust und Security in serviceorientierten Architekturen) erreicht. Hierzu wurden Transformationen in die formale Spezifikationssprache von AVANTSSAR erstellt. Da der SecureMDD-Ansatz eine lauffähige Anwendung inklusive dem konkreten Verhalten generiert, sind die generierten formalen Spezifikationen sehr groß. Dies führt dazu, dass der Suchraum des Model Checkers ebenfalls zu groß wird und der Model Checker in absehbarer Zeit nicht terminiert bzw. der Speicher überläuft. Somit musste der Suchraum des Model Checkers eingeschränkt werden. Dies wurde durch Abstraktionen erreicht, die die Funktionalität aber auch die Angreiferfähigkeiten der Anwendung einschränken. Damit diese Abstraktionen einfach und übersichtlich für den Entwickler sind, werden diese nicht auf der generierten Spezifikation gemacht, sondern im Anwendungsmodell. Hierzu wurde die Modellierungssprache von SecureMDD um die Modellierung von Abstraktionen erweitert. Die erreichten Ergebnisse wurden in [18, 20] publiziert.

1.4 Aufbau der Arbeit

Diese Arbeit befasst sich mit der modellgetriebenen Entwicklung von sicheren Web Service-Anwendungen und besteht aus neun Kapiteln, die sich wie folgt gliedern lassen.

Kapitel 2 gibt ein Überblick über den SecureMDD-Ansatz und stellt die betrachteten Fallstudien vor. Anschließend wird der Ansatz mit anderen Forschungsarbeiten verglichen.

Kapitel 3 stellt die Modellierung von sicherheitskritischen Smart Card-Anwendungen vor und beschreibt somit die Grundlagen auf denen diese Arbeit aufbaut. Hierzu werden die vordefinierten Datentypen, das UML-Profil sowie die Sprache Model Extension Language (MEL) für die Modellierung der dynamischen Sicht vorgestellt sowie die Modellierung anhand einer Kopierkartenanwendung demonstriert.

Kapitel 4 beschreibt die Modellierung von sicherheitskritischen Web Service-Anwendungen. Hierzu wird in Abschnitt 4.1 zuerst ein UML-Profil vorgestellt, welches UML erweitert, sodass Web Service-Anwendungen modelliert werden können, aber auch die Modellierung von Sicherheitseigenschaften, Abstraktionen und die Kommunikation mit externen (existierenden) Services unterstützt wird. In Abschnitt 4.2 wird die Modellierung der statischen und dynamischen Sicht von Web Services vorgestellt und diese anhand zwei Fallstudien demonstriert. In Abschnitt 4.3 wird die Modellierung von anwendungsspezifischen Sicherheitseigenschaften beschrieben. Hierzu werden OCL-Erweiterungen für Sicherheitseigenschaften vorgestellt und deren Modellierung anhand einem Beispiel demonstriert. In Abschnitt 4.4 werden Abstraktionen vorgestellt, die das Anwendungsmodell einschränken, um Sicherheitslücken auch in großen Anwendungen zu finden. Es werden die Einschränkungen der Protokollabläufe sowie der Angreiferfähigkeiten vorgestellt. In Abschnitt 4.5 wird die Modellierung einer sicheren Integration und Austauschbarkeit von externen Services beschrieben. Hierzu wird die Modellierung der Kommunikation vorgestellt, sowie die Modellierung der Annahmen für einen externen Service. Zum Schluss wird die sichere Kommunikation mit externen Services beschreiben. In Abschnitt 4.6 wird die Modellierung von Web Services, der Sicherheitseigenschaften, der Abstraktionen und der sicheren Integration und Austauschbarkeit von externen Web Services in Bezug zu bestehenden Arbeiten gebracht.

Kapitel 5 beschreibt die Generierung einer lauffähigen Anwendung. Der Fokus liegt auf dem generierten Web Service-Code und was alles beim Deployen des generierten Codes beachtet werden muss. Zum Schluss werden verwandte Arbeiten diskutiert.

Kapitel 6 erläutert die Integration eines Model Checkers in den SecureMDD-Ansatz. Hierfür wird zuerst die Übersetzung eines modellierten Anwendungsmodells in die formale Spezifikation des Model Checkers erläutert und anhand eines Beispiels demonstriert. Anschließend werden automatische Abstraktionen

vorgestellt um die Transformationen zu optimieren und diese zusammen mit den in Abschnitt 4.4 vorgestellten manuellen Abstraktionen evaluiert. Des Weiteren wird gezeigt wie das Ergebnis des Model Checkers zurück in die Sprache der modellierten Anwendung übersetzt wird. Somit kann ein Anwendungsentwickler im Modellierungstool unter Berücksichtigung seiner manuellen Abstraktionen, sein Anwendungsmodell auf Sicherheitslücken überprüfen und erhält anschließend einen Angriffspfad in Bezug zu seinem Anwendungsmodell. Zum Schluss werden noch verwandte Arbeiten diskutiert.

Kapitel 7 beschreibt die interaktive Verifikation von Web Service-Anwendung. Dazu werden zuerst die Grundlagen des formalen Modells erklärt und anschließend die Formalisierung von Web Service-Anwendungen beschrieben. Zudem wird die Übersetzung von OCL ins formale Modell erläutert und die formale Betrachtung von externen Web Service beschrieben. Zum Schluss wird auf die Verifikation von Web Service-Anwendungen eingegangen und verwandte Arbeiten diskutiert.

Kapitel 8 fasst diese Arbeit und deren Ergebnisse zusammen und gibt einen Ausblick über mögliche weiterführende Arbeiten in diesem Bereich.

KAPITEL 2

SecureMDD: Ein modellgetriebener Softwareentwicklungsansatz

2.1 Der SecureMDD-Ansatz im Überlick

SecureMDD ist ein modellgetriebener Softwareentwicklungsansatz für sichere Anwendungen, die auf kryptographischen Protokollen basieren und wurde von der Deutschen Forschungsgemeinschaft (DFG)[1] gefördert. Der im Folgenden vorgestellte SecureMDD-Ansatz beschreibt den Gesamtansatz und enthält die von Moebius [62] erarbeiteten Ergebnisse zur Entwicklung von sicheren Smart Card-Anwendungen sowie die in dieser Arbeit erreichten Ergebnisse zur Entwicklung von sicheren Web Service-Anwendungen (siehe Abschnitt 1.3). Der Fokus des Projekts liegt auf Smart Card- und Web Service-Anwendungen der Domänen E-Commerce und E-Government. Für solche Anwendungen ist es besonders wichtig anwendungsspezifische Eigenschaften unter Berücksichtigung eines Angreifers zu garantieren. Dieses Ziel wird durch eine durchgehende Integration von formalen Methoden in einen modellgetriebenen Softwareentwicklungsansatz erreicht.

Abb. 2.1 zeigt eine Übersicht des Ansatzes. Die Entwicklung einer Anwendung beginnt mit dem Erstellen eines plattformunabhängigen UML-Modells mit der Modellierungssprache SecureML [17, 19, 20, 21]. Dieses Model beschreibt eine abstrakte Sicht auf die Anwendung, indem es Implementierungsdetails weglässt. Um sicherheitskritische Anwendungen modellieren zu können wurde UML durch ein UML-Profil für diese Domäne angepasst. Der statische Teil der Anwendung wird durch ein Klassendiagramm und ein Deploymentdiagramm (Verteilungsdiagramm) modelliert. Im Klassendiagramm werden z.B. Komponenten, deren Attribute sowie die Nachrichten modelliert und ein Deploymentdiagramm modelliert die Kommunikationsstruktur. Für die Modellierung des dynamischen Teils einer Anwendung (die Kommunikationsprotokolle und das Verhalten der Komponenten) wurde im SecureMDD-Projekt eine domänenspezifische Sprache namens Model Extension Langauge (MEL) entwickelt [62, 17], die in UML-Aktivitätsdiagrammen verwen-

[1] www.dfg.de

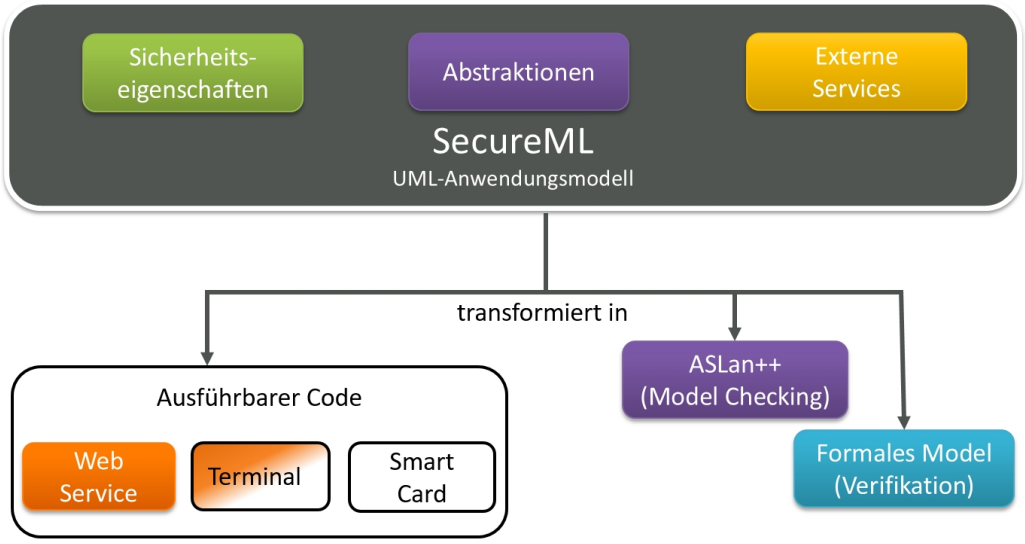

Abbildung 2.1: Übersicht über den SecureMDD-Ansatz

det werden kann. Mit dieser Sprache ist es möglich den Attributen der Komponentenklassen Werte zuzuweisen, neue Objekte zu erstellen und vordefinierte kryptographische Operationen aufzurufen. Das plattformunabhängige UML-Modell einer Anwendung besitzt alle notwendigen Informationen um ausführbaren Programmcode sowie ein formales Modell der gesamten Anwendung automatisch zu generieren.

Um aus dem plattformunabhängige UML-Anwendungsmodell lauffähigen Programmcode zu generieren wird das Model mit Modell-zu-Modell-Transformationen in drei plattformspezifische Modelle (PSMs) transformiert. Somit eins für jeden Komponententyp (Smard Cards, Terminals, Web Services). Web Services sind Softwareprogramme, die eine Schnittstelle besitzen die über ein Netzwerk wie das Internet für andere zugänglich ist. Diese Programme befinden sich in einer sicheren Umgebung (z.B. auf einem Server in einem abgeschlossenen Raum) und können nur über ihre Schnittstelle angesprochen werden. Web Services sind ein wichtiger Bestandteil für viele sicherheitsrelevante Anwendungen wie elektronische Bezahlsysteme, Handelssystemen oder Behördensysteme wie Melderegister oder Strafregister. Smart Cards hingegen sind kleine, sichere und manipulationsgeschützte Geräte, die an Personen ausgegeben werden und mit denen über einen Kartenleser kommuniziert wird. Terminals sind Geräte wie Heim-PCs oder ein Geldautomaten, die von Benutzern zur Eingabe verwendet werden und mit Web Services sowie Smart Cards kommunizieren können. Die plattformspezifischen Modelle enthalten alle relevanten Informationen für einen Komponententyp und fügen technische Implementierungsdetails hinzu. Mit den plattformspezifischen Modellen

als Eingabe und Modell-zu-Text-Transformationen wird ausführbarer Programmcode der modellierten Anwendung automatisch generiert. Die Web Services sind in Java realisiert und verwenden JAX-WS [77] (Java API for XML - Web Services) welches Bestandteil von Java EE ist. Für die Smart Card-Komponenten wird Java Card [48] Programmcode generiert und für Terminals wird Java-Code generiert.

Der SecureMDD-Ansatz generiert für die modellierte Anwendung ein formales Modell, welches auf algebraischen Spezifikationen und ASMs basiert. Dieses Modell kann mit dem interaktiven Theorembeweiser KIV [8] für die Verifikation von anwendungsspezifischen Sicherheitseigenschaften verwendet werden. Die Sicherheitseigenschaften können im SecureML-Anwendungsmodell mit OCL definiert werden. Beispiele für anwendungsspezifische Eigenschaften sind:

- *Ein elektronisches Ticket kann nicht gefälscht werden*
- *In einem elektronischen Bezahlsystem geht kein Geld verloren*

Durch anwendungsspezifische Sicherheitseigenschaften lassen sich viele verschiedene Eigenschaften ausdrücken, die sich durch Standardeigenschaften wie Geheimhaltung, Integrität oder Authentizität nur schwer oder gar nicht beschreiben lassen.

Interaktive Verifikation benötigt oft einen erheblichen Aufwand und wenn Fehler im Protokoll gefunden werden muss die Verifikation mit dem korrigierten Modell von neu angefangen werden. Um Fehler früh und schnell zu erkennen, wird das Anwendungsmodell automatisch nach ASLan++ transformiert. Hierbei handelt es sich um die Eingabesprache des automatischen Protokollanalysierers AVANTSSAR [4]. AVANTSSAR generiert systematisch alle möglichen Pfade der Anwendung für eine feste Anzahl von Komponenten und eine feste Anzahl von Protokollabläufen und überprüft, ob die Sicherheitseigenschaften verletzt werden. Nach eigener Erfahrung [18] können dadurch jedoch nur einfache Fehler schnell gefunden werden. Ein Problem dabei ist, das der Suchraum zu groß wird, sodass AVANTSSAR nicht terminiert oder der Speicher überläuft. Deshalb wurde SecureML um die Modellierung von Abstraktionen erweitert. Dadurch kann der Suchraum eingeschränkt werden, wodurch auch komplexe Fehler in großen Anwendungen gefunden werden können. Der Nachteil ist jedoch, dass durch die Einschränkungen unter Umständen nicht alle Fehler gefunden werden. Nur die interaktive Verifikation beweist die Sicherheit der gesamten Anwendung.

Ein besonders wichtiger Punkt bei der Modellierung von sicheren Anwendungen sind externe (existierende) Komponenten. Hierfür wurde SecureMDD um die Integration von externen Services erweitert. Dadurch kann eine mit SecureMDD entwickelte Anwendung Services von Drittanbietern verwenden, Altsysteme integrieren sowie auf zusätzliche Bibliotheken zugreifen. Externe Services werden auf Modellebene betrachtet und somit auch bei der Codegenerierung sowie im formalen Modell berücksichtigt.

Ein Anwendungsmodell wird mit einem UML-Tool (MagicDraw[2]), welches kompatibel zum Eclipse modeling framework (EMF) ist, erstellt. Alle Transformationen sind mit EMF realisiert. QVT [70] wird für Modell-zu-Modell-Transformationen verwendet und XPand[3] für Modell-zu-Text-Transformationen. Alle Artefakte können mit einem Klick in Eclipse generiert werden. Die Transformationen nach ASLan++, das Ausführen des Model Checkers sowie die Rückübersetzung des Angriffspfades ist als MagicDraw-Plugin realisiert und kann somit während der Modellierung verwendet werden, um Sicherheitslücken in der modellierten Anwendung frühzeitig zu finden.

Der Ansatz von Moebius [62] wurde für Smart Card-Anwendung entwickelt. In dieser Arbeit liegt der Fokus auf Web Services. Hierzu musste die Modellierungssprache sowie die Codegenerierung und das formale Modell erheblich erweitert werden [17, 16]. Zudem kann ein SecureML-Anwendungsmodell nach ASLan++ transformiert werden und mittels Model Checking auf Sicherheitslücken überprüft werden [18]. Des weiteren wurde SecureML um die Modellierung von Sicherheitseigenschaften mit OCL erweitert [19] und es können auf Modellebene Abstraktionen, die für das Model Checking von großen Anwendungen notwendig sind, modelliert werden [20]. Zudem wurde der Ansatz um die Integration von externen Web Services erweitert [21, 22].

2.2 Fallstudien

Um den vorgestellten Ansatz zu demonstrieren, wurden während dieser Arbeit mehrere Web Service-Anwendungen mit SecureMDD modellgetrieben entwickelt. Hierzu zählt eine Onlinebanking-Anwendung um Überweisungen in Echtzeit zu tätigen, ein Smart Card-basiertes Ticketsystem zum Kaufen und Kontrollieren von Fahrkarten sowie eine Debitcard-Anwendung zum Abheben von Geld an einem Bankautomaten und eine OAuth-Anwendung um anderen Anwendungen eingeschränkten Zugriff auf bestimmte Ressourcen zu geben.

2.2.1 Onlinebanking

Onlinebanking ist eine reine Web Service-Anwendung, um Überweisungen von einem Bankkonto auf ein anderes in Echtzeit zu tätigen. Das System unterstützt eine beliebige Anzahl von Web Services und jeder dieser Web Services repräsentiert eine Bank. Somit werden auch Überweisungen von einer Bank an eine andere unterstützt. Eine zu geltende Sicherheitseigenschaft für diese Anwendung ist, dass kein Geld verloren gehen darf. Obwohl das Standardprotokoll TLS zur Absicherung der Web Service-Kommunikation verwendet wird könnte trotzdem Geld verloren gehen, indem ein Angreifer während einer Überweisung zwischen zwei Bankservices

[2] https://www.nomagic.com/products/magicdraw.html
[3] http://projects.eclipse.org/projects/modeling.m2t.xpand

Nachrichten unterdrückt. Aus diesem Grund wurde ein spezielles Protokoll entwickelt, welches alle nicht vollständig ausgeführten Überweisungen zu einem späteren Zeitpunkt nochmal ausführt, solange bis die Überweisung erfolgreich war. Hierbei musste sichergestellt werden, dass der selbe Betrag der zu einem Konto hinzugebucht wurde, von einem anderen Konto abgebucht wird, aber auch das vollständige Überweisungen nicht mehrfach ausgeführt werden. Im Mittelpunkt dieser Fallstudie stand die Überweisungsfunktion, die den sicherheitskritischen Kern darstellt. Die gesamte Anwendung wird detailliert in Kapitel 4.2.3 beschrieben.

2.2.2 ETicket

ETicket ist ein Smart Card-basiertes Ticketsystem für Fahrkarten wie es die Deutsche Bahn verwenden könnte und stellt eine Kombination aus Smart Card- und Web Service-Anwendungen dar. Jeder Kunde erhält eine Smart Card mit der er an einem Computer mit einem Kartenleser Tickets online kaufen kann. Diese Tickets werden auf einem Web Service erstellt und anschließend an die Karte übertragen und dort gespeichert. Beim Kontrollieren übergibt der Fahrgast seine Karte einem Schaffner, der ein Ticket auswählen, kontrollieren und entwerten kann. Zudem haben die Kartenbesitzer die Möglichkeit ihre gespeicherten Tickets zu verwalten. Dieses System garantiert einige Sicherheitseigenschaften für den Betreiber aber auch für die Kunden. Hierzu gehört, dass nur ausgestellte Tickets gestempelt werden, dass Tickets nicht mehrfach gestempelt werden, sowie dass ausgestellte Ticket aufgrund von Unterbrechungen nicht verloren gehen dürfen. Um diese Sicherheitseigenschaften zu garantieren, mussten spezielle Protokolle entwickelt werden. Zum Beispiel konnte nicht einfach TLS für die sichere Nachrichtenübertragung verwendet werden, sondern es musste eine Ende-zu-Ende-Verschlüsselung zwischen einer Karte und dem Ticketservice sichergestellt werden. Zudem musste ein Recovery-Protokoll entwickelt werden mit dem es möglich ist, bezahlte Tickets, die wegen eines böswilligen Angreifers, einer unterbrochenen Internetverbindung oder dem frühzeitigen Entfernen der Karte aus dem Kartenleser nicht auf der Karte gespeichert werden konnten, zu einem anderen Zeitpunkt nochmal abzuholen. Durch diesen Mechanismus sollte es jedoch nicht möglich sein, dass ein Angreifer gekaufte Tickets mehrfach verwenden kann. Der Vorteil gegenüber einem nicht Smart Card-basierten Ticketsystem ist, dass die Kunden anonym bleiben können. Die gesamte Anwendung wird detailliert in Kapitel 4.2.4 beschrieben.

2.2.3 Debitcard

Debitcard ist ein EC-Karten-System. Ein Kunde kann mit seiner Debitcard an einem Bankautomaten Geld abheben, aber auch an einem Service-Terminal Überweisungen tätigen. Diese Anwendung enthält Smart Cards als auch Web Services und unterscheidet zwischen Banken mit Filialen und reinen Onlinebanken. Ein Kunde einer Onlinebank kann ebenfalls Geld an einem Bankautomaten einer anderen Bank abheben. Hierzu ruft der Web Service der Bankfiliale den Web Service

der Onlinebank auf. Im Gegensatz zum Onlinebanking wurde jedoch angenommen, dass die Verbindung zwischen den Bankservices sicher gegenüber einem Angreifer ist und nicht unterbrochen werden kann. Die komplette Fallstudie kann auf der Webseite[4] unter dem Namen Banking System eingesehen werden.

2.2.4 OAuth 2.0

Eine weitere Fallstudie ist OAuth 2.0. Hierbei handelt es sich um ein Protokoll durch das ein Benutzer einer Anwendung (Relying Party) eingeschränkten Zugriff auf bestimmte Ressourcen erlauben kann, die von einem Identity-Provider verwaltet werden. Die Sicherheitseigenschaft, die garantiert werden soll ist, dass ein Angreifer keinen Zugriff auf die Ressourcen erhält. Diese Eigenschaft wurde gebrochen, da die Adresse des Identity-Providers an die Relying Party über eine unsichere Leitung übertragen wurde und somit vom Angreifer verändert werden konnte. Somit hat die Relying Party geheime Informationen anstatt an den Identity-Provider an den Angreifer gesendet und dieser konnte sich mit den Daten vom Identity-Provider die Zugangsdaten zu den Ressourcen abholen. Mithilfe des in SecureMDD integrierten Model Checkers konnte diese Sicherheitslücke gefunden werden. Die Modelle der Anwendungen können im Anhang in Abschnitt 9.1 betrachtet werden.

2.3 Verwandte Arbeiten

Es existieren einige verwandte Arbeiten, die sich mit der Modellierung von sicheren Anwendungen befassen. Ein Überblick kann in [52] und [49] gefunden werden. Im Folgenden werden die für diese Dissertation wichtigsten verwandten Arbeiten vorgestellt und mit diesem Ansatz verglichen.

2.3.1 Schwerpunkt auf kryptographische Sicherheit

UMLSec

Mit UMLSec [51], entwickelt von Jürjens, ist es möglich sicherheitskritische Systeme mit UML zu modellieren und Sicherheitseigenschaften für diese Systeme formal zu verifizieren. Dazu wurde ein UML-Profil definiert mit dem sich eine sicherheitskritische Anwendung modellieren lässt sowie eine Reihe vordefinierter Standardsicherheitseigenschaften formulieren lassen. Beispiele sind Geheimhaltung und Integrität von Daten, die Definition einer rollenbasierten Zugriffskontrolle sowie Informationsflusseigenschaften. Die Anwendung wird in UML-Aktiviätsdiagrammen, Sequenzdiagrammen und Zustandsmaschienen modelliert. Zum Modellieren von kryptographischen Protokolle wurde eine eigene Sprache definiert. Diese ist jedoch nicht Bestandteil der UML-Diagramme. Basierend auf dem Anwendungsmodell

[4] http://www.isse.uni-augsburg.de/projects/reif/securemdd/

kann mit Hilfe des Model-Checkers SPIN [43] sowie verschiedenen automatischen Theorembeweiser überprüft, ob die angegebenen Sicherheitseigenschaften eingehalten werden. Der genannte Ansatz ist jedoch nicht für Web Services ausgelegt. Somit werden wichtige Aspekte dieser Arbeit wie z.B. Serviceadressierung, Sitzungsmanagement oder externe Web Services nicht betrachtet. Zudem wird nur die Kommunikation und nicht das gesamte Verhalten modelliert. Das bedeutet, dass Nachrichten z.B. verschlüsselt und entschlüsselt werden können, jedoch Verzweigungen, die ein Verhalten abhängig vom empfangenen Nachrichteninhalt ermöglichen nicht unterstützt werden. Außerdem werden nur Standartsicherheitseigenschaften betrachtet. SecureMDD erlaubt jedoch die Modellierung und Verifikation von anwendungsspezifischen Sicherheitseigenschaften, die abhängig von einer konkreten Anwendung sind. Meiner Meinung nach geben diese bessere Garantien über die Sicherheit einer Anwendung. Jedoch unterstützt SecureMDD auch die Verifikation von Standardeigenschaften, da diese bei der Verifikation von anwendungsspezifischen Sicherheitseigenschaften in vielen Fällen ebenfalls bewiesen werden müssen. Ein weiterer Punkt in dem sich SecureMDD von UMLSec unterscheidet ist, dass UMLSec anhand der kryptographischen Protokolle keinen lauffähigen Code generiert. Stattdessen wird Code generiert, der die rollenbasierte Zugriffskontrolle sicherstellt.

SecureUML und SecureSOA

Basin et al. [10] stellen einen modellgetriebenen Ansatz namens SecureUML vor, mit dem sicherheitskritische Anwendungen sowie deren Sicherheitsanforderungen mit UML modelliert werden können. Das Anwendungsgebiet beschränkt sich jedoch auf die Betrachtung rollenbasierter Zugriffskontrolle, die um dynamische Aspekte, wie zum Beispiel Datumsstempel, erweitert werden. Es ist möglich aus den Modellen automatisch Access Control Infrastrukturen zu generieren. Die betrachteten Zielplattformen sind Enterprise Java Beans, Enterprise Services für dotNet sowie Java Servlets. Kryptographische Protokolle, die sicher gegenüber einem Angreifer sind, werden nicht betrachtet.

Eine Erweiterung des Ansatzes stellt jedoch Menzel [59] vor. Diese erweitert SecureUML um die Modellierung und Formalisierung von Sicherheitsentwurfsmustern für serviceorientierten Architekturen (SecureSOA). Die Sicherheitsentwurfsmuster dienen als Grundlage zur Generierung von WS-Policies. Durch die formale Verifikation des Transformationsprozesses von Entwurfsmustern wird die Korrektheit der Entwurfsmusteranwendung sichergestellt. Die Sicherheitseigenschaften sind jedoch nicht anwendungsspezifisch und wurden unabhängig vom konkreten Verhalten der Anwendung sichergestellt.

In beiden Fällen wird nicht das gesamte Verhalten der Anwendung modelliert, weshalb auch kein lauffähiger Web Service-Code generiert wird. Außerdem werden keine externen Web Services betrachtet, deren Verhalten die Sicherheitseigenschaften beeinflussen oder die ausgetauscht werden können ohne das die Sicherheitseigenschaften verletzt werden.

MDD4SOA

Ein weiterer modellgetriebener Ansatz für serviceorientierten Architekturen ist MDD4SOA [56]. Aus einem Modell, welches UML, SoaML[5] und ein eigenes UML Profil (UML4SOA [57]) benutzt, wird automatisch ein Service Meta-Modell erzeugt und aus diesem anschließend Code für die Sprachen BPEL, WSDL und Java. Der Focus von MDD4SOA ist jedoch Serviceorchestrierung. Deshalb wird aus dem Service Meta-Modell eine Spezifikation generiert zum formalen Analysieren von Eigenschaften wie Deadlock-Freiheit. Die formale Analyse erfolgt mit MIO[6]. Dieser Ansatz betrachtet jedoch nicht die Sicherheit von Serviceanwendungen gegenüber einem Angreifer.

Eine Erweiterung des Ansatzes um nicht funktionale Eigenschaften wie Sicherheit, aber auch Performance und Zuverlässigkeit stellen Foster et al. [33] vor. Diese Eigenschaften sind in der Arbeit von Gilmore et al. [34] im Detail beschrieben. Es wird modelliert welcher Teil einer Nachricht verschlüsselt bzw. signiert werden soll und ob Zeitstempel und Authentisierungstokens verwendet werden. Diese Annotationen werden anschließend verwendet um WS-Policies zu generieren. Sicherheitseigenschaften werden jedoch nicht verifiziert. Des weiteren wird nicht das gesamte sicherheitsrelevante Verhalten der Services modelliert. Es werden z.B. keine Operationen auf Datenbanken oder arithmetische Operationen unterstützt, die jedoch für viele Sicherheitseigenschaften im Business Bereich notwendig sind.

2.3.2 Weitere Ansätze mit Schwerpunkt auf Web Services

Diese Ansätze lassen sich wiederum in die Bereiche Serviceorchestrierung und Access Control unterteilen.

Serviceorchestrierung

Lie et al. [54] entwickelten einen modellgetriebenen Ansatz zum Sicherstellen von validen Servicekompositionen. Zum Modellieren werden UML-Sequenzdiagramme verwendet, die in Statecharts und anschließend in die Eingabesprache von Modellcheckern wie SMV [15] und SPIN [43] transformiert werden. Dadurch können Konsistenzüberprüfungen automatisch durchgeführt werden. Zum Schluss wird BPEL-Code generiert. Biana et al. [7] verwenden Zustandsmaschinen zum Beschreiben von Servicekommunikationen und generieren BPEL-basierte Servicestrukturen, welche die Konversationslogik implementieren. Gronmo et al. [36] importiert Webservice Beschreibungen in UML, kombiniert diese und generiert daraus neue Webservice Beschreibungen. Weitere Ansätze, deren Schwerpunkt jedoch nur auf der Verifikation von Serviceorchestrierung liegt, sind [9][82][64].

Die erwähnten Arbeiten betrachten Serviceorchestrierung bzw. Servicekomposition, befassen sich jedoch weder mit dem gesamten Verhalten von Services noch mit der Sicherheit gegenüber einem Angreifer.

[5] http://www.omg.org/spec/SoaML/
[6] http://www.miowb.net/

Access Control
Der zweite große Bereich befasst sich mit der Modellierung und anschließender Generierung von Access Control Policies. SecureBPMN von Brucker et al. [27] ist ebenfalls ein modellgetriebener Ansatz mit dem es möglich ist Access Conrtol-Eigenschaften in Geschäftsprozessen durch eine Erweiterung von BPMN zu Modellierung und mithilfe von Extended Access Control Markup Language (XACML) Policies zu erzwingen. Neben klassischem Role Based Access Control werden zusätzlich die Sicherheitseigenschaften "Separation and Binding of Duty" sowie "Need to Know" (Zugriff nur auf Informationen, die für eine bestimmte Aufgabe notwendig sind) betrachtet. Alam und Breu befassen sich in [2, 1] mit der modellgetriebenen Entwickelung von Service Sicherheitsinfrastrukturen. Ihr Fokus liegt auf rollen-basierter Zugangskontrolle. Sie verwenden OCL um daraus XACML Policies für Webservices zu generieren. Eine Weiterentwicklung der Arbeit wird in [25] vorgestellt.

Der Fokus dieser Dissertation liegt nicht auf der Modellierung und Verifikation von Access Control-Eigenschaften oder der Generierung von Access Control-Policies, sondern auf der Entwicklung von sicheren Web Service-Anwendungen und der Sicherstellung von anwendungsspezifischen Eigenschaften, die Access Control-Eigenschaften implizieren, jedoch über diese hinausgehen.

KAPITEL 3

Modellierung von sicherheitskritischen Smart Card-Anwendungen

SecureMDD unterstützt die Modellierung von sicherheitskritischen Smart Card- und Web Service-Anwendungen. Die Entwicklung von sicheren Smart Card-Anwendungen wurde in einer eigenständigen Arbeit von N. Moebius vorgestellt [62]. Um die Modellierung von sicherheitskritischen Web Service-Anwendungen verstehen zu können, muss zuerst die dort verwendete Modellierungssprache kurz vorgestellt werden. Sie besteht aus vordefinierten Datentypen, die in primitive und kryptographische Datentypen unterteilt werden, einem UML-Profil, welches Stereotypen definiert, die das UML-Metamodel erweitern sowie einer plattformunabhängigen Sprache MEL und den Modellierungsrichtlinien für drei unterstützte Diagrammtypen. Zum Modellieren des statischen Teils werden Klassendiagramme und ein Deploymentdiagramm benutzt. Das Klassendiagramm verwendet die vordefinierten Datentypen sowie Stereotypen aus dem UML-Profil. Zum Modellieren des dynamischen Teils werden Aktivitätsdiagramme, die eine Abhängigkeit zum statischen Teil besitzen sowie die Sprache MEL verwendet.

3.1 Vordefinierte Datentypen

Mithilfe von vordefinierten Datentypen ist die Modellierungssprache unabhängig von UML-Datentypen und unterstützt zudem kryptographische Daten. Diese Datentypen werden als Modul in ein SecureMDD-Modell importiert. Dieses Modul enthält ein Klassendiagramm, in dem primitive sowie kryptographische Datentypen enthalten sind. Die primitiven Datentypen sind:

1. Number
2. String
3. Boolean

Durch *Number* können nummerische Werte (z.B. Kontostände) gespeichert werden. *String* enthält eine Zeichenfolge (z.B. Name) und *Boolean* enthält

Wahrheitswerte.

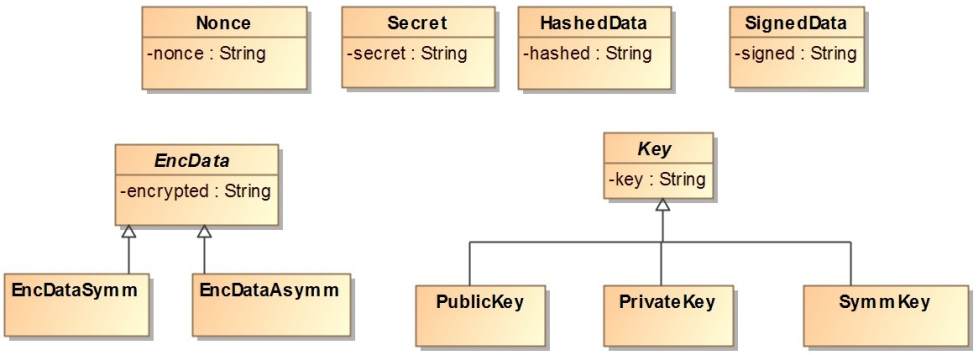

Abbildung 3.1: Kryptographische Datentypen [62]

Um kryptographische Protokolle modellieren zu können werden Sicherheitsdatentypen mithilfe von Klassen vordefiniert. Abb. 3.1 zeigt diese Sicherheitsdatentypen. *Nonce* repräsentiert eine zufällige und einmalige Zeichenfolge, die von einem Angreifer nicht erraten werden kann. Diese wird in kryptographischen Protokollen verwenden um Replay-Attacken zu verhindern. *Secret* ist eine Zeichenfolge die ein Geheimnis darstellt und somit gegenüber einem Angreifer unbekannt ist und bleiben soll. Dieser Datentyp wird verwendet um z.B. eine PIN oder ein Passwort zu modellieren. *HashedData* stellt einen Hashwert dar. Dieser entsteht durch eine Einwegfunktion, bei der eine große Eingabemenge auf eine kleinere Zielmenge abgebildet wird. Dadurch kann man z.B. zeigen, dass man ein Geheimnis kennt ohne das Geheimnis zu verraten. *EncData* ist eine Oberklasse für verschlüsselte Daten. Es wird zwischen *EncDataSymm* (symmetrisch verschlüsselten Daten) und *EncDataAsymm* (asymmetrisch verschlüsselten Daten) unterschieden. Verschlüsselte Daten können nur mit dem dazugehörigen Schlüssel entschlüsselt werden und sind somit bestens geeignet um vertrauliche Informationen zu übertragen. Symmetrisch verschlüsselte Daten werden mit demselben Schlüssel entschlüsselt mit dem sie verschlüsselt wurden. Asymmetrisch verschlüsselte Daten werden mit einem öffentlichen Schlüssel verschlüsselt und mit einem privaten Schlüssel entschlüsselt. *Key* ist eine Oberklasse für die verschiedenen Arten von Schlüsseln. Dabei wird zwischen *PublicKey* und *PrivateKey* für die asymmetrische Verschlüsselung und *SymmKey* für die symmetrische Verschlüsselung unterschieden. *SignedData* enthält eine Signatur. Mithilfe von Signaturen lässt sich die Authentizität und Integrität der Daten überprüfen, und somit, von wem die Daten stammen und ob sie verändert wurden. Eine Signatur wird mit dem privaten Schlüssel des Erzeugers erstellt und mit dem öffentlichen Schlüssel des Erzeugers verifiziert.

Die Datentypen *Nonce*, *Secret* und alle Unterklassen von *Key* können direkt als Typen von Klassenattributen verwendet werden. Für die Modellierung von ver-

schlüsselten Daten, Signaturen, Hashes und Zertifikaten werden zusätzliche Stereotypen im UML-Profil definiert (siehe Abschnitt 3.2).

3.2 UML-Profil

Das UML-Profil von SecureMDD enthält eine Vielzahl von Stereotypen um das UML-Metamodel zu erweitern. Es werden nur die Stereotypen erwähnt, die zum Verständnis dieser Arbeit notwendig sind. Diese Stereotypen können in folgende Kategorien strukturiert werden: Komponenten, Nachrichten, Protokollmodellierung, Angreifer und kryptographische Daten.

Komponenten

Um eine Anwendung zu modellieren, müssen die beteiligten Komponenten des Systems definiert werden. Dazu werden im UML-Profil die folgenden drei Stereotypen definiert:

- ≪User≫
- ≪Terminal≫
- ≪Smartcard≫

Diese Stereotypen können auf die Metaklasse *Class* angewendet werden, um eine solche Komponente zu modellieren. ≪User≫ repräsentiert reale Personen, welche das modellierte System verwenden. Die Schnittstelle zwischen den Benutzern und dem technischen System erfolgt über Komponenten mit dem Stereotyp ≪Terminal≫. Dabei handelt es sich um sichere Eingabegeräte wie z.B. einen Geldautomaten. Ein solches Eingabegerät kann Chipkartenleser besitzen, wodurch es mit Komponenten vom Typ ≪Smartcard≫ kommunizieren kann. Smart Cards repräsentieren Chipkarten.

Nachrichten

Für die Übertragung der Nachrichten zwischen den Komponenten werden die folgenden Nachrichtentypen definiert.

- ≪Message≫
- ≪Usermessage≫

Diese Stereotypen sind ebenfalls für die Metaklasse *Class* definiert. Als ≪Usermessage≫ werden die Nachrichten, die zwischen ≪User≫ und ≪Terminal≫ ausgetauscht werden bezeichnet. ≪Message≫ repräsentiert die Nachrichten zwischen ≪Terminal≫ und ≪Smartcard≫. Diese Stereotypen werden auf Klassen angewendet, die dann als Nachrichten zwischen den jeweiligen Komponenten ausgetauscht werden können.

Protokollmodellierung

Zum Charakterisieren der Komponentenattribute existieren zwei weitere Stereotypen, die auf die Objekte der Metaklasse *Property* angewendet werden können.

- ≪Initialize≫
- ≪status≫

Mit ≪Initialize≫ annotierte Attribute müssen bei der Instanziierung der Komponentenklassen übergeben werden. Somit können Chipkarten mit Namen, Adressen aber auch kryptographischen Schlüsseln vorinitialisiert werden. ≪Status≫ wird auf Attribute vom Typ *Enumeration* angewendet, um den Zustand einer Komponente zu modellieren. Dies ist wichtig, damit eine bestimmte Nachrichtenreihenfolge sichergestellt werden kann.

Angreifer

Des weiteren wurde der Stereotyp ≪Threat≫ eingeführt, um die Angreiferfähigkeiten auf den Kommunikationskanälen zwischen den Komponenten zu definieren. ≪Threat≫ ist für die Metaklasse *CommunicationPath* definiert und besitzt die drei Eigenschaften *read*, *send* und *suppress* vom Typ *boolean*. Somit kann ein Angreifer auf verschiedenen Kommunikationskanälen jeweils eine Teilmenge dieser Angreiferfähigkeiten besitzen.

Kryptographische Daten

Mithilfe der folgenden Stereotypen kann modelliert werden, welche Klassenattribute verschlüsselt, signiert oder gehasht sind bzw. ob es sich um ein Zertifikat handelt.

- ≪PlainData≫ (Daten können verschlüsselt werden)
  - ≪encrypted≫ (symmetrisch verschlüsselte Daten)
  - ≪encryptedAsymm≫ (asymmetrisch verschlüsselte Daten)
- ≪SignData≫ (Daten können signiert werden)
  - ≪signed≫ (signierte Daten)
- ≪HashData≫ (Daten können gehasht werden)
  - ≪hashed≫ (gehashte Daten)
- ≪Certificate≫ (beschreibt ein Zertifikat)

≪PlainData≫, ≪SignData≫, ≪HashData≫ und ≪Certificate≫ sind für die Metaklasse *Class* definiert und ≪encrypted≫, ≪encryptedAsymm≫, ≪signed≫ und ≪hashed≫ für die Metaklasse *Property*.

Mit ≪PlainData≫ annotierte Klassen können symmetrisch oder asymmetrisch verschlüsselt werden. Ob Klassenattribute vom Typ einer solchen Klasse verschlüsselt sind, wird durch den Stereotypen ≪encrpyted≫ für symmetrische Verschlüsselung oder durch ≪encryptedAsymm≫ für asymmetrische Verschlüsselung modelliert. Durch die Kombination aus ≪SignData≫ und ≪signed≫ werden signierte Attribute modelliert und mit ≪HashData≫ und ≪hashed≫ gehashte Attribute. Die Stereotypen ≪PlainData≫, ≪SignData≫ und ≪HashData≫ können durch die Anwendung auf die selbe Klasse kombiniert werden. Dadurch können verschiedene Attribute des selben Typs unterschiedliche kryptographische Daten darstellen.

Eine mit ≪Certificate≫ annotierte Klasse stellt ein Zertifikat dar. Dieses wird verwendet um öffentliche Schlüssel zu verteilen. Ein Zertifikat enthält normalerweise einen öffentlichen Schlüssel, den Namen des Besitzer sowie mögliche weitere Informationen in Klartext sowie in signierter Form. Ein solches Zertifikat wird meistens von einer vertrauenswürdigen Stelle mittels eines geheimen privaten Schlüssels erstellt und kann mit einem öffentlichen Schlüssel auf seine Korrektheit überprüft werden. Somit wird sichergestellt, dass nur die vertrauenswürdige Stelle ein solches Zertifikat ausstellen kann.

3.3 Model Extension Language

Model Extension Language (MEL) ist eine plattforumunabhängige Sprache, die für die Domäne von kryptographischen Protokollen entwickelt wurde. Sie verwendet eine textuelle Notation innerhalb von UML-Elementen, welche in Aktivitätsdiagrammen vorkommen. Hierdurch kann nicht nur das Übertragen der Nachrichten, sondern auch die Verarbeitung der Nachrichten modelliert werden. Die Syntax und Semantik der Sprache wird in [62] ausführlich vorgestellt. Im folgenden wird nur ein kurzer Überblick gegeben, um die vorgestellten Fallstudien sowie die Modellierung von Web Services besser zu verstehen.

Senden und Empfangen von Nachrichten

Das Senden einer Nachricht geschieht in einer *SendSignalAction* und das Empfangen in einer *AcceptEventAction*. Innerhalb dieser Elemente wird eine übertragene Nachricht durch den Konstruktoraufruf einer im Klassendiagramm modellierten Nachrichtenklasse definiert. Bei einer Nachrichtenklasse *Msg* mit den Attributen *att1, att2* wird der MEL-Ausdruck *Msg(att1,att2)* unterstützt. Beim Senden können die Parameter lokale Variablen oder Attribute der Komponentenklasse sein. Beim Empfangen der Nachricht können die Nachrichtenattribute als lokale Variablen verwendet werden.

Datentypen

MEL unterstützt zwei Arten von Datentypen. Die in Abschnitt 3.1 vorgestellten vordefinierten Datentypen sowie die selbst modellierte Klassen, die als Datentypen von Attributen verwendet werden.

Lokale Variablen und Zuweisungen

Lokale Variablen werden durch den MEL-Ausdruck *var : Type* deklariert. Dabei ist *var* ein beliebiger Variablenname und *Typ* einer der unterstützten Datentypen. Eine Zuweisung wird durch den MEL-Ausdruck $var := x$ beschrieben, wobei x der zuzuweisende Wert ist und vom gleichen Typ sein muss. Dabei ist es egal ob *var* bzw. x lokale Variablen oder Klassenattribute sind. Um eine Variable zu deklarieren und ihr gleichzeitig einen Wert zuzuweisen kann der MEL-Ausdruck $var : Type := x$ verwendet werden. Variablendeklarationen sowie Zuweisungen sind in einer *Action* möglich.

Erzeugen von Objekten

Das Erzeugen von Objekten wird durch den MEL-Ausdruck *create Clas(att1, ..., attn)* beschrieben. Dabei ist *Clas* ein im Klassendiagramm modellierter Datentyp. Diese Objekte können wiederum Klassenattributen und lokalen Variablen zugewiesen werden. Primitive Werte wie *(Number, String und Boolean)* müssen nicht mittels *create* erzeugt werden. *Number* wird durch nicht negative Zahlen erzeugt, *String* durch eine Zeichenfolge in Anführungszeichen und *Boolean* durch die Werte *true* und *false*. Kryptographische Datentypen werden durch vordefinierte Operationen erzeugt. Das erzeugen von Objekten ist in einer *Action* möglich.

Modellierung von bedingten Verzweigungen

Bedingte Verzweigungen werden durch einen *DecisionNode* modelliert. Die *Guards* der ausgehenden Kanten enthalten die Bedingung, dessen Auswertung einem Wahrheitswert entsprechen muss. Einer der *Guards* muss das Schlüsselwort *else* enthalten.

Vordefinierte Operationen

MEL unterstützt eine Reihe an vordefinierten Operationen. Diese lassen sich in unäre und binäre Operationen sowie Listenoperationen und kryptographische Operationen unterteilen.

- **Unäre und Binäre Operationen**
  Zum inkrementieren bzw. dekrementieren von Zahlenwerten sind die Operationen $++$ und $--$ auf den primitiven Typ *Number* definiert und durch die *not*-Operation werden boolesche Ausdrücke negiert. Zudem werden die

binären Vergleichsoperationen *==, !=, <, >, <=, >=* sowie die Operationen *+, -, *, /, %* für den primitiven Typ *Number* definiert. Für boolesche Ausdrücke werden zudem *and* und *or* unterstützt. Die Vergleichsoperatoren *==* und *!=* werden auch für Objekte bzw. primitive Werte unterstützt. Dazu müssen diese vom selben Typ sein. Listen können nicht verglichen werden.

- **Listenoperationen**
  Listen werden als Attribute mit einer Multiplizität größer 1 modelliert. Diese Listen besitzen folgende vordefinierte Operationen:

  - *add(Element e) : void*
    Dadurch wird ein Element *e* an das Ende der Liste hinzugefügt.
  - *remove(Element e) : void*
    Entfernt das erste Element *e* aus der Liste.
  - *at(Number i) : Element*
    Gibt das Element an der Stelle *i* zurück. Falls keins vorhanden ist, wird eine Exception geworfen.
  - *hasFree() : Boolean*
    Gibt an ob Elemente zu der Liste noch hinzugefügt werden können.
  - *size() : Number*
    Gibt die Anzahl der gespeicherten Elemente zurück.

- **Kryptographische Operationen**
  Zum Erstellen der in Abschnitt 3.1 definierten kryptographischen Datentypen werden die folgenden Operationen unterstützt:

  - *generateNonce() : Nonce*
    Generiert eine bisher noch nicht verwendete zufällige Zahl (Nonce).
  - *generateKey() : SymmKey*
    Generiert einen symmetrischen Schlüssel.
  - *encrypt(Symmkey sk, PlainData p) : EncDataSymm*
    Symmetrische Verschlüsselung von *p* mit dem symmetrischen Schlüssel *sk*.
  - *encrypt(PublicKey pk, PlainData p) : EncDataAsymm*
    Asymmetrische Verschlüsselung von *p* mit dem öffentlichen Schlüssel *pk*.
  - *decrypt(Symmkey sk, EncDataSymm eds) : PlainData*
    Entschlüsselt ein symmetrisch verschlüsseltes Objekts *eds* mit dem symmetrischen Schlüssel *sk*.
  - *decrypt(PrivateKey privk, EncDataAsymm edas) : PlainData*
    Entschlüsselt eine asymmetrisch verschlüsseltes Objekts *edas* mit dem privaten Schlüssel *privk*.

- *hash(HashData h) : HashedData*
  Erstellt einen Hash-Wert vom Objekt *h*.
- *sign(PrivateKey privk, SignData signd) : SignedData*
  Signiert das Objekt *signd* mit dem privaten Schlüssel *privk*.
- *verify(PublicKey pubk, SignedData signature, SignData signd) : Boolean*
  Verifiziert die Signatur *signature* anhand eines öffentlichen Schlüssels *pubk* und den in der Signatur enthaltenen Daten *signd*.
- *generateCertificate(PrivateKey issuerprivkey, Type_Att_1 att_1, ..., Type_Att_n att_n) : C*
  Generiert ein Zertifikat mit den den Attributen des Zertifikats *Type_Att_1 att_1, ..., Type_Att_n att_n*.
  C steht für jede Klasse die mit dem Stereotyp ≪Certifikate≫ annotiert ist.
- *verifyCertificate(PublicKey key) : Boolean*
  Wird auf einer Zertifikatklasse C aufgerufen und verifiziert anhand eines öffentlichen Schlüssels das Zertifikat.

3.4 Fallstudie Kopierkartenanwendung

In diesem Abschnitt wird die Modellierung von Smart Card-Anwendungen anhand einer Kopierkartenanwendung aus [62] kurz vorgestellt. Um Kopien anzufertigen benötigt man eine Kopierkarte. Diese kann an einem Ladeautomaten aufgeladen und später zum Bezahlen der Kopien an einem Kopiergerät verwendet werden. Die Anwendung besteht aus vier Komponenten. Dabei kann es sich um technische Komponenten wie die Kopierkarte, den Ladeautomat und das Kopiergerät oder um reale Personen wie den Kartenbesitzer handeln. Es wird eine beliebige Anzahl an Personen betrachtet und jeder dieser technischen Komponenten kann beliebig oft instanziiert werden. Somit wird nicht nur eine Kopierkarte, sondern eine Vielzahl von Kopierkarten in diesem System betrachtet.

Statische Sicht

Abb. 3.2 zeigt das Deploymentdiagramm der Kopierkartenanwendung. Es beschreibt zwischen welchen Komponenten Nachrichten ausgetauscht werden können. Die Komponenten werden als *Nodes* mit dem entsprechenden Stereotypen modelliert. Der Kartenbesitzer (*CardOwner*) wird mit dem Stereotyp ≪User≫ annotiert, der Ladeautomat (*DepositMachine*) und das Kopiergerät (*CopyingMachine*) erhalten den Stereotyp ≪Terminal≫ und die Kopierkarte (*Copycard*) verwendet den Stereotypen (*Smartcard*). Zwischen den *Nodes* werden die Kommunikationskanäle mittels gerichteten *CommunicationPaths* dargestellt. Die Richtung beschreibt wer eine Kommunikation anfangen darf. Eine Kommunikation fängt

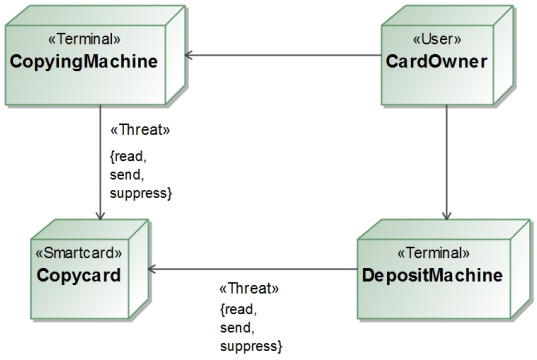

Abbildung 3.2: Deploymentdiagramm für die Kopierkartenanwendung [62]

immer beim Benutzer an. Somit kann der *CardOwner* einen Kopierauftrag an *CopyingMachine* oder einen Aufladebefehl an *DepositMachine* senden. Anschließend kann ein solches *Terminal* eine *Copycard* aufrufen, um den auf der Karte gespeicherten Betrag zu verändern. Eine aufgerufene Komponente wie z.B. *Copycard* kann zwar keine andere Komponente aufrufen, aber auf einen Aufruf mit einer Rücknachricht antworten. Durch den Stereotypen ≪Threat≫ werden die Angreiferfähigkeiten auf den Kommunikationskanälen modelliert. Die Kopierkartenanwendung betrachtet einen Angreifer, der zwischen dem Ladeautomaten und der Kopierkarte sowie zwischen dem Kopiergerät und der Kopierkarte die Fähigkeiten besitzt Nachrichten zu lesen, zu senden und zu unterdrücken.

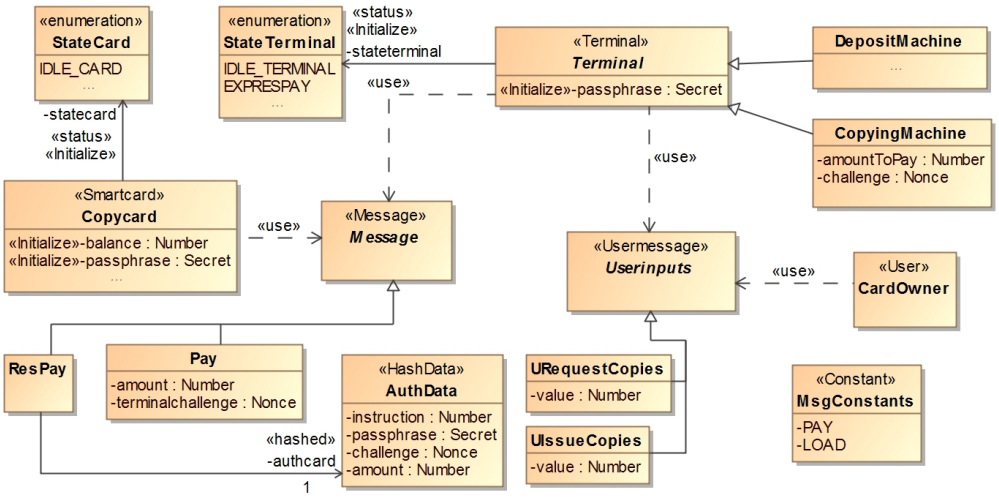

Abbildung 3.3: Klassendiagramm für die Kopierkartenanwendung [62]

Abb. 3.3 zeigt ein Teil des Klassendiagramms, welches die Komponenten, de-

ren Attribute sowie die übertragenen Nachrichten modelliert. Die Komponenten werden durch Klassen mit den Stereotypen ≪User≫, ≪Terminal≫ oder ≪Smartcard≫ modelliert. Diese Klassen können Attribute besitzen, welche als eigene Attribute bzw. als gerichtete Assoziationen mit Rollennamen modelliert sind. Die Kopierkarte *Copycard* besitzt unter anderem einen Geldbetrag *balance* vom Typ *Number* sowie ein Geheimnis *passphrase* vom Typ *Secret*. Bei den Datentypen handelt es sich um vordefinierte Datentypen, welche im Abschnitt 3.1 vorgestellt wurden. Ein weiteres Attribut von *Copycard* ist der Zustand *statecard* vom Typ *StateCard*. Dabei handelt es sich um eine Enumeration mit Literalen wie z.B. *IDLE_CARD*. Der Zustand ist als gerichtete Assoziation mit dem Rollennamen *statecard* sowie dem Stereotyp ≪status≫ modelliert. Die Zustände werden für die Modellierung der Protokolle verwendet, damit definiert werden kann in welchem Protokollschritt sich eine Komponente gerade befindet und welche Nachrichten sie akzeptiert. Die Attribute *balance*, *passphrase* und *statecard* besitzen den Stereotyp ≪Initialize≫, weshalb beim Initialisieren der Karte diesen Attributen Werte übergeben werden müssen. Der Ladeautomat *DepositMachine* und das Kopiergerät *CopyingMachine* erben von der abstrakten Oberklasse *Terminal*. Diese Oberklasse besitzt ebenfalls einen Zustand *stateterminal* sowie eine *passphrase* als gemeinsames Geheimnis zwischen allen Kopierkarten, allen Ladeautomaten und allen Kopiergeräten. Das Kopiergerät *CopyingMachine* besitzt zusätzlich zu den zwei Attributen aus der Oberklasse ein Attribute *amountToPay* um sich den zu zahlenden Betrag zu merken sowie einen zufälligen Wert *challenge* vom Typ *Nonce*, welcher für die Protokollmodellierung wichtig ist. Die konkreten Nachrichten, die zwischen Terminals und Smartcards ausgetauscht werden, erben von einer abstrakte Oberklasse mit dem Stereotyp ≪Message≫. Die Beziehung zwischen den übertragenen Nachrichtenklassen und den Komponenten wird durch *use*-Beziehungen verdeutlicht. Die Nachrichtenklassen können genauso wie die Komponenten eigene Attribute bzw. gerichtete Assoziationen mit Rollennamen besitzen, welche die Nachrichtenparameter darstellen. Abb. 3.3 zeigt nur die Nachrichten, die für das Kopieren notwendig sind. Die Nachricht *Pay* enthält den Betrag *amount*, der von der Kopierkarte abgezogen werden soll sowie eine *terminalchallenge*. Die Nachricht *ResPay* besitzt einen Hashwert *authcard* von einem *AuthData*-Objekt, welches wieder mehrere Attribute hat. Ein Klassenattribut, das gehasht werden soll wird wie in in Abschnitt 3.2 beschrieben mit den Stereotypen ≪HashData≫ und ≪hashed≫ modelliert. Die Konstanten *PAY* und *LOAD*, welche als Attribute einer Klasse mit dem Stereotyp ≪Constant≫ modelliert sind, können als *instruction* eines *AuthData*-Objekt verwendet werden. Dadurch kann sichergestellt werden, dass ein gehashtes *AuthData*-Objekt nicht zwischen den verschiedenen Protokollen ausgetauscht werden kann. Die Benutzernachrichten erben ebenfalls von einer abstrakten Oberklasse mit dem Stereotyp *Usermessage*. Sie werden zwischen Kartenbesitzern (*CardOwner*) und den Terminals *CopyingMachine* und *DepositMachine* ausgetauscht. Für das Kopieren werden die zwei Benutzernachrichten *URequestCopies* (Kopierauftrag) sowie *UIssueCopies* (Rückantwort) mit jeweils der zu kopierenden

Menge definiert.

Dynamische Sicht

Abb. 3.4 zeigt ein Aktivitätsdiagramm, welches das Kopierprotokoll beschreibt. Ein Protokoll enthält mehrere Komponenten, die Nachrichten austauschen um eine Funktionalität des Systems zu realisieren. Der Anfang eines Protokolls wird durch einen *InitialNode* (1) in der *Swimlane* des Benutzers modelliert und das Ende des Protokolls wird durch einen *AktivityFinalNode* (2) in der selben *Swimlane* modelliert. Das Senden von Nachrichten zu anderen Komponenten wird durch eine *SendSignalAction* (3) und das Empfangen von Nachrichten durch eine *AcceptEventAction* (4) modelliert. Die übertragenen Nachrichten sind im Klassendiagramm als Nachrichtenklassen definiert. Welche Nachricht übertragen wird, ist durch den Konstruktor einer Nachrichtenklasse im *SendSignalAction* und *AcceptEventAction* definiert. Diesem Konstruktor werden die Nachrichtenparameter übergeben. Nach jedem Empfangen einer Nachricht kann mithilfe von MEL das Verhalten der Komponenten in Abhängigkeit der übertragenen Nachricht definiert werden. Nachdem das Kopiergerät (*CopyingMachine*) die Nachricht *URequestCopies* mit der Anzahl der zu tätigenden Kopien (*val*) erhalten hat (4), wird überprüft, ob der Wert größer Null ist (5). Falls dies nicht der Fall ist, wird das Protokoll abgebrochen. Die Verzweigung des Protokolls wird durch einen *DecisionNode* modelliert und die Überprüfungen durch *Guards* an den ausgehenden *ControlFlows*. Der Protokollabbruch wird durch einen *FlowFinalNode* repräsentiert. Falls der Wert größer Null ist, wird er in der Klassenvariable *amountToPay* von *CopyingMachine* zwischengespeichert (6). Anschließend wird ein zufälliger Wert durch die vordefinierte Operation *generateNonce* erstellt und der Klassenvariable *challenge* zugewiesen (7). Zum Schluss wird die Zustandsvariable *stateterminal* auf den Wert *EXPRESPAY* gesetzt (8) und eine Nachricht *Pay* mit der Kopienanzahl sowie der generierten Zufallszahl an die Kopierkarte gesendet (9). Die Kopierkarte empfängt die Nachricht und setzt ihren Zustand (10). Wenn die Anzahl der zu tätigenden Kopien (*value*) negativ ist oder das Kartenguthaben (*balance*) für die Kopien nicht ausreicht, wird das Protokoll abgebrochen (11). Ansonsten wird der zu kopierende Betrag vom Kartenguthaben abgezogen (12) und ein *AuthData*-Objekt erstellt und in der lokalen Variable *tmp* gespeichert. Das Objekt enthält die Konstante *PAY*, um dieses Objekt dem Kopierprotokoll eindeutig zuzuordnen. Zudem enthält das Objekt das Geheimnis *passphrase* (welches ein Angreifer nicht erfahren darf), die vom Kopiergerät generierte Zufallszahl, sowie die Anzahl der Kopien (13). Anschließend wird das Objekt mit der vordefinierten Operation *hash* gehasht und der lokalen Variable *authhashed* vom Typ *HashedData* zugewiesen (14). Zum Schluss wird der Hashwert in der Nachricht *ResPay* an das Kopiergerät zurückgeschickt (15). Nachdem die Nachricht *ResPay* vom Kopiergerät empfangen wurde, wird überprüft, ob das Kopiergerät im richtigen Zustand ist (16). Wenn dies der Fall ist, wird der Zustand wieder auf den Ausgangszustand zurückgesetzt (17). Ansonsten wird das Protokoll abgebrochen. Dadurch wird sichergestellt, dass die Protokollschritte nur

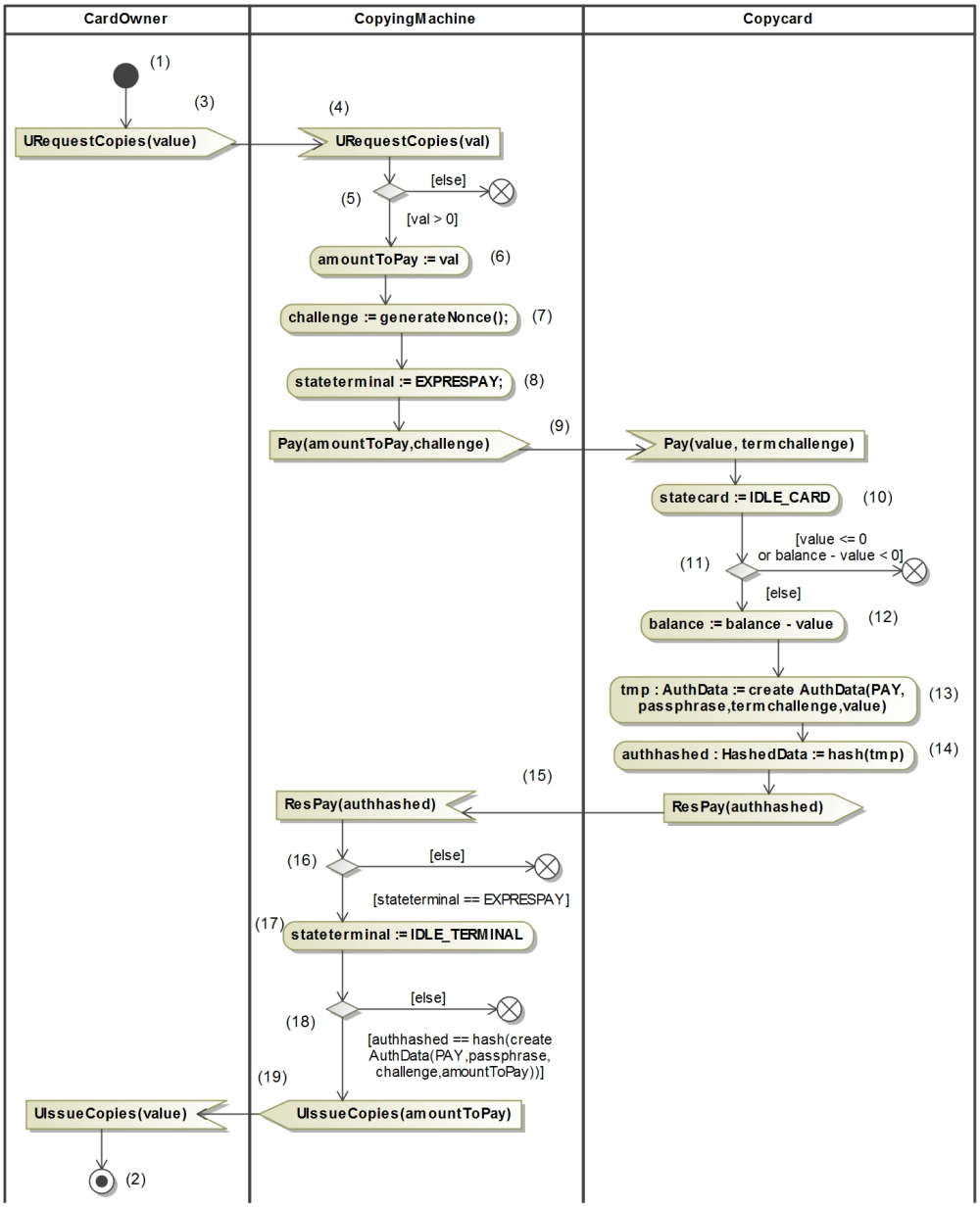

Abbildung 3.4: Aktivitätsdiagramm für das Bezahlen von Kopien [62]

in der richtigen Reihenfolge ausgeführt werden können. In diesem Fall kann der zweite Protokollschritt (*ResPay*) nur nach dem ersten (*URequestCopies*) ausgeführt werden. Bei erfolgreicher Zustandsüberprüfung wird ein Hashwert erstellt und dieser mit dem empfangenen verglichen (18). Durch das Geheimnis *passphrase* wird sichergestellt, dass die Nachricht nicht vom Angreifer stammt und durch die Zufallszahl *challenge*, dass es sich nicht um eine Nachricht aus einem früheren Protokollablauf handelt. Abschließend können die Kopien angefertigt werden und der Benutzer erhält zur Bestätigung die Anzahl der Kopien angezeigt (19).

Dieses Kapitel hat die Modellierung von Chipkarten kurz vorgestellt, da diese Arbeit darauf aufbaut. Im nächsten Kapitel wird auf die Modellierung von Web Services-Anwendungen und den damit verbunden Herausforderungen eingegangen.

KAPITEL 4

Modellierung von sicherheitskritischen Web Service-Anwendungen

Um eine sichere Web Service-Anwendung mit SecureMDD entwickeln zu können musste die Modellierungssprache von SecureMDD angepasst werden. Die sich daraus ergebende Modellierungssprache wird als SecureML bezeichnet. Mit SecureML können sichere Anwendungen modelliert werden, welche Web Services als auch Chipkarten verwenden und diese sogar miteinander interagieren können. Dieses Kapitel beschreibt die folgenden vier Teilbereiche von SecureML, die in dieser Arbeit betrachtet wurden.

1. **Web Services**
   Web Services unterscheiden sich erheblich von Chipkarten und müssen schon bei der Modellierung speziell betrachtet werden. Sie können von mehreren Teilnehmern gleichzeitig aufgerufen werden, weshalb ein Sessionmanagement notwendig ist. Des weiteren muss im Gegensatz zu Chipkarten eine dynamische Adressierung der Services modelliert werden. Kaskadierende Serviceaufrufe sowie die Interaktion mit Chipkarten soll möglich sein, aber auch das Wiederverwenden und Orchestrieren der Services soll unterstützt werden. Die Modellierung von Web Services ist in [17] publiziert.

2. **Sicherheitseigenschaften**
   Um die Sicherheit einer Anwendung zu verifizieren muss zuerst definiert werden welche Sicherheitseigenschaften gelten sollen. Die Definition dieser Eigenschaften auf Modellebene hat mehrere Vorteile. Einerseits muss eine Eigenschaft nur einmal definiert werden und kann anschließend in verschiedene Spezifikationssprachen transformiert werden, z.B. für ein ASLan++ Modell (Model Checking) sowie für ein formales Modell von SecureMDD (interaktive Verifikation). Andererseits ist die Eigenschaft anhand des Anwendungsmodells definiert, was zu einer einfacheren Verständlichkeit und besseren Dokumentation der Sicherheitseigenschaften dient. Im Gegensatz zu einer Standardsicherheitseigenschaft wie Integrität oder Vertraulichkeit kann eine anwendungsspezifische Sicherheitseigenschaft nicht durch ein standardisier-

tes Schlüsselwort ausgedrückt werden, sondern wird in SecureMDD durch logische Aussagen über die internen Zustände der Komponenten beschrieben. Um solche anwendungsspezifischen Sicherheitseigenschaften auf Modellebene definieren zu können wurde OCL um einen Angreifer sowie kryptographische Operationen erweitert. Zudem können auch initiale Eigenschaften sowie Hilfsaussagen auf dem Modell definiert werden. Die Modellierung von Sicherheitseigenschaften, initialen Eigenschaften sowie Hilfsaussagen ist in [19] publiziert.

3. **Abstraktionen** Besonders beim Model Checking ist es gängige Praxis, große Anwendungen von Hand stark zu abstrahieren, damit der Suchraum kleiner wird und somit auch größere Anwendung in gewünschter Zeit auf Schwachstellen überprüft werden können. Da in SecureMDD die zu überprüfende formale Spezifikation aus dem Anwendungsmodell generiert wird, muss schon das Anwendungsmodell abstrahiert werden. Da dieses jedoch sehr detailliert sein muss, da aus diesem lauffähiger Code generiert wird, werden nicht die vorhandenen Diagramme geändert, sondern die Abstraktionen durch zusätzliche Diagramme modelliert. Die Modellierung der Abstraktionen ist in [20] publiziert.

4. **Externe Services** Die Kommunikation mit existierenden Systemen ist für eine Vielzahl von Anwendungen notwendig. Durch die Integration von externen Services in den SecureMDD Ansatz kann die Interaktion mit bestehenden Dienstleistern, Bibliotheken oder auch Altsystemen schon auf Modellebene realisiert werden. Dadurch muss der generierte Code nicht verändert werden, wodurch die Refinment-Beziehung zwischen formalen Modell und ausführbarer Anwendung nicht verändert wird. Die Integration von externen Services ist in [22] publiziert.

4.1 UML-Profil

Damit die Web Services, Sicherheitseigenschaften, Abstraktionen sowie externe Services modelliert werden können musste das UML-Profil aus 3.2 erheblich erweitert werden.

4.1.1 Web Services

Tabelle 4.1 zeigt eine Auflistung des erweiterten UML-Profils, um Web Services sowie die sichere Kommunikation mit Web Services zu modellieren.

Web Service-Komponenten

Ein Web Service wird mit dem Stereotyp ≪Service≫ modelliert. Dieser Stereotyp erweitert die UML-Metaklasse *Class* und besitzt die optionale Eigenschaft *singleton*. Ein solcher Service ist eine technische Komponente wie eine Smartcard. Es

Stereotyp	Erweiterte Metaklasse	Eigenschaften
Service	Class	singleton
Terminal	Class	insecure
static	Property	
openSession	SendSignalAction	
closeSession	AcceptEventAction	
sendTo	SendSignalAction	URL
key	Property	
Exception	Activity	
Message	Class	fixMessageOrder
TLS	Class	SSA, MA
reliable	CommunicationPath	

Tabelle 4.1: Definition des erweiterten UML-Profils für Web Services

muss jedoch beachtet werden, dass im Gegensatz zur Smartcard kein exklusiver Zugriff auf einen Service besteht. Das bedeutet, dass ein Service nicht nur einen Aufrufer, sondern eine Vielzahl von Aufrufern gleichzeitig verwalten muss. Die Eigenschaft *singleton* gibt an, dass ein Service nur genau einmal instanziiert werden soll und das somit alle Kommunikationspartner mit der selben Instanz interagieren.

Die Kommunikation mit einem Service erfolgt über einen anderen Service oder über ein Terminal. Ein Terminal ist ein sicheres Gerät dessen interner Speicher ein Angreifer nicht auslesen oder manipulieren kann. Der Stereotyp ≪Terminal≫ wurde um die optionale Eigenschaft *insecure* erweitert. Dadurch können auch unsichere Terminals, wie z.B. ein Personal Computer auf dem Malware installiert ist, betrachtet werden. In der ETicket-Fallstudie kommt ein solches unsicheres Terminal zum Einsatz und ermöglicht, dass der generierte Terminal-Code auf ein beliebiges Java-fähiges Gerät installiert werden kann, ohne dass die Sicherheitseigenschaften dadurch verletzt werden. Hierzu musste ein spezielles Protokoll modelliert werden, bei dem eine Ende-zu-Ende Verschlüsselung zwischen Smartcard und Service erfolgt und das unsichere Terminal die Nachrichten nur weiterleitet.

Sitzungsverwaltung

Ein Service verwaltet für jeden Aufrufer eine neue Sitzung. Dies ist besonders wichtig bei der Modellierung von kryptographischen Protokollen, da der Service für jeden Aufrufer Sitzungsdaten wie Schlüssel oder Nonces verwalten muss. Die Daten, die ein Service während einer Sitzung empfängt, können dann einfach in den modellierten Serviceattributen gespeichert werden und jeder Aufrufer hat nur Zugriff auf die Attributwerte aus seiner Sitzung. Um Daten unabhängig von einer Sitzung zu speichern (wie z.B. einen Kontostand, der auch nach einer Sitzung noch vorhanden sein muss) wurde der Stereotyp ≪static≫ eingeführt. Dieser erweitert die UML-Metaklasse Property und darf nur auf Serviceattribute angewendet wer-

den.

Mittels der Stereotypen ≪openSession≫ und ≪closeSession≫ kann der Gültigkeitsbereich einer Sitzung angegeben werden. ≪openSession≫ erweitert die UML-Metaklasse *SendSignalAction*, welche zum Senden einer Nachricht verwendet wird und ≪closeSession≫ erweitert die UML-Metaklasse *AcceptEventAction*, die zum Empfangen einer Nachricht verwendet wird.

Mehrfachinstanziierung

Wenn eine Servicekomponente mehr als einmal instanziiert wird, dann muss beim Modellieren einer zu sendenden Nachricht an diese Komponente angegeben werden an welche konkrete Instanz die Nachricht gesendet werden soll. Da eine Servicekomponente beliebig oft instanziierbar sein kann, ist eine statische Repräsentation bzw. Zuordnung der verschiedenen Instanzen nicht möglich. Deshalb wurde der Stereotyp ≪sendTo≫ definiert. Dieser erweitert die UML-Metaklasse *SendSignalAction* und besitzt das Feld *URL* vom Typ String. In diesem Feld können MEL-Ausdrücke definiert werden wie z.B. der Zugriff auf Variablen. Dadurch kann die Serviceadresse dynamisch übergeben und die dazugehörige Serviceinstanz aufgerufen werden. In den Protokollen müssen somit nicht alle Instanzen modelliert werden, sondern nur die Serviceklasse und der Aufruf mit dem Stereotypen ≪sendTo≫.

Key-Value-Listen

Der Stereotyp ≪key≫ erweitert Listen (Attribute mit Multiplizität *) zu Key-Value-Listen. ≪key≫ erweitert die UML-Metaklasse *Property* und darf nur auf ein Attribut in einer Liste annotiert werden. Dadurch wird dieses Attribut zum Key und die ganze Klasse zum Value.

Ausnahmebehandlung

Der Stereotype ≪Exception≫ erweitert die UML-Metaklasse *Activity*. Dadurch kann die Ausnahmebehandlung von einzelnen Komponenten in Aktivitätsdiagrammen modelliert werden. Die Ausnahmebehandlungen werden dann bei modellierten (*FlowFinalNode*) und insbesondere bei internen Ausnahmefällen (Aufrufe von vordefinierten Operationen wie *decrypt*) aufgerufen.

Nachrichtenreihenfolge

Oft ist die Reihenfolge, in der Nachrichten akzeptiert werden sollen entscheidend für die Sicherheit einer Anwendung. Bisher musste die Nachrichtenreihenfolge explizit durch das Setzen und Abfragen von Zuständen modelliert werden. Da die Nachrichtenreihenfolge innerhalb eines Aktivitätsdiagramms durch die Anordnung der Nachrichten gegeben ist und die zusätzliche Modellierung zur Sicherstellung

der Nachrichtenreihenfolge eine häufige Fehlerquelle ist, kann die Nachrichtenreihenfolge innerhalb eines Aktivitätsdiagramms automatisch sichergestellt werden. Dies geschieht durch die Eigenschaft *fixMessageOrder* im Stereotyp ≪Message≫.

Kommunikationssicherheit

Bei der Entwicklung von neuen Protokollen können vordefinierte Datentypen und kryptographischen Operationen verwendet werden um die Kommunikation zu sichern. Für Services existieren anwendungsunabhängige Sicherheitsprotokolle um die Kommunikationssicherheit zu gewährleisten. Darunter zählt auch das bekannte Transport Layer Security (TLS) Protokoll, welches durch den Stereotypen ≪TLS≫ verwendet werden kann. ≪TLS≫ erweitern die UML-Metaklasse *Class* und besitzt die zwei Eigenschaften *SSA* und *MA*. *SSA* beschreibt TLS mit serverseitiger Authentifizierung (Server Side Authentication) und *MA* steht für gegenseitige Authentifizierung (Mutual Authentication). In beiden Fällen können Nachrichten verloren gehen, wenn ein Angreifer die Fähigkeit hat Nachrichten zu unterdrücken. Wenn eine Anwendung darauf angewiesen ist, das alle Nachrichten irgendwann ankommen, dann kann das durch den Stereotypen ≪reliable≫ modelliert werden. Dieser erweitert die UML-Metaklasse *CommunicationPath*.

4.1.2 Sicherheitseigenschaften

Stereotyp	Erweiterte Metaklasse
ApplicationConstraints	Class
Attacker	Class
OCLOperation	Operation
InitializeConstraints	Class
HelperConstraints	Class

Tabelle 4.2: Definition des erweiterten UML-Profils für Sicherheitseigenschaften

Tabelle 4.2 zeigt die Erweiterung des UML-Profils um Stereotypen, die zum Modellieren von Sicherheitseigenschaften notwendig sind. Die anwendungsspezifischen Sicherheitseigenschaften werden aufgrund einer besseren Übersichtlichkeit in einer Klasse mit dem Stereotypen ≪ApplicationConstraints≫ als OCL-Constraints definiert. Ein Angreifer wird durch eine Klasse mit dem Stereotypen ≪Attacker≫ definiert. Dadurch ist es möglich innerhalb einer Constraint Zugriff auf einen Angreifer bzw. sein Angreiferwissen zu haben. Durch den Stereotyp ≪OCLOperation≫ ist es möglich das Verhalten von Operationen durch *Body Conditions* zu definieren und diese Operationen in OCL-Constraints zu verwenden. Da Sicherheitseigenschaften nicht für jeden beliebigen initialen Zustand gelten, werden solche Aussagen ebenfalls im Anwendungsmodell definiert. Dies geschieht durch OCL-Constraints in einer Klasse, welche den Stereotypen ≪InitializeConstraints≫

anwendet. Neben den initialen Aussagen werden auch noch Hilfsaussagen unterstützt. Diese sind für die Verifikation nicht notwendig jedoch hilfreich, da sich die Sicherheit der Anwendung darauf stützt.

4.1.3 Abstraktionen

Stereotyp	Erweiterte Metaklasse
ModelCheckingDiagram	Package
Threat	CommunicationPath, CallBehaviorAction
MaxInvoCount	Activity

Tabelle 4.3: Definition des erweiterten UML-Profils für manuelle Abstraktionen

Tabelle 4.3 beschreibt Stereotypen, die für die Modellierung von Abstraktionen notwendig sind. Der Stereotyp ≪ModelCheckingDiagram≫ wird im Anwendungsmodell auf ein *Package* angewendet. In diesem *Package* können anschließend die Abstraktionen in einem Testdiagramm modelliert werden. Der Stereotyp ≪Threat≫ wurde um die Anwendung auf *CallBehaviorActions* erweitert, wodurch die Angreiferfähigkeiten nicht nur auf den Kommunikationskanälen, sondern auch auf den Aktivitäten definiert werden können. Mithilfe von ≪MaxInvoCount≫ kann pro Testdiagramm die Ausführungshäufigkeit bestimmt werden.

4.1.4 Externe Services

Stereotyp	Erweiterte Metaklasse
ExternalServiceClassDiagramm	Package
ExternalService	Class
ExternalMessage	Class
ExternalClass	Class
ExternalServiceProxy	Class

Tabelle 4.4: Definition des erweiterten UML-Profils für externe Services

Tabelle 4.4 zeigt die wichtigsten Stereotypen die für die UML-Repräsentation von externen Services verwendet werden. Für die Modellierung der Kommunikation mit einem existierenden Web Services wird eine WSDL (Web Service Definition Language), die die Dienste und Nachrichten des Web Services beschreibt, automatisch nach UML transformiert. Ein UML-Package mit dem Stereotypen ≪ExternalServiceClassDiagramm≫ enthält anschließend diese UML-Klassen. Es wird zwischen drei Arten von Klassen unterschieden. Die Klasse, die den externen Service repräsentiert erhält den Stereotyp ≪ExternalService≫. Bei den restlichen Klassen handelt es sich entweder um die übertragenen Nachrichten, welche mit dem Stereotyp ≪ExternalMessage≫ automatisch annotiert werden oder

um Datenklassen, die den Stereotype ≪ExternalClass≫ erhalten. Der Stereotyp ≪ExternalServiceProxy≫ wird vom Anwendungsentwickler verwendet, um eine Proxy-Klasse zu modellieren, welche die Konvertierung der selbst modellierten Nachrichtenklassen in die aus der WSDL generierten Klassen durchführt und an den externen Services weiterleitet.

Stereotyp	**Erweiterte Metaklasse**
Policy	Class
AsymmetricBinding	Class
SymmetricBinding	Class
TransportBinding	Class
EncryptedHeader	Class
EncryptedBody	Class
SignBody	Class
SignHeader	Class
SignBeforeEncrypting	Class
EncryptBeforeSign	Class
PolicyContainer	Class

Tabelle 4.5: Definition des erweiterten UML-Profils für WS-Policies

Tabelle 4.5 beschreibt eine Reihe von Stereotypen, die notwendig sind, falls der externe Service für eine sichere Nachrichtenübertragung WS-SecurityPolicy verwendet. In diesem Fall werden die Policies, welche im WSDL-Dokument integriert sind ebenfalls in eine UML-Repräsentation transformiert (es wird nur ein Teilmenge der vollständigen WS-SecurityPolicy Spezifikation unterstützt). Hierzu wird für jede Policy eine Klasse generiert. Diese Klasse wird mit dem Stereotypen ≪Policy≫ annotiert. Für eine Policy können unterschiedliche "security bindings" definiert werden. Der Stereotyp ≪AsymmetricBinding≫ beschreibt, dass die Nachrichtensicherheit durch asymmetrische Verfahren sichergestellt wird. Durch ≪SymmetricBinding≫ wird definiert, dass ein symmetrischer Schlüssel mit asymmetrischen Verfahren ausgehandelt wird und anschließend für die restliche Kommunikationssicherheit symmetrische Verfahren verwendet werden. Im Gegensatz dazu beschreibt ≪TransportBinding≫, dass die Nachrichtensicherheit auf der Transportebene sichergestellt wird (z.B. durch HTTPS). Welche kryptographischen Operationen, auf welche Teile einer Nachricht und in welcher Reihenfolge angewendet werden, wird ebenfalls mit Stereotypen beschrieben. ≪EncryptedHeader≫ und ≪EncryptedBody≫ wird zum Verschlüsseln verwendet, ≪SignBody≫ und ≪SignHeader≫ zum Signieren und ≪SignBeforeEncrypting≫ sowie ≪EncryptBeforeSign≫ um die Reihenfolge anzugeben. Eine solche Policy kann vor der Nachrichtenübertragung einem ≪PolicyContainer≫ zugewiesen werden. Hierzu braucht ein Client, der einen externen Service aufruft ein Klassenattribut vom Typ *PolicyContainer*.

4.2 Web Services

Für die Modellierung von Web Services mussten einige Aspekte betrachtet werden. Hierzu gehören Serviceadressen sowie der Kommunikationsaufbau über diese, Serviceoperationen, Sitzungsdaten und gemeinsam genutzte Daten, aber auch die Kommunikationsstruktur sowie die Kommunikationssicherheit. Zudem wurden unsichere Benutzergeräte, Key-Value-Listen sowie selbstständig wiederholende Ereignisse und Ausnahmebehandlungsroutinen betrachtet, da diese für viele Web Service-Anwendung einschließlich der in dieser Arbeit betrachteten Fallstudien relevant sind.

4.2.1 Statische Sicht

Services

Ein Service wird durch eine Klasse mit dem Stereotyp *Service* modelliert. Die Attribute eines Services werden wie auch bei den anderen Komponenten als eigene Attribute oder als gerichtete Assoziationen mit Rollennamen im Modell definiert. Der Typ eines Attributes kann eines der vordefinierten Datentypen (siehe Abschnitt 3.1) oder eine selbst modellierte Klasse sein. Das ermöglicht die Modellierung von komplexen Datentypen, da diese Klassen wiederum weitere Attribute und Assoziationen besitzen können. Zudem werden sie verwendet um kryptographische Daten zu modellieren (siehe Abschnitt 3.2). Der Anwendungsentwickler kann auf alle Serviceattribute den Stereotype ≪static≫ anwenden. Dadurch haben alle Aufrufer dieses Services Zugriff auf die selben Attribute und nicht auf Kopien, die für die Sitzung des Aufrufers erstellt wurden. Dieses Verhalten wird z.B. bei einem Service, der eine Vielzahl von Konten verwalten muss benötigt. Des Weiteren können Services ebenfalls von einer abstrakten Oberklasse erben. Dies ist dann hilfreich, wenn z.B. zwei Services gleiche aber auch unterschiedliche Funktionalitäten besitzen. Die vorgestellte Debitcard-Fallstudie besitzt zwei Arten von Banken: Solche, die eigene Geldautomaten besitzen und solche die ihren Kunden das Abheben von Geldbeträgen nur über Bankautomaten anderer Banken ermöglichen. Zudem unterstützen aber beide Arten von Banken das Tätigen von Überweisungen auf die gleiche Art und Weise.

Abbildung 4.1 zeigt die Modellierung von zwei Services mit jeweils einem unterschiedlichen Attribut sowie zwei gemeinsamen Attributen. *WS* ist eine abstrakte Servicekomponente mit einem symmetrischen Schlüssel *key* und einer Nonce n sowie einem *static* Attribut a vom Typ A, welches als Assoziation modelliert ist. *WS1* und *WS2* sind konkrete Services, die von der abstrakten Servicekomponente erben.

Die Kommunikation mit einem Service findet durch den Austausch von Nachrichten statt. Eine Unterscheidung zwischen Smartcard- und Service-Nachrichten ist nicht notwendig, da diese während der Transformationen automatisch ermittelt wird. Somit können Nachrichten, die zwischen Smartcard und Service über ein

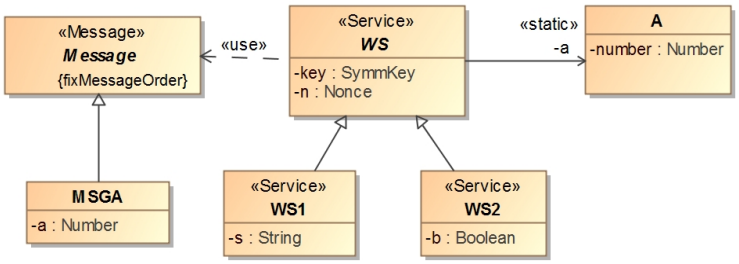

Abbildung 4.1: Modellierung von Web Services

Terminal ausgetauscht werden, als Unterklasse der abstrakten Nachrichtenklasse mit dem Stereotyp ≪Message≫ modelliert werden (siehe *MSGA* in Abbildung 4.1). Eine Strukturierung der Nachrichten nach Komponenten bzw. Funktionalitäten kann mittels Klassendiagrammen modelliert werden. Die Einhaltung einer festen Nachrichtenreihenfolge innerhalb eines Aktivitätsdiagramms kann durch die Eigenschaft *fixMessageOrder* des Stereotyps ≪Message≫ modelliert werden.

Unsichere Terminals

Unsichere Terminals werden durch eine Klasse mit dem Stereotyp ≪Terminal≫ und der Eigenschaft *insecure* modelliert. Die Modellierungsrichtlinien für ein unsicheres Terminal erlauben nur das Weiterleiten von Nachrichten. Dies kann während der Validierung des Anwendungsmodells geprüft werden.

Serviceadressen

Serviceadressen werden durch den Datentyp *Address* modelliert. Hierbei handelt es sich um einen vordefinierten Datentypen (wie *Secret* und *Nonce* siehe Abschnitt 3.1), der als Klasse mit dem Attribut *address* vom Typ *String* definiert ist (siehe Abbildung 4.2).

Abbildung 4.2: Vordefinierter Datentyp Address

*-Listen und Key-Value-Listen

Listen werden durch Attribute mit einer Multiplizität größer 1 modelliert. Da Chipkarten ressourcenbeschränkt sind konnten bisher nur Listen mit einer festen Länge (0..n) unterstützt werden. Services hingegen können nun auch Listen beliebiger Länge besitzen. Dies wird durch eine *-Mulitiplizität modelliert.

Eine Erweiterung von Listen stellen Key-Value-Listen dar. Durch Key-Value-Listen kann nicht nur über den Index auf die Elemente zugegriffen werden, sondern auch über ein Schlüsselwort. Dieses Schlüsselwort wird in der Klasse, welches den Elementtyp beschreibt durch den Stereotypen ≪key≫ definiert. Jeder Schlüsselwert in einer solchen Key-Value-Liste ist eindeutig und wird genau einem Value zugeordnet. Der Datentyp des Schlüsselwortes ist beliebig wählbar. Diese Key-Value-Listen werden für verschiedene Aufgaben verwendet. Zum Beispiel werden sie in einem Banksystem für die Zuordnung zwischen Bankleitzahl und der Serviceadresse benötigt, aber auch für die Modellierung von Konten, um anhand der Kontonummer die Kontoklasse zu erhalten.

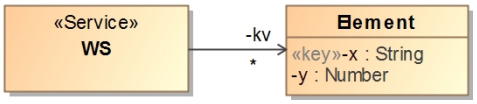

Abbildung 4.3: Modellierung einer Key-Value-Liste

Abbildung 4.3 zeigt die Modellierung einer solchen Key-Value-Liste. Der Service *WS* besitzt das Attribut *kv* mit *-Mulitiplizität, welches eine Key-Value-Liste beliebiger Größe mit dem Schlüsselwort x vom Typ *String* darstellt und dessen Elemente vom Typ *Element* sind.

Kommunikationsstruktur

Die Kommunikationspartner eines Services werden in einem Deploymentdiagramm mit den anderen Komponenten definiert. Dazu wird ein Service als Knoten (*Node*) mit dem Stereotyp ≪Service≫ sowie der optionalen Eigenschaft *singleton* modelliert. Ein Kommunikationskanal zu einem Service wird durch einen Kommunikationspfad (*CommunicationPath*) repräsentiert. Es werden unidirektionale sowie bidirektionale Kommunikationspfade unterstützt. Die Navigierbarkeit (Richtung) des *CommunicationPath* gibt an, wer der Aufrufer ist. Jeder aufgerufene *Node* darf unabhängig von der Navigierbarkeit eine Rückantwort senden. Da zudem Services andere Services aufrufen können, sind somit komplexe Kommunikationsstrukturen möglich.

Abbildung 4.4 zeigt einen Client (*Terminal*) sowie drei Services (*WS*, *WS1* und *WS2*) die miteinander kommunizieren. *WS* kann einerseits vom *Client* aufgerufen werden und andererseits *WS1* und *WS2* aufrufen. *WS1* und *WS2* dürfen *WS* nur eine Rückantwort zurücksenden genauso wie *WS* nur eine Rückantwort an *Client* senden darf. Im Gegensatz dazu ist die Verbindung zwischen *WS1* und *WS2* bidirektional, weshalb z.B. *WS* den Service *WS2* direkt oder indirekt über WS1 aufrufen kann. Zudem kann *WS2* sich selbst aufrufen. Dadurch können verschiedene Instanzen der selben Servicekomponente miteinander kommunizieren. Ein Beispiel hierfür sind Überweisungen zwischen zwei Banken, mit dem gleichen Verhalten. Ob ein Aufrufer eines Services genau eine Instanz dieses Services kennt

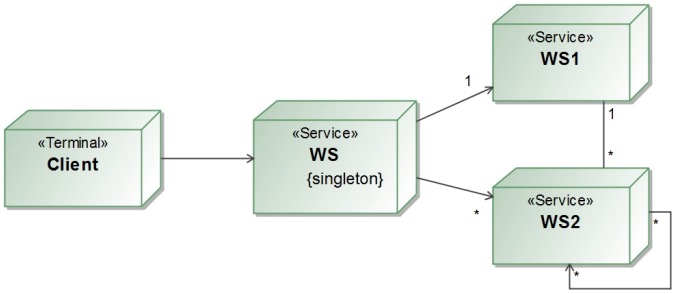

Abbildung 4.4: Modellierung einer Web Services Kommunikationsstruktur

oder beliebig viele, kann durch die die Mulitiplizitäten 1 oder * an die Enden der Kommunikationspfade zu Serviceknoten modelliert werden. Wird keine Multiplizität angegeben, dann wird 1 angenommen. Durch die Angabe der Multiplizitäten kann aber nicht angegeben werden, dass im gesamten System nur eine Instanz eines Services existiert. Es kann modelliert werden das jeder zwar nur eine kennt, aber jeder kann eine andere Instanz kennen. Diese Eigenschaft wird mit der Serviceeigenschaft *singleton* modelliert. Somit kennen in Abbildung 4.4 alle Instanzen von *Client* nur einen Service *WS*.

Kommunikationssicherheit

Web Services werden in öffentlichen bzw. privaten Netzwerken veröffentlicht. Öffentliche Netzwerke wie das Internet sind jedoch nicht sicher [35]. Somit wird die Kommunikation mit Web Services über ein unsicheres Netzwerk durch den Stereotyp ≪Threat≫ mit den Fähigkeiten *read, send, suppress* auf dem Kommunikationspfad modelliert. Bei privaten Netzwerken wie einem Intranet wird davon ausgegangen, dass ein Angreifer zu einem solchen Netzwerk keinen Zugang besitzt und dieses somit sicher ist. Deshalb wird zusätzlich ein sicheres Netzwerk in dem der Angreifer keine Fähigkeiten besitzt unterstützt. Die Kommunikation über ein sicheres Netzwerk wird durch einen Kommunikationspfad mit dem Stereotyp ≪Threat≫ ohne Angreiferfähigkeiten modelliert. Die zwischen den Komponenten übertragenen Nachrichten können in einem unsicheren Netzwerk durch Sicherheitsprotokolle vor dem Angreifer und seinen Fähigkeiten geschützt werden. Diese Protokolle können mit SecureML entwickelt werden, aber es können auch bestehende Sicherheitsprotokolle verwendet werden. Die Verwendung von Standardprotokollen wie TLS machen die Modellierung von sicherheitskritischen Anwendungen leichter. Eine Ende-zu-Ende-Verschlüsselung z.B. zwischen einem Web Service und einer Chipkarte ist mit TLS jedoch nicht möglich. Durch den Stereotype ≪TLS≫ kann angegeben werden, dass das existierende Transport Layer Security-Protokoll angewendet wird. Die aufgerufene Servicekomponente wird direkt mit ≪TLS≫ annotiert. Dadurch sind alle Kommunikationspfade zu diesem

Service mit TLS geschützt und es wird verhindert, dass es eine gesicherte und eine ungesicherte Verbindung zu einem Service und seinen Serviceoperationen gibt. TLS verhindert zwar, dass ein Angreifer in einem unsicherem Netzwerk Nachrichten lesen, manipulieren und deren Reihenfolge verändern kann, aber es verhindert nicht, dass Nachrichten aufgrund einer Verbindungsunterbrechung verloren gehen. Bei einer Überweisung zwischen zwei Banken muss der Geldbetrag, welcher von einem Konto abgebucht wurde, dem anderen hinzugebucht werden. Wenn aber die letzte Nachricht niemals ankommt, dann ist unklar ob der Betrag tatsächlich hinzugebucht wurde. Hierfür können eigene Protokolle mit SecureML modelliert werden, es können aber auch bestehende Lösungen wie Standleitungen oder "Guaranteed Delivery"-Systeme verwendet werden . Die Eigenschaft, dass Nachrichten über ein sicheres Netzwerk bzw. über ein mit TLS und gegenseitiger Authentifizierung gesicherte Verbindung nicht verloren gehen können, wird durch den Stereotyp ≪reliable≫ auf einen solchen Kommunikationspfad modelliert.

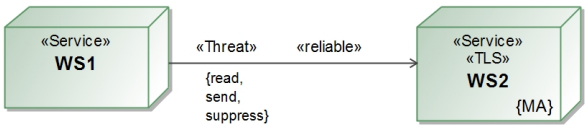

Abbildung 4.5: Modellierung der Web Service Kommunikationssicherheit

Abbildung 4.5 zeigt, dass *WS1* die Servicekomponente *WS2* über eine zuverlässige (*reliable)* und TLS gesicherte Verbindung aufrufen kann. Obwohl der Angreifer auf dieser Verbindung die Fähigkeiten hat Nachrichten zu lesen, selbst erstellte Nachrichten zu senden sowie Nachrichten zu unterdrücken werden ihm diese Fähigkeiten durch existierende Mechanismen wieder genommen.

4.2.2 Dynamische Sicht

Das Verhalten eines Services wird in Swimlanes innerhalb von Aktivitätsdiagrammen modelliert. Welches Verhalten zu welchem Service gehört wird durch die Zuordnung einer Serviceklasse zu einer Swimlane modelliert. Innerhalb der Swimlanes können UML-Elemente mit MEL-Ausdrücken sowie vordefinierten Stereotypen zur Modellierung des Serviceverhaltens verwendet werden.

Serviceaufrufe

Ein Serviceaufruf wird durch die Übertragung von Nachrichten modelliert. Es wird eine Nachricht an den Service gesendet und dieser antwortet mit einer Rücknachricht. Währenddessen kann er wiederum andere Services aufrufen. Das Senden einer Nachricht geschieht mit einer *SendSignalAction* und das Empfangen mit einer *ReceiveEventAction*. Der Bereich dazwischen wird als Protokollschritt bezeichnet.

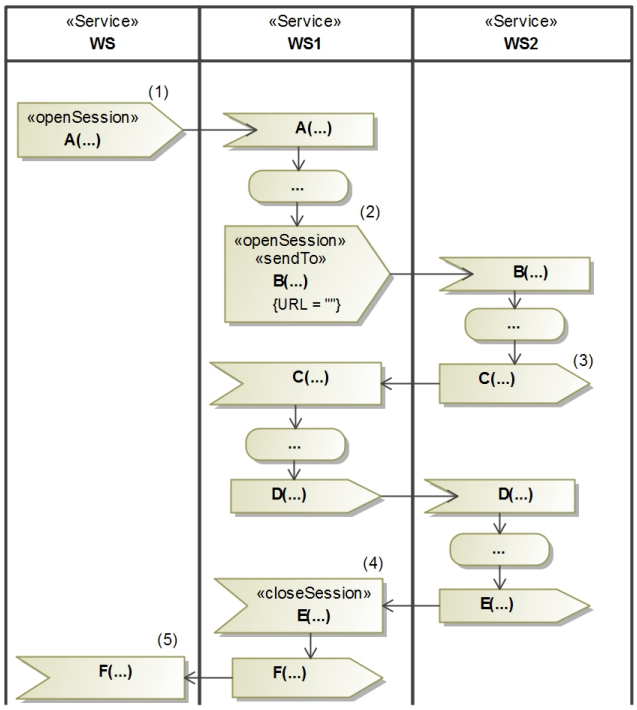

Abbildung 4.6: Modellierung von kaskadierenden Serviceaufrufen

Abbildung 4.6 zeigt einen kaskadierenden Serviceaufruf zwischen drei Services. Dabei wird das Deploymentdiagramm aus Abbildung 4.4 angenommen. *WS* sendet die Nachricht *A* and den Service *WS1* (1). Da es sich hierbei um den ersten Aufruf des Services *WS1* handelt muss eine Sitzung mit *openSession* gestartet werden. Dies ist wichtig, da es sonst nicht eindeutig ist an welcher Stelle in einem Protokoll eine Sitzung anfängt. Dadurch, dass *WS* nur genau eine Instanz von *WS1* kennt, muss der aufgerufene Service nicht näher spezifiziert werden. Nach dem Empfangen der Nachricht *A* kann der Service *WS1* durch MEL z.B. Überprüfungen, Zuweisungen oder Operationsaufrufe durchführen bevor er den Service *WS2* aufruft (2). Nun ruft *WS1* den Service *WS2* auf. Auch hier wird wieder eine neue Sitzung angefangen. Aber im Gegensatz zum ersten Serviceaufruf kennt der Service *WS1* eine Vielzahl von *WS2*-Instanzen. Deshalb wird der Stereotyp ≪sendTo≫ verwendet. Dieser besitzt die Property *URL*, der die konkrete Serviceadresse als MEL-Ausdruck übergeben wird. Somit wird die aufzurufende Serviceinstanz eindeutig beschrieben. *WS2* schickt daraufhin dem Aufrufer *WS1* eine Antwortnachricht (*C(...)* (3)) zurück. Da die Sitzung noch nicht geschlossen wurde, kann der Service *WS1* beliebig viele Nachrichten innerhalb der selben Sitzung an *WS2* schicken. Beim Empfangen der letzten Rückantwort eines Services wird die Sitzung durch *closeSession* am *AcceptEventAction* des Aufrufers geschlossen (4). Eine Sitzung muss jedoch nicht im selben Protokoll geschlossen werden in dem sie geöffnet wurde (5).

Serviceoperationen

Im vorherigem Abschnitt wurde der Serviceaufruf zusammen mit dem Serviceverhalten in einem Aktivitätsdiagramm definiert. Der Vorteil von dieser Modellierung ist die Darstellung eines Protokolls als Ganzes. Der Nachteil ist jedoch, dass das so definierte Serviceverhalten nicht wiederverwendet werden kann. Deshalb werden zu der bereits vorgestellten Modellierungsweise von Services zusätzlich Serviceoperationen unterstützt.

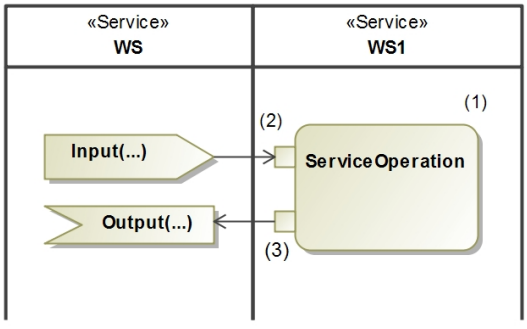

Abbildung 4.7: Aufruf einer Serviceoperation

Abbildung 4.7 zeigt den Aufruf einer Serviceoperation. Eine Serviceoperation wird als "Call Behavior"-Aktion (1) mit Ein- (2) und Ausgabepin (3) modelliert. Die Pins beschreiben die Eingabe- und Ausgabeparameter der Serviceoperation, die wiederum als Nachrichtenklassen modelliert sind. Das Aufrufen einer Serviceoperation wird wie bisher durch ein *SendSignalAction* und die Rückantwort durch ein *AcceptEventAction* modelliert. Das Serviceverhalten wird in einem separatem Aktivitätsdiagramm modelliert.

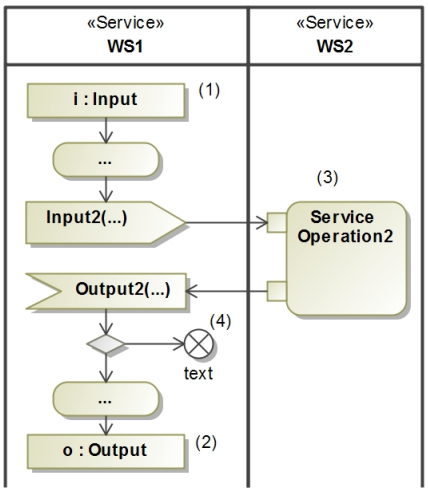

Abbildung 4.8: Modellierung des Verhaltens einer Serviceoperation

Abbildung 4.8 zeigt wie eine Serviceoperation modelliert wird. Die empfangene Nachricht (1) sowie die Rückantwort (2) werden durch jeweils eine *ActivityParameterNode* modelliert. Der *ActivityParameterNode* enthält einen Namen sowie einen Typ. Der Typ ist die Nachrichtenklasse die übertragen wurde, und der Name kann in MEL als Variablenname verwendet werden, um Zugriff auf die übertragene Nachricht zu erhalten. Der Bereich zwischen einem *ActivityParameterNode* und einem *SendSignalAction* sowie zwischen einem *AcceptEventAction* und einem *ActivityParameterNode* wird ebenfalls als Protokollschritt bezeichnet. Somit kann eine Serviceoperation aus mehreren Protokollschritten bestehen und innerhalb einer Serviceoperation eine andere Serviceoperation aufgerufen (3). Das Verhalten dieser kann dann wiederum auf die selbe Weise modelliert werden. Ein *FlowFinalNode* der in einem Service modelliert ist, kann zudem eine Benachrichtigung für den Aufrufer enthalten (4). Diese Benachrichtigungen können bis zum Benutzer weitergeleitet werden, wodurch dieser Informationen wie "Falsche Passworteingabe" erhalten kann. Es dürfen jedoch keine MEL-Ausdrücke und somit keine dynamischen Inhalte übergeben werden.

Ausnahmebehandlung

Die Modellierung von internen Operationen und Ausnahmebehandlungsroutinen werden wie eine Serviceoperation modelliert mit dem Unterschied, dass diese keine weiteren Services aufrufen können (enthalten nur eine Swimlane). Interne Operationen können mit MEL wie eine vordefinierte Operation aufgerufen werden. Die Ausnahmebehandlungsroutinen werden automatisch bei Ausnahmen aufgerufen, die durch vordefinierte Operationen entstehen können oder durch einen *FlowFinalNode* modelliert sind. Aus diesem Grund haben Ausnahmebehandlungsroutinen keine Input- oder Outputparameter, sondern greifen nur auf die Klassenvariablen des Services zu für den sie definiert sind. Der Start einer solche Ausnahmebehandlungsroutine wird wie ein Protokollstart durch einen *InitialNode* und das Ende durch einen *ActivityFinalNode* modelliert.

Wiederholende Ereignisse

Wiederholende Ereignisse ermöglichen das Ausführen von Serviceaktionen, ohne dass ein Benutzer mit der Anwendung interagiert. Dies kann wichtig sein, wenn periodisch Aktionen ausgeführt werden sollen. Dabei kann es sich z.B. um Benachrichtigungen oder um den Abschluss von unterbrochenen Geschäftsprozessen handeln. Die Modellierung findet anhand einer *AcceptEventAction* mit einem *Trigger* statt.

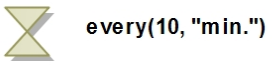

Abbildung 4.9: Modellierung von wiederholenden Ereignissen

Abbildung 4.9 zeigt die Modellierung eines Ereignisses, welches alle 10 Minuten angestoßen wird. Wie oft das Ereignisse ausgeführt werden soll, wird mit MEL modelliert.

Key-Value-Liste

Die Modellierung einer Key-Value-Liste als Klassenattribut eines Services wurde in Abbildung 4.3 gezeigt. Auf ein solches Attribut kann in Aktivitätsdiagrammen mit MEL zugegriffen werden. Hierzu besitzen Key-Value-Listen folgende vordefinierte Operationen:

- *get(Key k) : Value*
  Gibt den zum Key k passenden Value zurück. Dabei handelt es sich um eine Kopie. Änderungen an dieser Kopie werden nicht automatisch in der Key-Value-Liste geändert. Wenn es kein passenden Key gibt, dann findet ein Protokollabbruch statt.

- *set(Value v)* : void
  Fügt den Value v der Liste hinzu. Wenn ein Value mit dem selben Key schon vorhanden ist, wird dieser gelöscht. Anschließend wird v ans Ende der Liste angefügt.

- *containsKey(Key k) : Boolean*
  Überprüft ob es zum Key ein passenden Value gibt.

- *removeKey(Key k)* : void
  Löscht ein Element mit Key k. Wenn kein passender Key k enthalten ist wird nichts gemacht.

Zusätzlich zu den oben definierten Operationen, können auch die Listenoperationen *at*, *size*, *remove* und *contains* verwendet werden sowie *hasFree* falls die Anzahl der Elemente beschränkt ist (siehe Abschnitt. 3.3).

4.2.3 Fallstudie Onlinebanking

Dieser Abschnitt demonstriert die Modellierung von sicherheitskritischen Webserviceanwendungen anhand einer Onlinebanking-Anwendung. Die Benutzer dieser Anwendung können Geldtransaktionen zwischen zwei Konten über das Internet durchführen. Der Schwerpunkt dieser Anwendung ist der interne Überweisungsprozess zwischen zwei Banken.

Statische Sicht

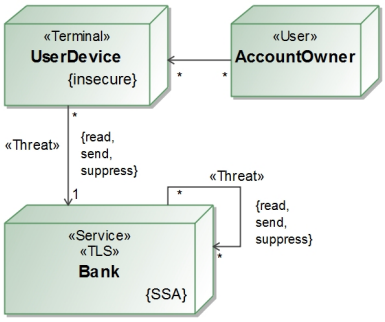

Abbildung 4.10: Deploymentdiagramm der Onlinebanking-Fallstudie

Abbildung 4.10 zeigt das Deploymentdiagramm, welches die Kommunikationsstruktur und die Kommunikationssicherheit der Onlinebanking-Anwendung beschreibt. Die Anwendung besteht aus den Kontobesitzern (*AccountOwner*), einem Heimcomputer (*UserDevice*) und den Bankservices (*Bank*). Das *UserDevice* kann von einem *AccountOwner* Eingaben empfangen und diese an den Bankservice *Bank* schicken. Da *Bank* einen Kommunikationspfad zu sich selbst hat, kann eine Instanz

der Servicekomponente *Bank* andere Instanzen von *Bank*, insbesondere sich selbst, aufrufen. Der Knoten *Bank* repräsentiert somit verschiedene Banken (z.B. Sparkasse oder Raiffeisenbank), die miteinander kommunizieren können, um Überweisungen zwischen verschiedenen Banken auszuführen. Dieses Diagramm beschreibt ein System, das aus mehreren Terminals und mehreren Bankservices besteht, die das selbe Protokoll verwenden, aber unterschiedlich initialisiert werden können. Zwischen *UserDevice* und *Bank* sowie zwischen *Bank* und *Bank* wird ein Angreifer mit der Fähigkeit Nachrichten zu lesen, selbst zu erstellen und zu unterdrücken angenommen. Da über diese Kommunikationspfade jedoch sensible Daten übertragen werden, ist ein Zugriff auf eine Instanz von *Bank* nur über TLS mit serverseitiger Authentifizierung möglich (siehe TLS mit der Eigenschaft SSA). Dadurch muss sich jede Instanz von *Bank* gegenüber ihrem Aufrufer authentifizieren und die übertragenen Nachrichten werden durch das TLS-Protokoll geschützt.

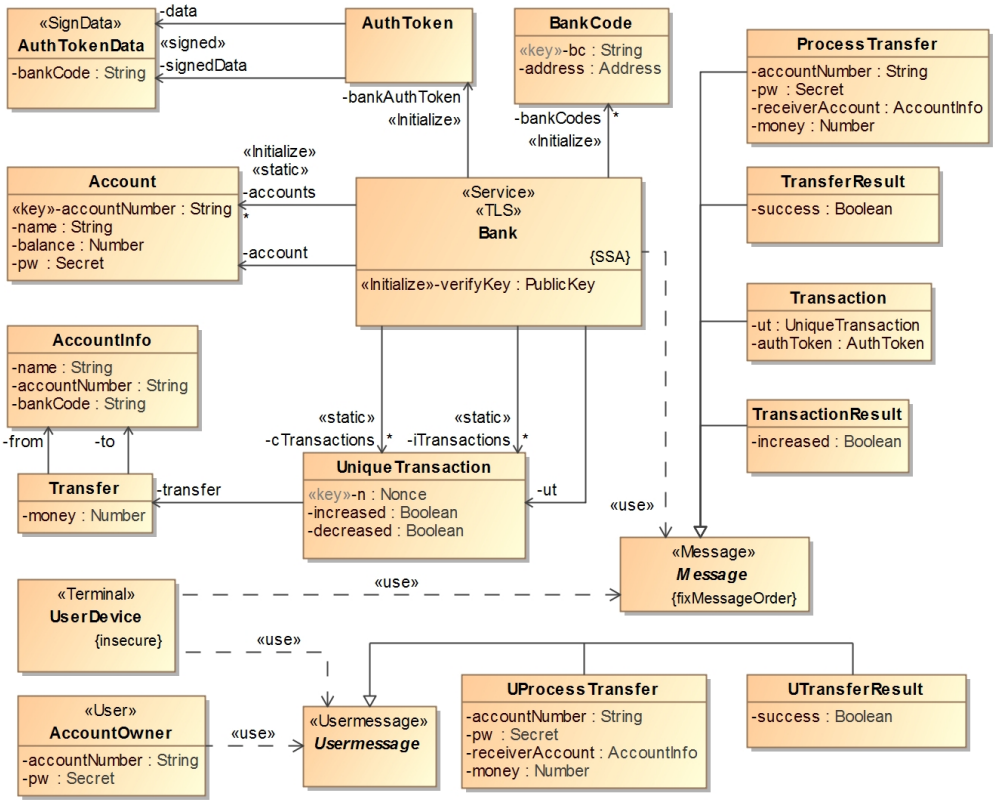

Abbildung 4.11: Klassendiagramm der Onlinebanking-Fallstudie

Abbildung 4.11 stellt das Klassendiagramm der Anwendung dar. Es zeigt die Serviceklasse *Bank*, das *UserDevice*, den Karteninhaber *AccountOwner* sowie die Nachrichtenklassen. Die Klasse *Bank* kann nur durch TLS mit serverseitiger Authentifizierung aufgerufen werden. Die Klasse *Bank* besitzt die zu initialisierenden

Attribute *accounts*, *BankCodes*, *bankAuthToken* sowie *verifyKey*. *accounts* ist eine Key-Value-Liste von Konten mit Kontoguthaben und Passwort. Anhand einer Kontonummer *accountNumber* kann eine solche Kontoklasse eindeutig identifiziert werden. *bankCodes* ist eine Key-Value-Liste, die in Abhängigkeit von Bankleitzahlen die Bankserviceadressen speichert. *bankAuthToken* speichert ein Authentifizierungstoken, welches eine Signatur der eigenen Bankleitzahl enthält. Dadurch kann sich eine Bank gegenüber einer anderen Bank authentifizieren. Das überprüfen dieser Tokens geschieht mit dem *verifyKey*. Zusätzlich besitzt die Klasse *Bank* eine Key-Value-Liste *accounts* in dem die Kontodaten aller Kunden gespeichert sind sowie zwei Key-Value-Listen namens *cTransactions* und *iTransactions*. *cTransactions* beschreibt erfolgreiche Überweisungen, während *iTransactions* unvollständige Überweisungen enthält. Eine Überweisung wird eindeutig durch eine *Nonce* identifiziert. Diese Key-Value-Listen sind notwendig um durch ein Recovery-Protokoll sicherzustellen, dass eine Überweisung komplett durchgeführt wurde oder gar nicht. Sie werden im Protokoll für Überweisungen in Abbildung 4.12 verwendet. Durch den Stereotypen ≪static≫ werden die zwei Attribute unabhängig von einer bestimmten Sitzung gespeichert, weshalb alle Serviceaufrufe auf die selben Attributinstanzen zugreifen. Dies ist notwendig, damit die Informationen nicht verloren gehen und somit in einer neuen Sitzung wieder vorhanden sind. Das *UserDevice* ist ein unsicheres Terminal, welches keine Daten speichert, sondern nur Nachrichten weiterleitet. Die Kontobesitzer *AccountOwner* kennen ihre Kontonummer sowie ihr Passwort. Die Nachrichtenklassen sind in *Message* und *Usermessage* aufgeteilt.

Dynamische Sicht

Wenn ein Kontobesitzer in diesem System von seinem Konto auf ein anderes Geld überweisen möchte, muss dieser zur Authentifizierung seine Kontonummer und sein Passwort an seinem Heimcomputer eingeben, sowie die Kontonummer und die Bankleitzahl des Empfängers zusammen mit dem zu überweisenden Betrag. Anschließend kann er mit diesen Daten eine Überweisung veranlassen. Diese Überweisungsnachricht wird direkt an die entsprechende Bank weitergeleitet indem die Serviceoperation *processTransfer* aufgerufen wird (siehe im Anhang Abbildung 9.1). Danach wird er darüber informiert, ob die Transaktion erfolgreich war oder fehlgeschlagen ist. Das Fehlschlagen einer Überweisung kann zwei Grunde haben. Erstens weil falsche Benutzereingaben gemacht wurden (das Empfängerkonto existiert nicht) bzw. weil nicht genug Geld auf dem Konto war. In diesem Fall wird der Kontobesitzer darüber informiert, das die Überweisung nicht ausgeführt wurde. Der zweite Grund für eine abgebrochene Überweisung kann ein technisches Problem (z.B. Verbindungsabbruch) oder ein Angriff auf das System sein. In diesem Fall wird der Anwender darüber informiert, dass die Überweisung nicht ausgeführt wurde, aber falls genug Geld auf dem Konto ist zu einem späteren Zeitpunkt noch ausgeführt wird.

Interner Überweisungsprozess:

Beim Tätigen einer Überweisung wird die Serviceoperation *processTransfer* von

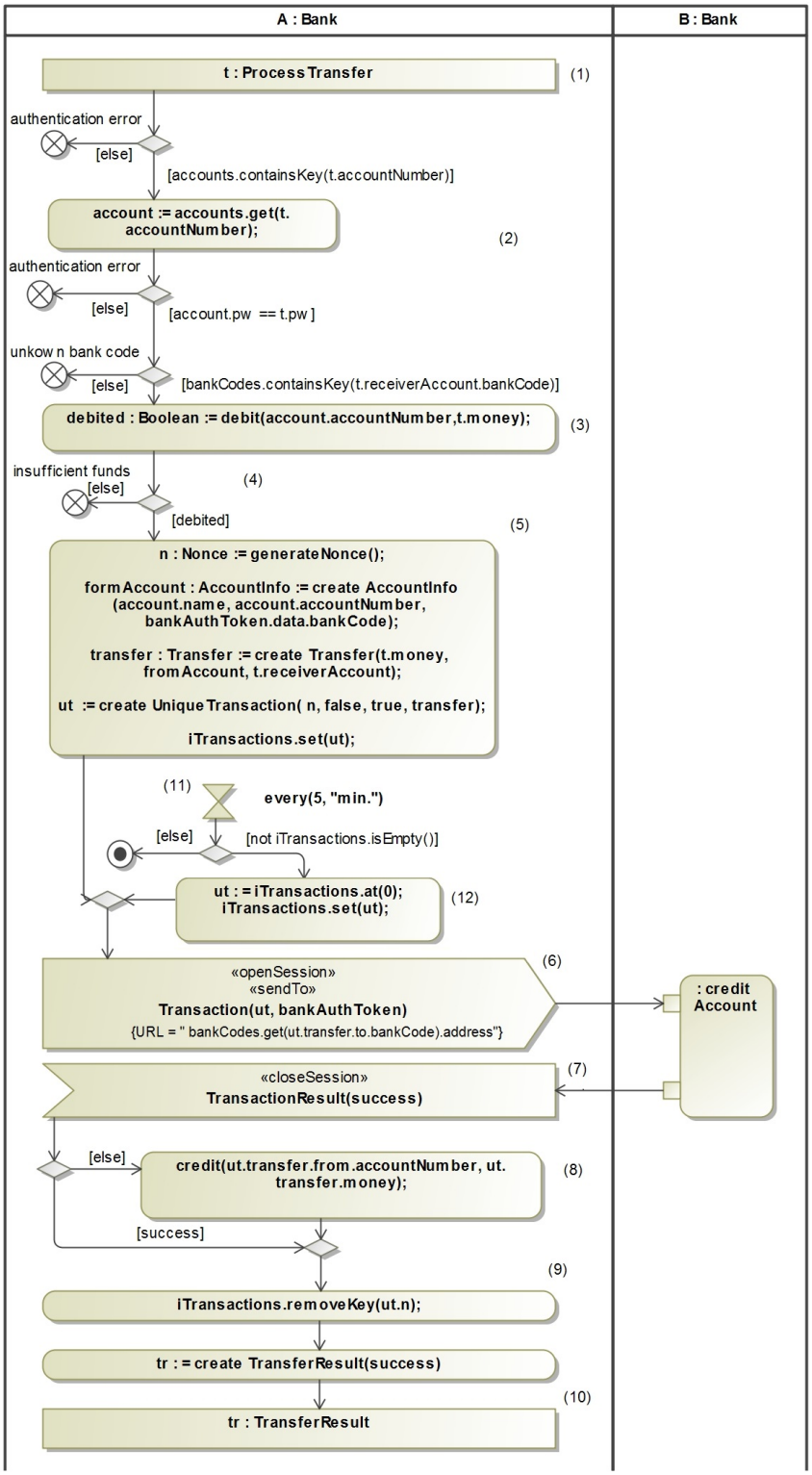

Abbildung 4.12: Serviceoperation zur internen Ausführung einer Überweisung

UserDevice aufgerufen. Abbildung 4.12 zeigt das Aktivitätsdiagramm, welches diese Serviceoperation definiert. Die Serviceoperation überprüft die Anmeldedaten, belastet (debit) ein Konto von Bank A und schreibt den Betrag einem Konto von Bank B gut (credit). A und B können verschiedene Serviceinstanzen sein, aber wenn beide Konten zur selben Bank gehören, dann repräsentieren A und B die selbe Instanz. Um den Betrag auf einem Konto von Bank B gutzuschreiben wird eine zweite Serviceoperation namens *creditAccount* aufgerufen. Zusätzlich enthält die Serviceoperation ein integriertes Recovery-Protokoll um sicherzustellen, dass die Überweisungen vollständig oder gar nicht ausgeführt werden, wenn die Verbindung abbricht oder ein Angreifer Nachrichten unterdrückt. Dies wird durch das Speichern von vollständigen und unvollständigen Überweisungen sowie einem Service, der wiederholt versucht unvollständige Überweisungen zu vervollständigen.

Im folgenden wird das Protokoll in Abbildung 4.12 detailliert beschrieben. Als erstes empfängt die Serviceoperation eine Nachricht vom Typ *ProcessTransfer* (1). Die Nachricht enthält die Authentifizierungsdaten (Kontonummer und Passwort), sowie das Empfängerkonto und den zu überweisenden Geldbetrag (siehe Abbildung 4.11). Dann wird überprüft ob die Kontonummer (*accountNumber*) der Bank bekannt ist und wenn dies nicht der Fall ist, dann wird eine *authentication error*-Nachricht an den Benutzer zurückgeschickt und der Überweisungsprozess wird abgebrochen (2). Falls die Kontonummer bekannt ist, werden die dazugehörigen Kontodaten aus der Key-Value-Liste mithilfe der *get*-Operation ausgelesen und dem Klassenattribut *account* zugewiesen. Im Anschluss wird das Passwort, welches sich in dem *account*-Objekt befindet mit dem übergebenen Passwort verglichen und bei Nichtübereinstimmung wird wieder eine *authentication error*-Nachricht an den Benutzer zurückgeschickt. Falls die Serviceadresse der Empfängerbank unbekannt ist (also kein passender Eintrag in der Key-Value-Liste *bankCodes* vorhanden ist), kann ebenfalls keine Überweisung durchgeführt werden und der Benutzer wird darüber mit der Nachricht *unkown bank code* informiert. Wenn, die Überprüfungen erfolgreich waren, wird anschließend die Operation *debit* aufgerufen (3). *debit* ist eine interne Operation welche das Konto des Kunden, der die Überweisung tätigt, mit dem zu überweisenden Betrag belastet. Diese Operation ist ebenfalls in einem Aktivitätsdiagramm modelliert (siehe im Anhang Abbildung 9.4). Wenn die Operation fehlschlägt, wird die lokale Variable *debited* auf false gesetzt, die Überweisung wird abgebrochen und der Benutzer wird durch die Nachricht *insufficient funds* informiert. Andererseits wenn das Belasten des Kontos erfolgreich war $debited == true$ (4) wird eine Nonce erstellt und in der lokalen Variable n gespeichert. Anschließend wird eine neue Instanz der Klasse *UniqueTransaction* (siehe Abbildung 4.11) mit der vorher genierten Nonce n und den Überweisungsdaten *transfer* erstellt und der globalen Variable *ut*, die als Klassenattribut modelliert wurde, zugewiesen und zu den unvollständigen Überweisungen (*iTransactions*) mithilfe der *set*-Operation hinzugefügt (5). Anschließend wird die Nachricht *Transaction* mit der eindeutigen Überweisung *ut* und einem Authentifizierungstoken an die Bank B geschickt (6). *bankAuthToken* und *bankCodes* werden während dem Deployment der Anwendung

initialisiert. *bankAuthToken* wird verwendet, damit unberechtigter Zugriff auf die kritische Serviceoperation *creditAccount* verhindert wird. Jede Bank erhält initial ein solches Authentifizierungstoken. Es besteht aus einer signierten Bankleitzahl und kann durch eine vordefinierte kryptografische Operation und dem passenden öffentlichen Schlüssel, den jede Bank besitzt, verifiziert werden. Da die Übertragung zwischen den Banken verschlüsselt ist, bleibt dieses Authentifizierungstoken geheim. *bankCodes* ist eine Key-Value-Liste von Bankleitzahlen und den dazugehörigen Serviceadressen. Der Aufrufer kennt eine beliebige Anzahl an Bankeninstanzen, was im Deploymentdiagramm durch eine Stern-Assoziation zum Bankknoten definiert wurde (siehe Abbildung 4.10). Deswegen wird der Stereotyp ≪sendTo≫ verwendet und die Adresse der aufzurufenden Bankinstanz dem Feld *URL* übergeben. Nachdem *Bank B* die Serviceoperation *creditAccount* ausgeführt hat, erhält *Bank A* die Rückantwort *TransactionResult* mit dem Nachrichtenparameter *success* (7). Wenn *success == true*, dann war die Überweisung erfolgreich und der zu überweisende Betrag wurde dem Empfängerkonto gutgeschrieben. Wenn der Nachrichtenparameter *success == false*, dann wurde der Betrag nicht dem Empfängerkonto hinzugebucht (z.B. wegen falsche Kontonummer), weshalb der Betrag dem Kunden, der die Überweisung getätigt hat, durch die interne Operation *credit* wieder hinzugebucht wird (8). Anschließend wird die vorher erstellte eindeutige Transaktion *ut* wieder entfernt, da die Nachricht empfangen und die Überweisung getätigt oder rückgängig gemacht wurde (9). Daraufhin wird eine Rückantwort vom Typ *TransferResult* an den Aufrufer gesendet. *debit* and *credit* sind private Operationen und können nur innerhalb der Komponente aufgerufen werden in der sie definiert wurden. Da Serviceanwendungen sehr umfangreich werden können, ist eine solche Kapselung der Funktionalität besonders wichtig.

Recovery-Protokoll:

Eine Überweisung kann durch einen Verbindungsausfall oder einen Angreifer der Nachrichten unterdrückt unterbrochen werden. Um sicherzustellen, dass eine Überweisung vollständig oder gar nicht ausgeführt wird, wird jede fünf Minuten ein Recovery-Protokoll ausgeführt. Dieses Protokoll fängt mit einem Zeitevent an (Accept-Event-Action mit dem Trigger *every(5, "min.")* (11)) und ist in der selben Swimlane modelliert, die auch die Serviceoperation *processTransfer* definiert (siehe Abbildung 4.12). Zuerst überprüft das Recovery-Protokoll, durch den Ausdruck *not iTransactions.isEmpty()* ob unvollständige Überweisungen existieren. Falls das Ergebnis falsch ist, ist der Protokollablauf zu Ende und wird nach fünf Minuten wiederholt. Wenn *iTransactions* nicht leer ist, wird das erste Element in der lokalen Variable *ut* zwischengespeichert und wieder ans Ende von *iTransactions* hinzugefügt (12). Danach wird die Serviceoperation *creditAccount* mit der in *ut* gespeicherten Überweisung aufgerufen (7). Ab dieser Stelle verhält sich das Protokoll genauso wie das Überweisungsprotokoll. Alle Überweisungsinformationen werden vor dem Aufrufen der Serviceoperation *creditAccount* zwischengespeichert. Falls ein Angreifer die Kommunikation zwischen den unterschiedlichen Banken stört, kann die Überweisung wiederholt werden. Aber wenn eine Überweisung wieder-

holt wird, soll der Betrag nicht zweimal dem Empfänger gutgeschrieben werden. Deshalb speichert die Serviceoperation *creditAccount* alle vollständig Überweisungen und überprüft ob die empfangene Überweisung schon zu einem früheren Zeitpunkt ausgeführt wurde. Falls sie schon einmal ausgeführt wurde, wird dies über die Rückantwort mitgeteilt. Da jede Überweisung eine eindeutige Nonce enthält, sind alle Überweisungen eindeutig zu unterscheiden. Die Serviceoperation *creditAccount* ist im Anhang in Abbildung 9.3 zu sehen. Die internen Operationen *credit* und *debit* sind ebenfalls im Anhang in Abbildung 9.5 und Abbildung 9.4.

Sicherheitsbetrachtung

Eine wichtige Sicherheitsanforderung an die Onlinebanking-Anwendung ist:

> *Der selbe Betrag der zu einem Konto hinzugebucht wurde, wurde von einem anderen Konto abgebucht.*

Für eine Überweisung zwischen zwei Banken wird TLS und ein zusätzliches Authentifizierungstoken verwendet. Durch TLS kann ein Angreifer die gesendeten Nachrichten sowie das übertragene Authentifizierungstoken nicht erfahren und nicht manipulieren. Trotzdem ist es sehr leicht Fehler zu machen, sogar in sehr kurzen Protokollen [55].

Tatsächlich ist bei der modellierten Anwendung ein Angriff durch das aktive verzögern von Nachrichten möglich. Die Idee ist es den Recovery-Mechanismus für fehlgeschlagene Überweisungen auszunutzen und eine doppelte Rückerstattung zu erhalten. Der Angreifer, der ebenfalls ein Konto bei einer der Banken besitzt, tätigt eine Überweisung zu einem nicht existierenden Konto. Die Bank des Empfängers informiert die Bank des Angreifers über den Abbruch der Überweisung (siehe `success == false` in Schritt (7) in Abbildung 4.12). Der Angreifer verzögert den Empfang dieser Nachricht bis der Recovery-Mechanismus (11) ausgeführt wurde. Der zweite Versuch die Überweisung auszuführen wird ebenfalls fehlschlagen, aber diesmal mischt sich der Angreifer nicht ein. Somit wird der Betrag, dem Konto des Angreifers rückerstattet. Anschließend sendet der Angreifer die verzögerte Nachricht vom ersten Versuch und der Betrag wird ein zweites Mal rückerstattet. Das TLS-Protokoll stellt zwar die korrekte Nachrichtenreihenfolge in einer TLS-Sitzung sicher, aber die Sicherheit einer Anwendung umfasst mehr als die Sicherheit einer Sitzung und kann somit von der Nachrichtenreihenfolge von verschiedenen TLS-Sitzungen abhängen. Eine Lösung wäre, direkt nach dem Empfangen der Antwortnachricht zu überprüfen ob die Überweisung *ut* in der Key-Value-Liste der unvollständigen Überweisungen *iTransactions* noch enthalten ist. Wenn der Eintrag nicht in der Key-Value-Liste enthalten ist (da die Rückantwort schon bearbeitet und der Eintrag entfernt wurde), dann soll das Protokoll abgebrochen werden damit der Betrag nicht doppelt überwiesen wird. Die korrigierte Version ist im Anhang in Abbildung 9.2 zu sehen.

Wurde der Fehler einmal erkannt ist es einfach diesen zu lösen. In dem oben beschriebenen Fall musste einfach eine Anweisung verschoben werden. Der Angriff

zeigt jedoch, dass das Zusammenspiel zwischen existierenden kryptographischen Protokollen, Serviceverhalten und einem aktiven und böswilligen Angreifer, die Entwicklung von sicheren Anwendungen schwierig macht. Um die Sicherheit von solchen Anwendungen zu überprüfen sind formale Methoden zu empfehlen.

Die hier vorgestellte Onlinebanking-Variante verwendet nur Kontonummer und Passwort zum Autorisieren von Überweisungen. Wenn auf dem Heimcomputer Malware vorhanden ist, kann ein Angreifer die Zugangsdaten erfahren und somit fremde Überweisungen tätigen. Die hier betrachtete Anwendung stellt somit nicht sicher, dass die Benutzer kein Geld verlieren. Diese würde auch bei der Verifikation herauskommen, da der Heimcomputer als unsicheres Terminal modelliert wurde. Bei einigen Bezahldiensten wird diese Art der Authentifizierung und Autorisierung immer noch akzeptiert. Der neueste Stand der Technik ist jedoch Zwei-Faktor-Authentifizierung, wodurch noch ein zweiter Faktor wie z.B. eine Chipkarte oder ein Handy verwendet werden. Eine Onlinebanking-Variante mit Chipkarte beschreibt die Debitcard-Anwendung (siehe Abschnitt 9.1 im Anhang). Eine sehr populäre Zwei-Faktor-Authentifizierung ist mobile TAN (m-TAN). Hierbei wird eine einmalige Transaktionsnummer (TAN) mit den Überweisungsdaten per SMS an ein separates Gerät des Kontobesitzers geschickt. Dadurch können die Überweisungsdaten nochmal überprüft werden und die TAN wird als zusätzliches Geheimnis zur Bestätigung genau dieser Überweisung verwendet. Die hier vorgestellte Onlinebanking-Variante kann mittels externer Services um m-TAN erweitert werden (siehe Abschnitt 4.5).

4.2.4 Fallstudie ETicket

ETicket ist ein elektronisches Ticketsystem, welches eine Kombination aus Smart Card- und Web Services-Anwendung darstellt. Ein Benutzer kann mit dieser Anwendung Zugtickets online über einen Heimcomputer mit Chipkartenleser kaufen. Die Tickets werden auf einer Chipkarte gespeichert und können mehrfach kontrolliert werden. Nur ein echtes Schaffnergerät ist in der Lage Tickets zu validieren und zu entwerten. Zusätzlich kann nur der Kartenbesitzer Tickets kaufen sowie gekaufte Tickets anzeigen oder löschen.

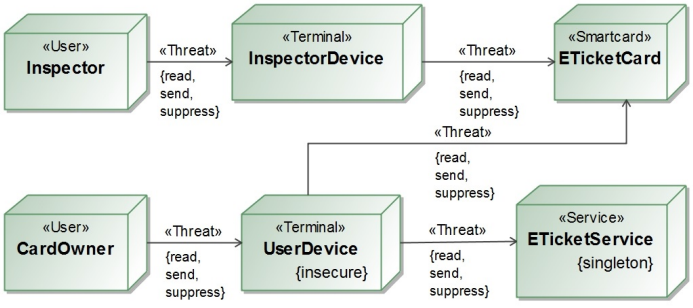

Abbildung 4.13: Deploymentdiagramm der ETicket-Fallstudie

Abbildung 4.13 zeigt das Deploymentdiagramm des elektronischen Ticketsystems. Es gibt zwei Arten von validen Benutzern, die mit dem System interagieren können: die Schaffner *Inspector* und die Kartenbesitzer *CardOwner*. Jeder Kartenbesitzer besitzt eine *ETicketCard* und kann an seinem Heimcomputer *UserDevice* Tickets kaufen. Dazu muss sich der Heimcomputer zum *ETicketService* verbinden. Ein Ticket wird durch den *ETicketService* erstellt und solange aufbewahrt, bis das Ticket von der Karte empfangen und gespeichert wurde. Jeder Schaffner besitzt ein Schaffnergerät *InspectorDevice* zum Validieren und Entwerten von Tickets auf echten Karten. Dies geschieht offline, weshalb keine Verbindung zum *ETicketService* benötigt wird. Von allen Komponenten (bis auf *ETicketService*) können beliebig viele Instanzen existieren. *ETicketService* kann nur genau einmal instanziiert werden (*singleton)*, weshalb alle *UserDevices* mit dem selben Service kommunizieren. Es wird ein Dolev-Yao [31] Angreifer angenommen, der Nachrichten lesen, senden und unterdrücken kann. Das Klassendiagramm dieser Anwendung ist im Anhang in Abbildung 9.6 zu sehen und die modellierten Nachrichten in Abbildung 9.7 und 9.8.

Schlüsselaustausch

Die vom *ETicketService* erstellten Tickets werden über den Heimcomputer eines Kartenbesitzers an seine Karte übertragen. Damit ein Kartenbesitzer diese Tickets nicht duplizieren bzw. sich eigene Tickets erstellen kann, ist eine End-zu-End-Verschlüsselung zwischen *ETicketService* und *ETicketCard* notwendig. Diese kann nicht mit TLS erreicht werden, weshalb ein sicheres Protokoll modelliert werden muss.

Abbildung 4.14 beschreibt einen Schlüsselaustausch zwischen *ETicketService* und *ETicketCard* um den Nachrichtenaustausch zwischen den zwei Komponenten zu verschlüsseln. Das Verhalten des Services wird nicht durch Serviceoperationen modelliert, da es sich hierbei nicht um eigenständige Funktionen handelt, die innerhalb der Anwendung wiederverwendet werden. Die gesamte Kommunikation zwischen *ETicketService* und *ETicketCard* findet über den Computer *UserDevice* statt. Dieser leitet lediglich die empfangenen Nachrichten weiter, da eine Chipkarte einen Computer mit Chipkartenleser und Internetverbindung braucht, um mit einem Service zu kommunizieren. Da es nur einen *ETicketService* gibt, besitzen alle Karten initial neben ihrem privaten Schlüssel (*privateKeyCard*) den öffentlichen Schlüssel *publicKeyService* vom *ETicketService* (siehe Abbildung 9.6 im Anhang). Da es jedoch eine Vielzahl von Karten gibt und neue Karten jederzeit ausgestellt werden können, müsste der *ETicketService* sich alle öffentlichen Kartenschlüssel in einer Liste merken und diese Liste müsste bei jedem Ausstellen einer Karte aktualisiert werden. Um dies zu vermeiden, besitzt jede Karte initial ein Zertifikat *certCard*, welches von einer vertrauenswürdigen dritten Partei (Zertifizierungsstelle) ausgestellt wurde. Ein solches Zertifikat enthält den öffentlichen Schlüssel der Karte und kann mit dem öffentlichen Schlüssel der Zertifizierungsstelle verifiziert werden. Deshalb besitzt der *ETicketService* neben seinem privaten Schlüssel *pri-*

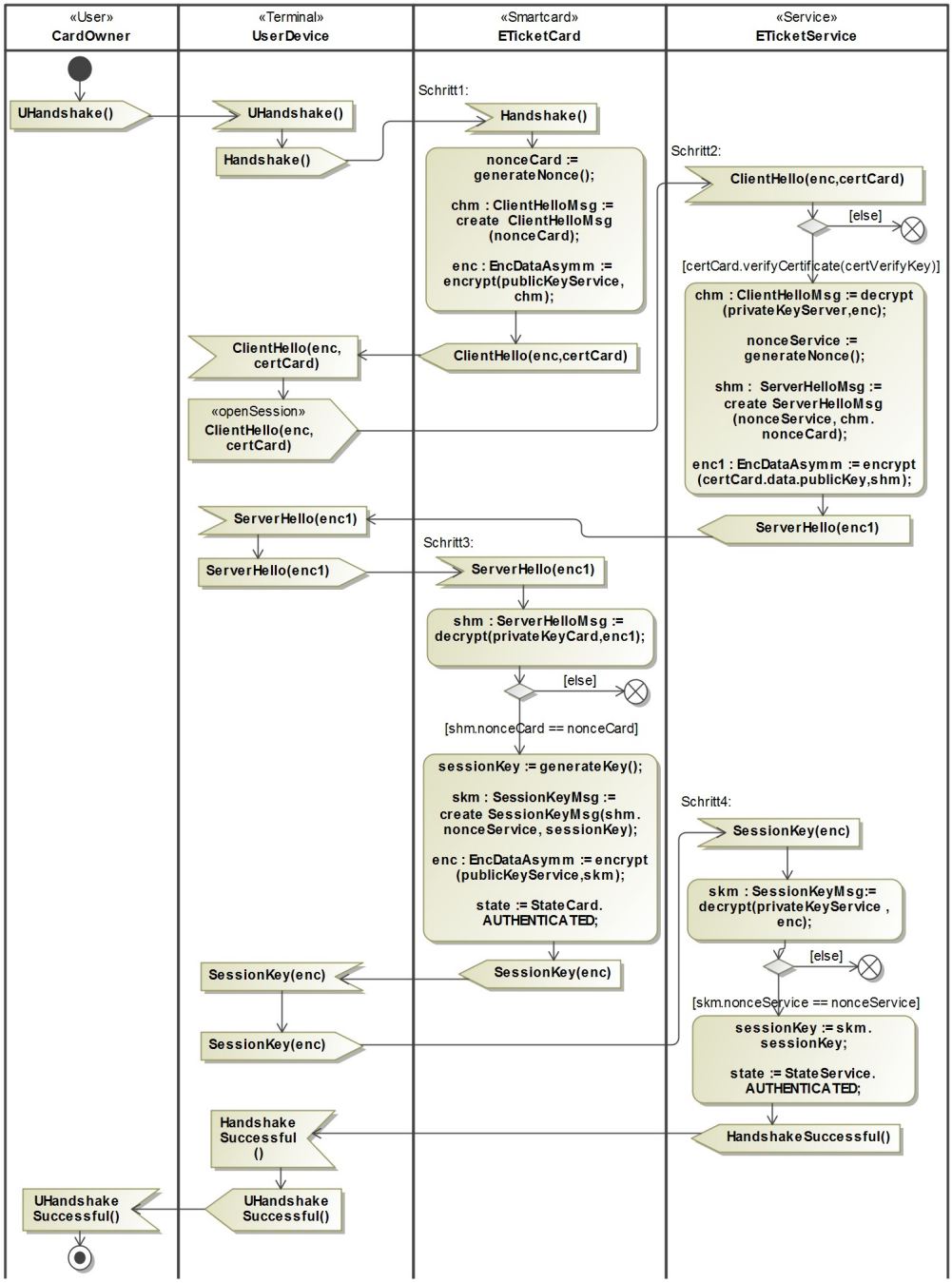

Abbildung 4.14: Schlüsselaustausch zwischen Karte und Service

vateKeyService den öffentlichen Schlüssel *certVerifyKey* zum Überprüfen der Zertifikate. Da diese Zertifizierungsstelle somit die öffentlichen Schlüssel der modellierten Komponenten signiert, kann sichergestellt werden, dass der *ETicketService* nur mit gültigen Karten kommuniziert. Das Schlüsselaustauschprotokoll besteht aus den folgenden vier Protokollschritten:

- Schritt 1 (Handshake): *ETicketCard* generiert eine Nonce (*nonceCard*) und schickt diese verschlüsselt mit dem öffentlichen Schlüssel *publicKeyService* und ihrem Zertifikat an *ETicketService*.

- Schritt 2 (ClientHello): *ETicketService* verifiziert das Zertifikat mit dem öffentlichen Schlüssel der Zertifizierungsstelle (*certVerifyKey*) und entschlüsselt die verschlüsselte Nonce mit seinem privaten Schlüssel *privateKeyService*. Daraufhin erstellt er eine neue Nonce *nonceService* und verschlüsselt beide Nonces mit dem im Zertifikat enthaltenen öffentlichen Schlüssel der Karte und schickt die verschlüsselten Nonces an die Karte zurück.

- Schritt 3 (ServerHello): *ETicketCard* entschlüsselt die empfangene Nachricht und überprüft, ob die vorher erstellte Nonce (*nonceCard*) mit der empfangenen *nonceCard* übereinstimmt. Ist dies der Fall, so wird ein symmetrischer Schlüssel *sessionKey* erstellt und zusammen mit der vom *ETicketService* erstellten Nonce (*nonceService*) mit dem öffentlichen Schlüssel von *ETicketService* verschlüsselt und an *ETicketService* übertragen. Bevor die Nachricht abgeschickt wird, wird noch der Status (*state*) auf *AUTHENTICATED* gesetzt, da der Schlüsselaustausch für die Karte abgeschlossen ist.

- Schritt 4 (SessionKey): *ETicketService* entschlüsselt die Nachricht mit seinem privaten Schlüssel *privateKeyService*. Überprüft ob die enthaltene Nonce mit der im Schritt 2 erstellten Nonce übereinstimmt. Wenn dies der Fall ist, speichert er sich den symmetrischen Schlüssel *sessionKey* und setzt seinen Status ebenfalls auf *AUTHENTICATED*.

Dieses Protokoll beschreibt einen Schlüsselaustausch mit gegenseitiger Authentifizierung. Dazu werden Zertifikate, asymmetrische Verschlüsselung sowie Nonces verwendet. Gute Protokolle sind minimal und somit übersichtlich. Ob ein Protokoll redundante Sicherheitsmaßnahmen besitzt, die das Protokoll unnötig kompliziert machen oder ob Angriffe möglich sind, weil bestimmte Sicherheitsmaßnahmen fehlen ist oft sehr schwierig zu entscheiden. Ist z.B. die *nonceService* wirklich notwendig?

Damit die Protokollschritte innerhalb eines Aktivitätsdiagramms nicht in beliebiger Reihenfolge ausgeführt werden können (z.B. Schritt3 vor Schritt1 oder zweimal hintereinander Schritt3) wird die Eigenschaft *fixMessageOrder* verwendet. Die Protokollreihenfolge zwischen den Aktivitätsdiagrammen wird mit Zuständen (wie z.B. *AUTHENTICATED*) sichergestellt. Damit nach einem modellierten Protokollabbruch (*FlowFinalNode*) oder einem internen Protokollabbruch (*decrypt* schlägt

fehl, wegen einem falschen Schlüssel) der Zustand wieder zurückgesetzt wird, wird für jede Komponente, die Zustände verwaltet, eine Ausnahmebehandlungsroutine modelliert.

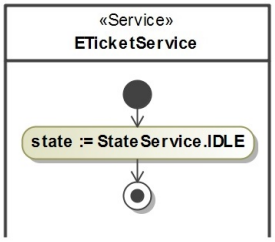

Abbildung 4.15: Modellierte Ausnahmebehandlungsroutine für *ETicketService*

Abbildung 4.15 zeigt eine solche Ausnahmebehandlungsroutine für *ETicketService*. Diese kann auf alle Klassenattribute von *ETicketService* zugreifen und somit auch den Status auf den initialen Wert *IDLE* setzen. Somit kann erzwungen werden, dass das Schlüsselaustauschprotokoll erfolgreich ausgeführt werden muss, damit ein Ticket gekauft werden kann.

Ticket kaufen

Das Kaufen eines Tickets ist in zwei separat aufrufbare Aktivitäten aufgeteilt. In der Ersten wird ein Ticket bezahlt, vom *ETicketService* erstellt und zwischengespeichert. In der Zweiten wird das Ticket an die Karte übertragen, dort gespeichert und anschließend aus dem Speicher des *ETicketServices* gelöscht. Dadurch wird sichergestellt, dass ein Ticket welches noch nicht auf der Karte angekommen ist, weil die Verbindung zum Service abgebrochen ist oder der Angreifer Nachrichten unterdrückt hat, zu jeder Zeit abgerufen werden kann.

Ticket bezahlen

Abbildung 4.16 beschreibt den ersten Schritt. Dazu gibt der Benutzer seine PIN ein sowie die Ticketinformationen *ticketInfo*, welche unter anderem den Startbahnhof und den Zielbahnhof beinhalten. Diese Informationen werden anschließend an die Karte geschickt. Damit eine valide Karte einen Ticketkauf beginnt werden vier Überprüfungen gemacht. Erstens, ob der Kartenzustand *AUTHENTICATED* ist und somit ein Schlüsselaustausch zwischen Karte und Service stattgefunden hat. Zweitens, wird die übergebene PIN mit der auf der Karte gespeicherten PIN verglichen. Drittens, wird durch das Klassenattribut *waitingForTicket* sichergestellt, dass kein Ticket zum Abholen bereitsteht und viertens, ob Platz zum Speichern eines Tickets vorhanden ist (*tickets.hasFree()*). Ist das erfüllt, wird ein Ticketkauf begonnen. Damit eine Karte keinen zweiten Ticketkauf beginnen kann solange der erste nicht abgeschlossen ist, wird *waitingForTicket* auf *true* gesetzt. Anschließend wird eine Transaktionsnummer (*transactionId*)

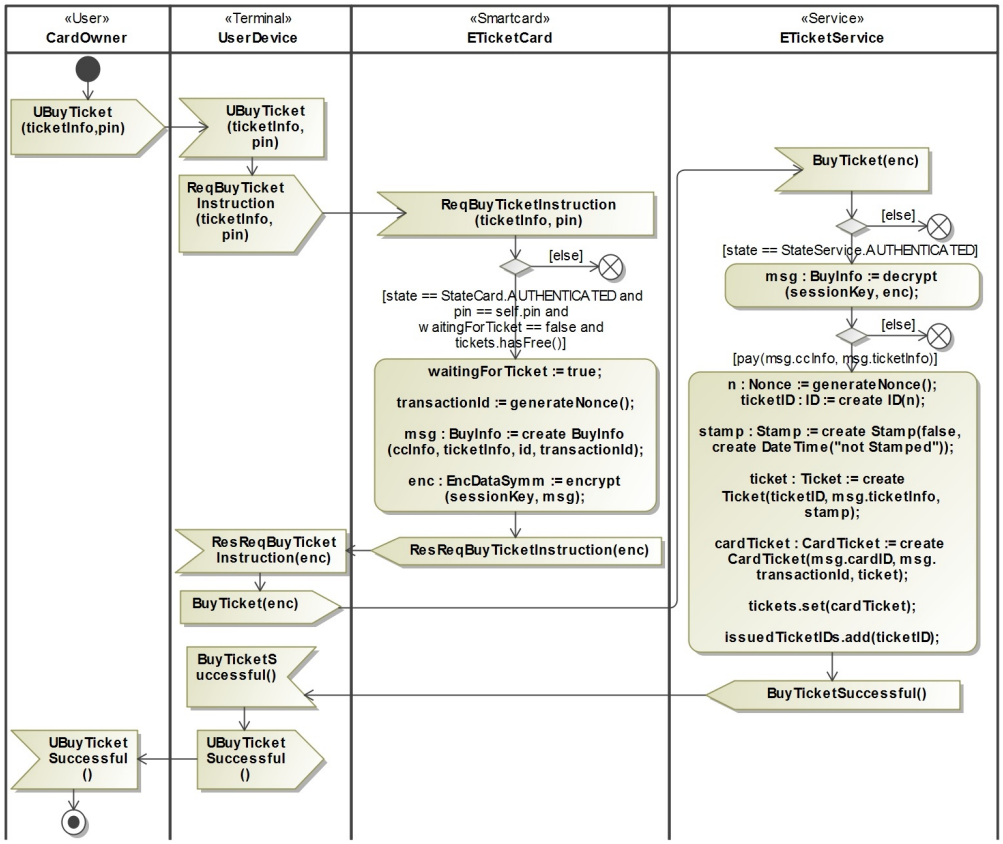

Abbildung 4.16: Ticket bezahlen

erstellt. Diese identifiziert einen Ticketkauf eindeutig. Daraufhin werden die Kreditkarteninformationen (*ccInfo*), die Ticketinformationen (*ticketInfo*) sowie eine Kartenidentifikationsnummer (*id*) und die erstellte Transaktionsnummer (*transactionId*) mit dem zuvor ausgetauschten symmetrischen Schlüssel (*sessionKey*) verschlüsselt und an den *ETicketService* geschickt. Der Service überprüft zuerst ob er im Zustand *AUTHENTICATED* ist und entschlüsselt anschließend die empfangene Nachricht. Anhand der Kreditkarteninformationen (*ccInfo*), der Ticketinformationen (*ticketInfo*) und der Operation *pay*, wird das Ticket bezahlt. Nach erfolgreicher Bezahlung wird das Ticket erstellt. Das Ticket (*ticket*) enthält eine noch nie verwendete Ticketidentifikationsnummer (*ticketID*) (dies geschieht durch die Verwendung von Nonces), sowie die empfangenen Ticketinformationen und ein nicht gestempeltes Stempelobjekt. Aus diesem Ticket, der empfangenen Kartenidentifikationsnummer sowie der empfangenen Transaktionsnummer wird ein Kartenticket (*cardTicket*) erstellt und anschließend in der Key-Value-Liste *tickets* mit der Kartenidentifikationsnummer als Schlüssel (siehe Abbildung 9.6 im Anhang) gespeichert. Diese Key-Value-Liste enthält alle noch nicht abgeholten Tickets. Zum Schluss wird die Ticketnummer in einer Liste von ausgestellten Tickets gespeichert und der Benutzer wird über den erfolgreichen Kauf des Tickets informiert.

Ticket empfangen

Abbildung 4.17 stellt das Protokoll zum Empfangen eines gekauften, aber noch nicht auf der Karte gespeicherten Tickets dar. Als erstes überprüft die Karte, ob sie ein Ticket erwartet (*waitingForTicket*) und ob sie noch im Zustand *AUTHENTICATED* ist. Wenn dies der Fall ist, dann schickt sie ihre Kartenidentifikationsnummer (*id*) und die im vorherigen Schritt erstellte Transaktionsnummer (*transactionId*) verschlüsselt mit dem ausgetauschten *sessionKey* an den Service. Dieser überprüft ebenfalls ob er im richtigen Zustand ist, entschlüsselt die Nachricht und holt sich anhand der empfangenen Kartenidentifikationsnummer eine Kopie des Kartentickets aus der Key-Value-Liste *cardTicket*. Damit die Tickets von anderen Kartenbesitzern nicht abgeholt werden können ist es wichtig, dass alle Karten unterschiedliche Kartenidentifikationsnummer besitzen. Anschließend wird die Transaktionsnummer in dem Kartenticket mit der empfangenen Transaktionsnummer verglichen und das Ticket wird zusammen mit der Transaktionsnummer und einer neu erstellten Nonce (*nonceDeleteTicket*) mit dem *sessionKey* verschlüsselt und an die Karte geschickt. Die Karte entschlüsselt die Nachricht und vergleicht die empfangene Transaktionsnummer mit der Transaktionsnummer, die beim Bezahlen erstellt wurde. Die Überprüfung der Transaktionsnummer ist besonders wichtig, da ansonsten ein Ticket mehrfach abgeholt werden könnte. Stimmen beide überein, dann wird das Ticket auf der Karte gespeichert und die Kartenidentifikationsnummer zusammen mit der vom Service erstellten Nonce (*nonceDeleteTicket*) verschlüsselt und an den Service übertragen. Vor dem Absenden der Nachricht wird noch der Status auf *IDLE* gesetzt, damit

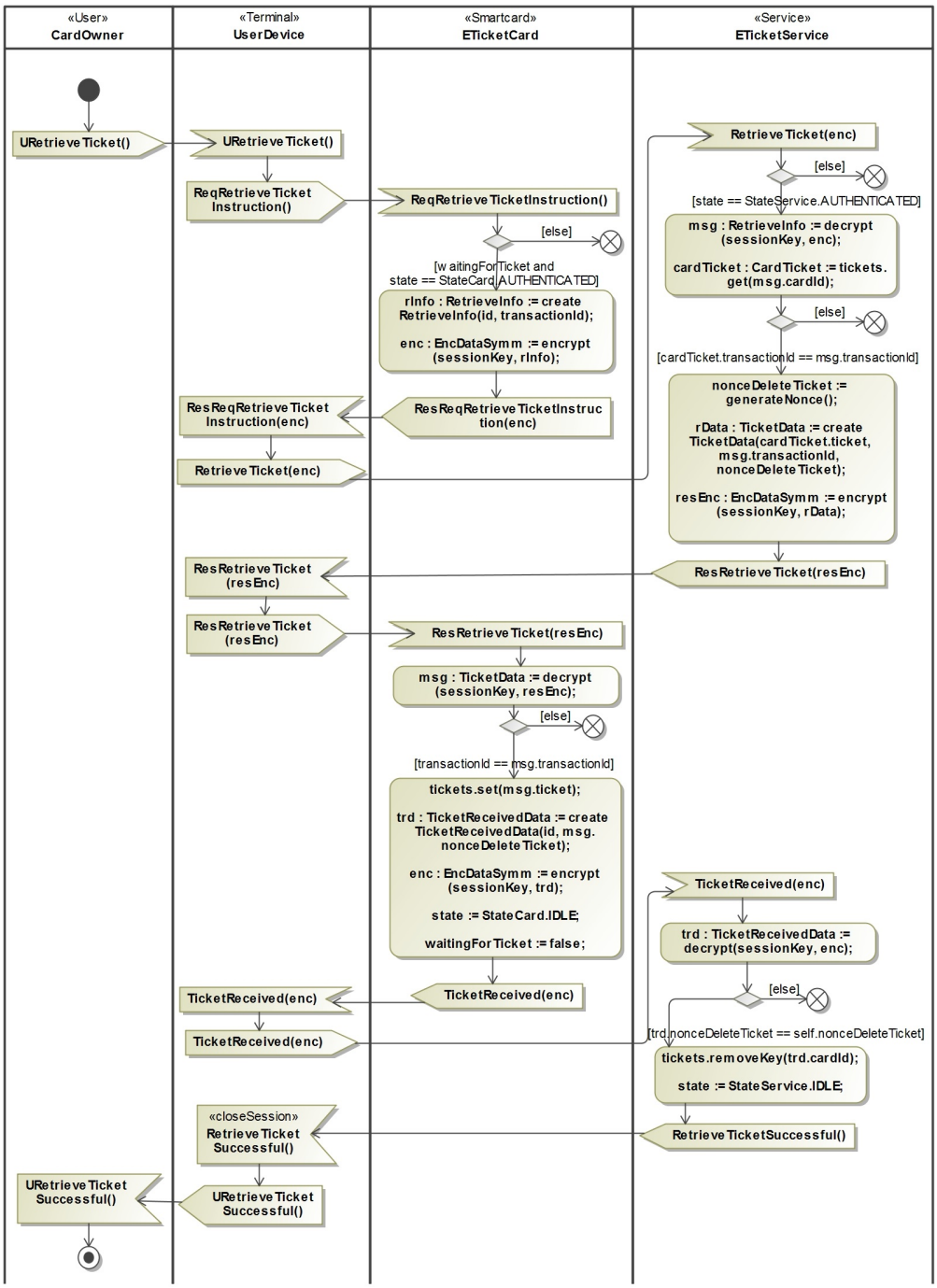

Abbildung 4.17: Ticket empfangen

für einen zweiten Ticketkauf ein neuer Schlüssel ausgetauscht werden muss und *waitingForTicket* auf *false*, da der Ticketkauf nun abgeschlossen ist und ein neuer begonnen werden kann. Der Service entschlüsselt die Nachricht, überprüft die Nonce und weiß somit, dass das Ticket auf der Karte gespeichert wurde und kann dieses nun anhand der Kartenidentifikationsnummer aus seinem Speicher löschen. Abschließend setzt auch dieser seinen Status zurück um Replay-Attacken zu vermeiden und benachrichtigt den Benutzer über den Erfolg des Ticketkaufs. An dieser Stelle wird dann auch die Sitzung, die beim Schlüsselaustausch erstellt wurde, beendet.

Tickets kontrollieren und stempeln

Bevor ein Ticket gestempelt werden kann muss es zuerst kontrolliert werden. Dazu wird eine neu generierte Nonce zusammen mit dem Zertifikat des Schaffnergerätes und der Position des zu kontrollierenden Tickets an die Karte geschickt. Die Karte verifiziert das Zertifikat und speichert sich den im Zertifikat enthaltenen öffentlichen Schlüssel. Anschließend holt sich die Karte das Ticket und verschlüsselt es mit dem gerade gespeichertem Schlüssel. Das Ticket wird verschlüsselt übertragen, damit ein Angreifer die auf der Karte enthaltenen Tickets nicht einsehen kann. Anschließend wird das Ticket zusammen mit der empfangenen Nonce signiert und der Status der Karte geändert, sodass das abgefragte Ticket später gestempelt werden kann. Zum Schluss wird die Nachricht mit der Nonce und dem verschlüsselten Ticket, sowie eine Signatur dieser Nachricht und dem Zertifikat der Karte an das Schaffnergerät geschickt. Dieses überprüft das Zertifikat anhand des gespeicherten öffentlichen Schlüssels der Zertifizierungsstelle, sowie die Signatur anhand dem im Zertifikat enthaltenen öffentlichen Schlüssel und vergleicht die empfangene Nonce mit der zuvor erstellen. Wenn alle Überprüfungen erfolgreich waren, dann ist sichergestellt, dass das empfangene Ticket tatsächlich auf dieser Karte vorhanden ist und die Karte echt ist. Daraufhin wird der Schlüssel aus dem empfangenen Zertifikat für die spätere Kommunikation gespeichert, das Ticket entschlüsselt und die Ticketidentifikationsnummer des zu stempelnden Tickets gemerkt. Zum Schluss wird der Status geändert, sodass das Stempelprotokoll ausgeführt werden kann und die Ticketinformationen wie Start und Ziel sowie ob ein Ticket schon gestempelt wurde werden dem Schaffner angezeigt. Das Protokoll zum Kontrollieren der Tickets ist im Anhang in Abbildung 9.9 zu sehen.

Nachdem ein Ticket kontrolliert wurde und der Schaffner sieht, dass es noch nicht gestempelt wurde, kann dieses Ticket nun gestempelt werden. Der Befehl zum Ticketstempeln enthält die Ticketidentifikationsnummer, einen Stempel und eine neue Nonce. Diese Daten werden verschlüsselt sowie signiert an die Karte geschickt. Durch die Signatur wird sichergestellt, dass ein Angreifer Tickets nicht stempeln und somit ungültig machen kann. Das Wiedereinspielen dieser Nachricht ist zwar möglich, aber ein erneutes Stempeln des Tickets hat keine Auswirkungen, da ein Fehler angezeigt und der ursprüngliche Stempel nicht überschrieben wird. Die Nonce ist dafür zuständig, dass dem Schaffner ein erfolgreicher Stempelvorgang

nicht vorgetäuscht werden kann. Nach einem erfolgreichen Stempelvorgang wird diese Nonce verschlüsselt als Bestätigung zurück an das Schaffnergerät gesendet. Das Protokoll zum Stempeln der Tickets ist im Anhang in Abbildung 9.10 zu sehen.

Tickets verwalten

Die Ticketverwaltung ermöglicht es dem Karteninhaber seine Tickets anzusehen und zu löschen. Dafür wird in beiden Fällen die PIN benötigt, um ein ausspionieren der Zugfahrten und das Entleeren der Karte durch einen böswilligen Angreifer zu verhindern. Die Protokolle zum Verwalten von Tickets sind im Anhang in Abbildung 9.11 und 9.12 zu sehen.

Zusammenfassung Das hier vorgestellte elektronische Ticketsystem für Zugfahrten ist höchst komplex und besonders sicherheitskritisch. Es beschreibt ein System, in dem Smart Cards und Web Services in einer unsicheren Umgebung miteinander interagieren und dazu anwendungsspezifische Sicherheitsprotokolle benötigen. Hierzu wurden Sicherheitsmechanismen wie asymmetrische und symmetrische Verschlüsselung, Signaturen und Zertifikate sowie Nonces verwendet. Zudem wurde die Sicherheit der gesamten Anwendung durch das Sicherstellen von übersichtlichen Hilfszusicherungen realisiert. Dazu zählt, dass nur echte Tickets (Tickets, die vom *ETicketService* ausgestellt wurden) auf einer echten Karte (Karte, die den privaten Schlüssel zu einem vom System ausgestellten Zertifikat besitzt) gespeichert werden können und nur Tickets, die auf einer echten Karte gespeichert sind gestempelt werden. Somit muss beim Stempeln die Echtheit eines Tickets nicht zusätzlich betrachtet werden. Es müssen alle Karten eindeutig unterscheidbar sein, da ein Ticket für eine bestimmte Karte gekauft und nur von dieser empfangen werden kann. Des weiteren muss jeder Ticketkauf innerhalb einer Karte eindeutig sein, um das Duplizieren von Tickets zu vermeiden. Die Modellierung der Sicherheitseigenschaften ist im nächsten Kapitel dargestellt.

4.3 Anwendungsspezifischen Sicherheitseigenschaften

Um eine sichere Web Service-Anwendung zu entwickeln, muss präzise definiert werden, was sicher bedeutet. Dies wird während der Analysephase durch informelle Sicherheitsanforderungen festgehalten. Diese können zwar systematisch durch Anwendungsfälle, Anwendererzählung oder anderen Dokumentationsmethoden beschrieben werden, sind aber immer noch nicht konkret genug, um verifiziert zu werden. Somit gibt es eine große Lücke zwischen informellen Sicherheitsanforderungen und formalen Sicherheitseigenschaften, die zum Verifizieren einer Anwendung benötigt werden. Für das in Abschnitt 4.2.4 vorgestellte elektronische Ticketsystem sind z.B. folgende Sicherheitsanforderungen gewünscht:

- *Nur ausgestellte Tickets werden gestempelt*
- *Tickets werden nicht mehrfach gestempelt*
- *Ausgestellte Tickets können aufgrund von Unterbrechungen nicht verloren gehen*

Da es sich bei der betrachteten Anwendungen um ein verteiltes System handelt, in dem die Systemteilnehmer über unsichere Netzwerke interagieren, sind diese Anforderungen eine Mischung aus funktionalen und nicht funktionalen Anforderungen. Diese Art von Anforderungen lassen sich allein durch Standardsicherheitseigenschaften wie Geheimhaltung, Integrität und Vertraulichkeit oder durch Zugriffskontrolleigenschaften in den meisten Fällen gar nicht ausdrücken, da sie höchst anwendungsspezifisch sind. Durch anwendungsspezifische Sicherheitseigenschaften lassen sich die erwähnten Sicherheitsanforderungen jedoch gut beschreiben und können zur Verifikation verwendet werden. Anwendungsspezifische Sicherheitseigenschaften sind logische Aussagen über die internen Zustände der Komponenten, die nach jedem Protokollschritt gelten sollen, obwohl ein Angreifer versucht diese zu verletzen.

Zudem wird die Modellierung von initialen Eigenschaften sowie Hilfsaussagen unterstützt. Initiale Eigenschaften sind oft notwendig, da viele Sicherheitseigenschaften nicht für eine beliebige Initialisierung gelten. Hilfsaussagen beschreiben Eigenschaften, die bei der Verifikation hilfreich sind, da sie die Gedanken und Konzepte des Modellierers festhalten.

In diesem Kapitel wird eine Erweiterung von OCL zum Beschreiben von anwendungsspezifischen Sicherheitseigenschaften sowie initialen Eigenschaften und Hilfsaussagen gezeigt und anschließend deren Modellierung anhand der ETicket-Fallstudie aus Abschnitt 4.2.4 sowie der Onlinebanking-Fallstudie aus Abschnitt 4.2.3 demonstriert.

4.3.1 Erweitertes OCL

Zum Beschreiben von anwendungsspezifischen Sicherheitseigenschaften sowie initialen Eigenschaften und Hilfsaussagen ist OCL eine geeignete Sprache. OCL ist

Teil der UML-Spezifikation und bezeichnet eine deklarative Sprache zur Spezifikation von Ausdrücken über Objekte eines UML-Modells. Um mit OCL die erwähnten Eigenschaften formal beschreiben zu können, wurde die Sprachmächtigkeit von OCL genutzt um sicherheitsrelevante Sprachkonstrukte wie einen Angreifer und kryptographische Operationen vorzudefinieren und somit anwendungsspezifische Sicherheitseigenschaften modellieren zu können. Hierbei wurde nicht der OCL-Parser verändert, sondern OCL durch UML-Klassen mit Attributen und Operationen erweitert.

Abbildung 4.18: Modellierung des Angreifers

Um Aussagen über das Angreiferwissen in OCL definieren zu können, muss eine Klasse mit dem Stereotypen ≪Attacker≫ modelliert werden. Abbildung 4.18 zeigt eine solche Angreiferklasse. Diese Klasse erhält ein Listenattribut vom Typ *AttackerData*, welches das Angreiferwissen enthält. *AttackerData* ist ein vordefinierter Datentyp und erweitert die Liste der vordefinierten Sicherheitsdatentypen in Abbildung 3.1. Zudem erhält die Angreiferklasse eine statische Operation *getInstance*, die eine konkrete Angreiferinstanz zurückgibt. Mit dieser Instanz kann innerhalb einer OCL-Constraint auf das Angreiferwissen zugegriffen werden.

Um Eigenschaften über kryptographischen Datentypen mit OCL zu formulieren wird das UML-Modell um folgende OCL-Operationen erweitert:

- decrypt(key : SymmKey, data : EncDataSymm) : PlainData
- decrypt(key : PrivateKey, data : EncDataAsymm) : PlainData
- verify(key : PublicKey, signed: SigenedData, data : SignData) : Boolean
- hash(data : HashData) : HashedData
- isKeyPair(pubKey : PublicKey, privKey : PrivateKey) : Boolean
- verifyCertificate(key : PublicKey, data : Certificate) : Boolean

Diese Operationen sind besonders bei initialen Eigenschaften wichtig. Durch diese Erweiterung ist es nun möglich auf Modellebene zu definieren, mit welchem Schlüssel etwas initial verschlüsselt oder signiert wurde, welche Schlüsselpaare zusammengehören oder aus welchen Daten ein zu initialisierender Hash besteht. Damit diese Operationen direkt verwendet werden können ohne die Angabe von Pfaden und Klassennahmen, werden diese Operationen automatisch während den

Transformationen in die Klasse, in der sie verwendet werden, hinzu generiert. Dadurch werden sie vom OCL-Parser erkannt, wodurch dieser nicht verändert werden muss. Durch den Stereotypen ≪OCL-Operation≫ bleibt zudem ersichtlich, dass diese Operationen nur für die OCL-Constraints relevant sind.

4.3.2 Modellierung von Sicherheitseigenschaften der ETicket-Fallstudie

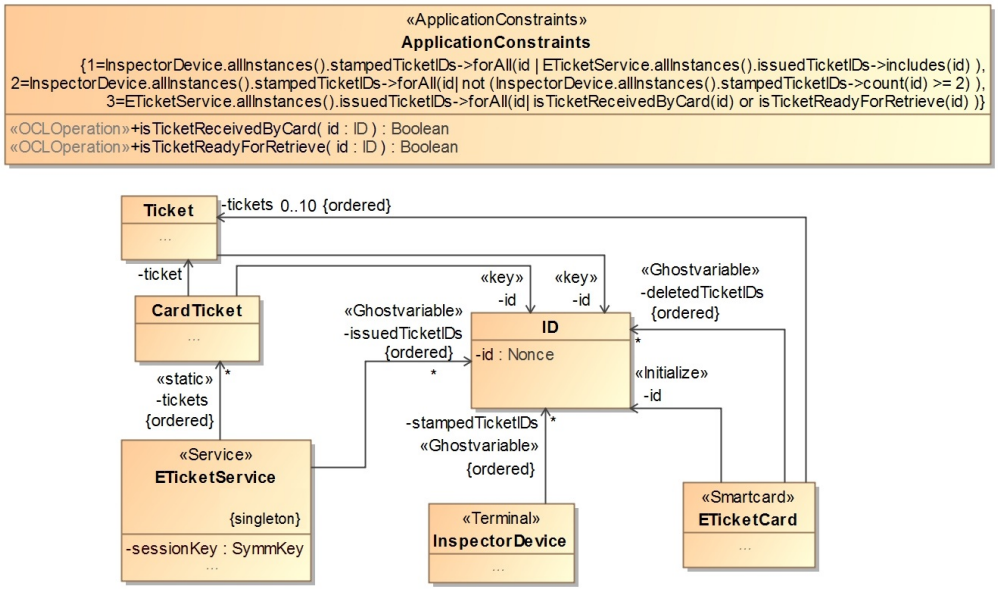

Abbildung 4.19: Modellierung von Sicherheitseigenschaften

Abbildung 4.19 zeigt die drei erwähnten Sicherheitseigenschaften als OCL-Constraints der Klasse *ApplicationsConstraints*. Das darunter liegende Klassendiagramm beschreibt nur die für die Eigenschaften relevanten Klassen. Jede Klasse, die eine Komponente beschreibt (den Angreifer ausgenommen), hat eine beliebige aber endliche Anzahl an Instanzen, falls keine weiteren Einschränkungen modelliert sind. Daher kann die vordefinierte OCL-Operation *allInstances* auf einer Komponentenklasse aufgerufen werden und gibt einen *set* von Instanzen dieser Klasse zurück. Ein *set* ist eine *collection* und in diesem Kontext werden drei Arten unterschieden: *sets* (nicht geordnet und ohne Duplikate), *bags* (nicht geordnet und mit Duplikaten) und *sequences* (Listen, geordnet und mit Duplikaten). Der Punktoperator, welcher gewöhnlich verwendet wird um auf Attribute von einzelnen Instanzen zuzugreifen, kann auch auf *collections* angewendet werden. Dabei handelt es sich um eine Kurzschreibweise um auf jedes Element dieser *collection* zuzugreifen. Daher gibt der Ausdruck *InspectorDevice.allInstances().stampedTicketIDs* eine *bag* von allen *stampedTicketIDs* aller *InspectorDevices* zurück. Mit dem Pfeiloperator

können auf *collections* Funktionen wie *forAll*, *include*, *exclude*, *count*, etc. aufgerufen werden. *forAll* bedeutet, dass für jedes Element der *collection* eine bestimmte Bedingung erfüllt sein muss und *includes* überprüft, ob sich ein Element in der *collection* befindet.

```
inv:
InspectorDevice.allInstances().stampedTicketIDs->
forAll(id | ETicketServer.allInstances().
  issuedTicketIDs->includes(id)
)
```

Listing 4.1: Nur ausgestellte Tickets werden gestempelt

Die erste Sicherheitseigenschaft ist *Nur ausgestellte Tickets werden gestempelt* und ist als OCL-Constraint in Listing 4.1 definiert. Die OCL-Constraint sagt aus, dass alle *stampedTicketIDs* von allen *InspectorDevices* in den *issuedTicketIDs* aller *ETicketServices* enthalten sind. Die Constraint ist sehr ähnlich zu der informellen Eigenschaft. Das ist ein großer Vorteil beim Formalisieren von Sicherheitseigenschaften. Um die Gleichheit zwischen informeller Eigenschaft und formeller OCL-Constraint jedoch zu verstehen ist es wichtig, den Vorgang des Ausstellens und Stempelns zu verstehen. Beim Ausstellen eines Tickets wird ein neues Ticket mit einer eindeutigen Identifikationsnummer (*id* vom Typ *ID*) vom *ETicketServices* erstellt und diese ID zur Liste *issuedTicketIDs* hinzugefügt. Anschließend wird das Ticket vom Typ *Ticket* an die *ETicketCard* geschickt und dort in der Liste *tickets* gespeichert. Beim Stempeln wird das Ticket auf der Karte als gestempelt markiert und das *InspectorDevice* speichert sich die ID des Tickets in seiner Liste *stampedTicketIDs*. Da die Eindeutigkeit der Tickets durch IDs realisiert ist, ist das Speichern der IDs ausreichend. Die Listen *issuedTicketIDs* und *stampedTicketIDs* sind mit dem Stereotyp ≪Ghostvariable≫ annotiert. Das bedeutet, dass diese Variablen nicht im ausführbaren Programmcode verwendet werden, sondern nur für die Spezifikation und Verifikation der Sicherheitseigenschaften benötigt werden. Somit werden diese Variablen nur innerhalb der OCL-Constraints verwendet. Besonders bei Chipkarten (ETicketCard) ist dieses Konzept notwendig, da es sich hierbei um ein ressourcenbeschränktes Gerät handelt, welches nicht in der Lage ist eine beliebige Anzahl an Elementen zu speichern. Deshalb wurde das Konzept von Ghostvariablen schon bei Chipkarten in [62] betrachtet.

```
inv:
InspectorDevice.allInstances().stampedTicketIDs->
forAll(id| not (InspectorDevice.allInstances().
           stampedTicketIDs->count(id) >= 2)
)
```

Listing 4.2: Tickets werden nicht mehrfach gestempelt

Eine weitere Sicherheitseigenschaft ist *Tickets werden nicht mehrfach gestempelt*. Hierbei handelt es sich um die zweite OCL-Constraint in der Klasse *ApplicationConstraints* und ist zudem in Listing 4.2 dargestellt. Es wird geprüft, ob die *collection* aller *stampedTicketIDs* von allen *InspectorDevices* duplikatfrei ist.

Dazu darf jede ID eines gestempelten Tickets nicht zweimal oder öfters in der Vereinigung aller *stampedTicketIDs* vorkommen.

```
inv:
ETicketServer.allInstances().issuedTicketIDs->
forAll(id| isTicketReceivedByCard(id) or
           isTicketReadyForRetrieve(id)
)
```

Listing 4.3: Ausgestellte Tickets können aufgrund von Unterbrechungen nicht verloren gehen

Listing 4.3 zeigt die dritte OCL-Constraint, welche die Sicherheitseigenschaft *Ausgestellte Tickets können aufgrund von Unterbrechungen nicht verloren gehen* formalisiert. Wenn gilt, dass ein Ticket entweder schon von einer Karte empfangen wurde oder noch auf dem Service zur Abholung bereitsteht, dann ist das Ticket nicht verloren. Mit diesem Wissen lässt sich die Eigenschaft wie folgt formulieren. Es soll gelten, dass zu jeder ID aus *issuedTicketIDs* das dazugehörige Ticket entweder auf der *ETicketCard* gespeichert ist oder noch beim *ETicketService* zur Abholung bereitsteht. Die Überprüfungen finden anhand der OCL-Operationen *isTicketReceivedByCard* und *isTicketReadyForRetrieve* statt. Sie besitzen den Stereotyp ≪OCLOperation≫ und dürfen keine Seiteneffekte haben. Hierzu wird die Property *isQuery* der UML-Operation automatisch gesetzt. Solche UML-Operationen, dürfen nur in OCL-Constraints verwendet werden und deren Verhalten wird als *Body Conditions* in OCL definiert. Die *Body Condition* für die OCL-Operation *isTicketReadyForRetrieve(id : ID)* lautet wie folgt:

$$ETicketService.allInstances().tickets.ticket.id- > includes(id)$$

Es werden alle Tickets von allen *ETicketServices* aufgesammelt. Diese Tickets sind vom Typ *CardTicket* (siehe Abbildung 4.19) und enthalten wiederum ein Attribut *ticket* vom Typ *Ticket*, welches wiederum eine ID besitzt. Somit werden durch den oberen OCL-Ausdruck alle IDs aufgesammelt und überprüft, ob die *id*, welche der Operation *isTicketReadyForRetrieve* übergeben wurde in dieser Liste vorhanden ist.

Die *Body Condition* für die OCL-Operation *isTicketReceivedByCard(id : ID)* ist:

$$ETicketCard.allInstances().tickets.id- > includes(id)\ or$$
$$ETicketCard.allInstances().deletedTicketIDs- > includes(id)$$

Das bedeutet, dass ein Ticket von einer Karte empfangen wurde, wenn das Ticket auf der Karte gespeichert ist oder es schon von der Karte gelöscht wurde. Konkreter bedeutet es, dass ein Ticket mit seiner eindeutigen ID von einer Karte empfangen wurde, nur wenn die Liste der gespeicherten Tickets (siehe das Attribut *tickets* von ETicketCard) ein Ticket mit der selben ID enthält oder diese ID in der Liste *deletedTicketIDs* enthalten ist. Durch diese OCL-Operationen ist

es möglich sehr große OCL-Constraints in Teilausdrücke aufzuteilen und diese in unterschiedlichen OCL-Constraints wiederzuverwenden.

Neben den Sicherheitseigenschaften werden für die Verifikation auch initiale Eigenschaften benötigt. Diese beschreiben Eigenschaften, die bei der Initialisierung einer Anwendung gelten müssen. Die meisten Sicherheitseigenschaften gelten nur bei einem bestimmten initialen Systemzustand. Dieser wird durch Attribute mit dem Stereotyp ≪Initialize≫ modelliert. Durch initiale Eigenschaften kann definiert werden, welche Werte diese Attribute annehmen dürfen. Dies ist besonders wichtig für die Verifikation und kann nun ebenfalls im Anwendungsmodell mit OCL in einer Klasse mit dem Stereotypen ≪InitializeConstraint≫ definiert werden.

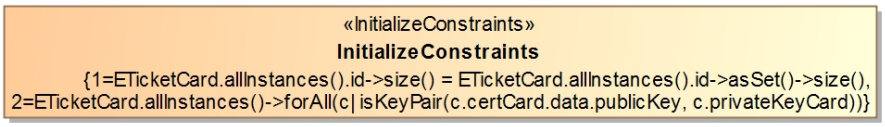

Abbildung 4.20: Initiale Eigenschaften

Abbildung 4.20 zeigt zwei initiale Eigenschaften. Die erste formalisiert, die Eigenschaft *Alle Karten haben unterschiedliche IDs*. Dazu wird überprüft, ob die Anzahl aller IDs mit Duplikaten gleich der Anzahl aller IDs ohne Duplikate ist. Das entfernen der Duplikate geschieht durch die OCL-Operation *asSet*. Diese konvertiert eine *bag* in eine *set*. Die Eigenschaft, dass alle *ETicketCards* unterschiedliche IDs besitzen, ist sicherheitskritisch und wird bei der Verifikation benötigt. Wenn zwei Karten die gleiche ID hätten, könnte man nicht garantieren, dass ausgestellte Tickets nicht verloren gehen können. Ein Ticket, welches vom Service zum Abholen bereitgestellt wird, könnte von einer anderen Karte mit der gleichen ID überschrieben werden. Dies liegt daran, dass das Attribut *tickets*, in dem die Tickets vom *ETicketService* gespeichert werden, eine Key-Value-Liste ist. Diese speichert Key-Value-Paare, wobei ein Key nur einmal vorkommen kann. Daher überschreibt es Elemente mit dem gleichen Key-Wert. In diesem Fall ist der Key die ID der Karte. Dies wird durch den Stereotypen ≪key≫, der auf das Attribut *id* von der Klasse *CardTicket* angewendet wurde (siehe Abbildung 4.19) modelliert. Die zweite initiale Eigenschaft stellt sicher, dass *für jede Karte das gespeicherte Zertifikat den öffentlichen Schlüssel der Karte enthält und somit dieser öffentliche Schlüssel und der private Schlüssel der Karte ein gültiges Schlüsselpaar sind.* *c.certCard.data.publicKey* beschreibt den öffentlichen Schlüssel im Zertifikat und *c.privateKeyCard* den privaten Schlüssel der Karte. Würde diese initiale Eigenschaft für die ETicket-Fallstudie nicht gelten, würden die Sicherheitseigenschaften höchstwahrscheinlich trotzdem gelten, jedoch wäre der Verifikationsaufwand unnötig größer. Es müsste nämlich auch eine falsche Schlüsselinitialisierung betrachtet werden, bei der das System nicht funktionsfähig wäre.

Um die Verifikation von Sicherheitseigenschaften zu erleichtern, können Hilfsaussagen im Modell definiert werden. Diese Aussagen müssen genauso wie die Sicherheitseigenschaften bewiesen werden, beschreiben jedoch einen Zwischenschritt bei der Verifikation. Dieses Vorgehen ist besonders relevant, wenn der Modellierer und der Verifikateur unterschiedliche Leute sind. Der Modellierer verfolgt beim Entwickeln einer Anwendung Konzepte, die für die Verifikation hilfreich sind und somit dokumentiert werden sollen.

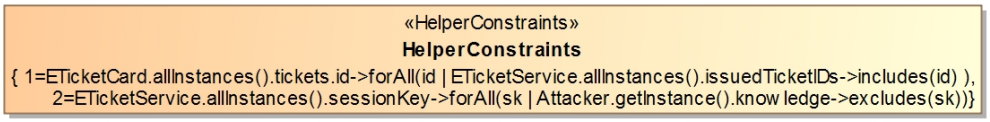

Abbildung 4.21: Hilfsaussagen

Abbildung 4.21 zeigt zwei Hilfsaussagen. Die erste Hilfsaussage formalisiert, *jedes Ticket welches auf einer gültigen Karte gespeichert ist wurde auch ausgestellt* und ist ähnlich zur ersten Sicherheitseigenschaft *(Nur ausgestellte Tickets werden gestempelt)*. Mit dieser Hilfsaussage muss lediglich gezeigt werden, dass nur Tickets von gültigen Karten gestempelt werden. Dieser Zwischenschritt erleichtert die Verifikation erheblich. Da die Tickets alle eine eindeutige ID besitzen, kann die erwähnte Hilfsaussage formalisiert werden, indem alle Ticket IDs von allen *ETicketCards* aufgesammlt werden und überprüft wird, dass jede dieser IDs in der Liste der ausgestellten Tickets *issuedTicketIDs* vorhanden ist. Die zweite Hilfsaussage beschreibt, dass der Angreifer die Sitzungsschlüssel, die auf den Instanzen vom *ETicketService* gespeichert sind, niemals erfährt. Die Formalisierung, dass ein Angreifer etwas nicht erfährt, geschieht mithilfe einer Angreiferklasse (siehe Abschnitt 4.3.1). In diesem Fall erfolgt es mit der Klasse *Attacker* und dem OCL-Ausdruck *Attacker.getInstance().knowledge->excludes(sk)*, wobei *knowledge* das Angreiferwissen darstellt und *sk* den Sitzungsschlüssel. Hilfsaussagen können ebenfalls initiale Eigenschaften enthalten. Die zuvor vorgestellte initiale Eigenschaft, dass *alle Karten unterschiedliche IDs haben* soll nicht nur bei der Initialisierung gelten, sondern während der gesamten Lebensdauer der Anwendungen. Aber nicht alle initialen Eigenschaften sind sinnvolle Hilfsaussagen. Ein Beispiel hierfür wäre, die Eigenschaft, dass alle Listen von gestempelten und ausgestellten Tickets leer sein sollen. Hierbei handelt es sich offensichtlich um eine initiale Eigenschaft, aber keine Hilfsaussage.

4.3.3 Modellierung von Sicherheitseigenschaften der Onlinebanking-Fallstudie

Für die Onlinebanking-Anwendung könnte eine Sicherheitseigenschaft sein, dass kein Geld verloren gehen darf. Konkreter bedeutet dies, dass die Summe von allen Kontoständen konstant bleibt. Der entsprechende OCL-Ausdruck wäre:

```
Bank.allInstances().accounts.balance->sum() = C
```

Wobei C eine unspezifizierte Konstante ist.

4.4 Abstraktionen

Im Rahmen dieser Arbeit wurde der SecureMDD-Ansatz um Model Checking erweitert. Der Vorteil von Model Checking ist, dass Sicherheitslücken automatisch gefunden werden können. Ein erheblicher Nachteil ist jedoch, dass abhängig von der Größe der Anwendung der zu überprüfende Suchraum sehr groß werden kann. Das führt dazu, dass viele Ressourcen (insbesondere Zeit) benötigt werden und Model Checking somit oft nicht praktikabel wäre. Die gängige Praxis, um dieses Problem zu umgehen, ist die manuelle Abstraktion der Anwendungsmodelle [5]. Jedoch benötigt der hier vorgestellte SecureMDD-Ansatz sehr detaillierte Modelle, um daraus lauffähigen Programmcode automatisch zu generieren. Die Lösung für diese Problem ist, dass die bestehenden Modelle, welche die Anwendung beschreiben, unverändert bleiben, jedoch zusätzliche Modelle, welche die Abstraktionen beschreiben unterstützt werden. Aus diesen zusätzlichen Modellen können anschließend Spezifikationen generiert werden, welche z.B. nur einen bestimmten Teilbereich des Systems überprüfen, diesen aber mit den zur Verfügung stehenden Ressourcen vollständig und in akzeptabler Zeit. Somit erhält der Anwendungsentwickler die Möglichkeit, auf Modellebene den Suchraum gezielt einzuschränken. Dadurch wird das Finden von Sicherheitslücken bei sehr großen und komplexen Anwendungen mithilfe von Model Checking im modellgetriebenen Ansatz SecureMDD erst möglich.

Die Modellierung von Abstraktionen erweitert den SecureMDD-Ansatz erheblich. Dieser unterstützt zwei Ansätze zum Testen einer Anwendung. Einerseits können konkrete Angriffe modelliert werden, aus denen Testcode für die generierte Anwendung automatisch erstellt wird [53]. Andererseits kann das Anwendungsmodell in eine formale Spezifikation fürs Model Checking automatisch transformiert werden [18]. Beide Ansätze haben ihre Stärken und Schwächen. Der erste Ansatz ist besonders nützlich für schnelle Regressionstests, jedoch muss der konkrete Angriff modelliert werden. Der zweite Ansatz findet Angriffe automatisch, benötigt jedoch viele Ressourcen. Durch die hier vorgestellte Modellierung von Abstraktionen werden die Vorteile von beiden Ansätzen kombiniert. Die Komplexität des Modells wird eingeschränkt, jedoch ohne den konkreten Angriff vollständig zu spezifizieren.

Im folgenden wird die Modellierung der Einschränkung bezüglich der Protokollabläufe, der Angreiferfähigkeiten sowie der Anzahl der Instanziierungen beschrieben. Da diese Einschränkungen oft notwendig sind, diese jedoch das Verhalten der Anwendung ändern, sodass manche Sicherheitslücken nicht gefunden werden können, liegt der Fokus auf einer einfachen und übersichtlichen Modellierung dieser Einschränkungen.

4.4.1 Einschränkung der Protokollabläufe

Ein Protokollablauf enthält gewöhnlich mehrere eingehende Nachrichten, die in SecureMDD über mehrere UML-Aktivitäten verteilt sind. Diese Aktivitäten können

mit anderen Aktivitäten kombiniert werden um neue Funktionen zu unterstützten ohne den Inhalt der Aktivitäten duplizieren zu müssen. Die Protokollabläufe können in Testdiagrammen eingeschränkt werden, indem nur relevante Aktivitäten verwendet werden, eine bestimmte Ausführungsreihenfolge zwischen den Aktivitäten definiert wird oder die Ausführungshäufigkeit der Aktivitäten eingeschränkt wird. Diese Einschränkungen beschleunigen das Model Checking enorm. Jedoch wird Expertenwissen benötigt, da es durch die Einschränkungen möglich ist, dass einige Angriffe nicht gefunden werden.

Auswahl relevanter Aktivitäten

Große Anwendungen bestehen aus mehreren Funktionen, welche in UML-Aktivitäten modelliert sind. Für einige Sicherheitseigenschaften sind jedoch nur bestimmte Aktivitäten relevant. Welche Aktivitäten relevant sind ist in manchen Fällen offensichtlich. Eine automatische Analyse kann jedoch viele Ressourcen benötigen und im schlimmsten Fall kann die automatische Analyse der relevanten Aktivitäten und die anschließende Überprüfung der Sicherheitseigenschaften genauso lange dauern wie die Überprüfung der Sicherheitseigenschaften anhand des gesamten Modells. In der ETicket-Fallstudie (siehe Abschnitt 4.2.4) sind für die Sicherheitseigenschaft *Ausgestellte Tickets können aufgrund von Unterbrechungen nicht verloren gehen* nur bestimmte Aktivitäten relevant. Damit nur die relevanten Aktivitäten verwendet werden, muss ein Testdiagramm erstellt werden, welches nur diese Aktivitäten besitzt (siehe Abbildung 4.22). Dieses Testdiagramm wird anschließend in eine formale Spezifikation transformiert, welche die nicht relevanten Aktivitäten wie *Ticket stempeln* oder *Ticket anzeigen* entfernt. Das Erstellen und Ändern dieses Diagramms ist sehr einfach im Vergleich zum Ändern der generierten Spezifikation.

Abbildung 4.22: Relevante Aktivitäten zum Kaufen eines Tickets

Einschränkung der Protokollreihenfolge

Eine weitere Möglichkeit die Komplexität zu reduzieren und das Model Checking zu beschleunigen ist die Einschränkung der Protokollreihenfolge. Abbildung 4.23 zeigt sechs ursprünglich unabhängige Aktivitäten mit einer Vielzahl an möglichen Kombinationen, welche zu einem konkreten Protokollablauf eingeschränkt werden, falls der Angreifer nicht interagiert. Diese Einschränkung verringert den Suchraum erheblich und bewirkt somit eine enorme Beschleunigung beim Model Checking. Damit die Sicherheitseigenschaft *Tickets werden nicht mehrfach gestempelt* verletzt werden kann, ist es notwendig ein Ticket zu kaufen und dieses zu stempeln. Ein Model Checker weiß jedoch nicht welche Kombinationen sinnvoller als andere sind.

Mit dem hier vorgestellten Ansatz können diese notwendigen Metainformationen im Anwendungsmodell beschrieben werden.

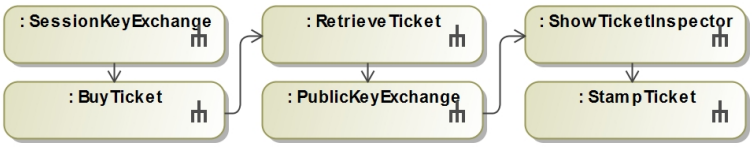

Abbildung 4.23: Ticket kaufen und stempeln

Einschränkung der Ausführungshäufigkeit

Wie oft die Nachrichten innerhalb einer Aktivität empfangen werden können hat ebenfalls großen Einfluss auf die Komplexität und die damit verbundene Ausführungszeit. Die Ausführungshäufigkeit von Aktivitäten kann auf zwei Arten beschrieben werden. Einerseits, kann eine Aktivität genau so oft aufgerufen werden wie sie in einem Testdiagramm vorkommt. Andererseits, kann die Ausführungshäufigkeit aller Aktivitäten innerhalb eines Testdiagramms durch den Stereotypen ≪MaxInvoCount≫ definiert werden. Dieser Stereotyp wird auf das Testdiagramm angewendet. Dies macht die Einschränkung der Ausführungshäufigkeit einfach und flexibel.

4.4.2 Einschränkung der Angreiferfähigkeiten

Die Angreiferfähigkeiten sind ebenfalls ein guter Ansatzpunkt um die Komplexität beim Model Checking von sicherheitskritischen Anwendungen zu reduzieren. Einerseits kann das Deploymentdiagramm geändert werden. Da die Angreiferfähigkeiten auf den einzelnen Kommunikationskanälen definiert sind, gibt es viele Variationsmöglichkeiten. Jede Anwendung besitzt ein solches Deploymentdiagramm. Für jedes Testdiagramm kann ein neues Deploymentdiagramm definiert werden, welches die Angreiferfähigkeiten einschränkt. Ein solches Deploymentdiagramm gilt innerhalb eines Packages für alle enthaltenen Testdiagramme. Durch die Aufteilung von Testdiagrammen in verschiedene Packages kann jedes Testdiagramm andere Angreiferfähigkeiten besitzen. In manchen Fällen kann es jedoch sein, dass die Einschränkung der Angreiferfähgikeiten auf den Kommunikationskanälen zu grob ist. Deshalb wurde der Ansatz erweitert, sodass die Angreiferfähigkeiten, die im Deploymentdiagramm definiert sind, zusätzlich auf den Aktivitäten eingeschränkt werden können. Dies geschieht durch das Anwenden des Stereotypes ≪Threat≫ auf eine Aktivität. Da die Modellierung es auch erlauben würde, die Angreiferfähigkeiten zu erweitern wird beim Transformieren der Modelle überprüft, ob es sich jeweils tatsächlich um Einschränkungen handelt. Durch diesen Ansatz erhält man genug Flexibilität um die Angreiferfähigkeiten einzuschränken ohne den Entwickler zu zwingen dies für jede übertragene Nachricht zu machen.

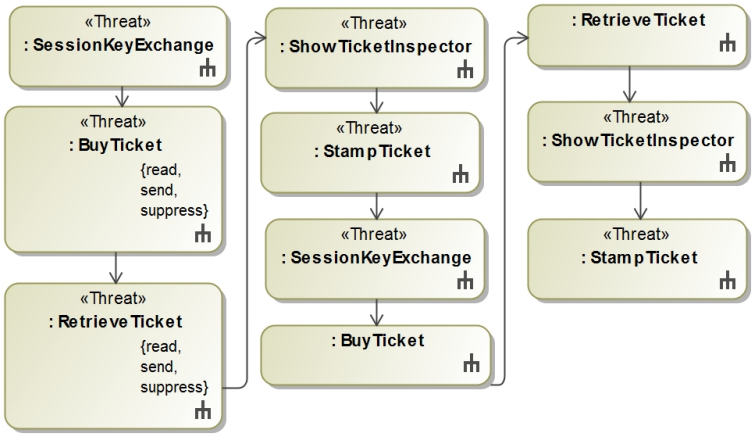

Abbildung 4.24: Eingeschränkte Angreiferfähigkeiten und Protokollabläufe

Abbildung 4.24 zeigt einen konkreten Ablauf in der ETicket-Fallstudie. Hierbei wird ein Sitzungsschlüssel ausgetauscht, ein Ticket gekauft, dieses auf der Chipkarte gespeichert, gestempelt und der ganze Vorgang nochmal wiederholt. Wenn im Deploymentdiagramm ein Dolev-Yao-Angreifer beschrieben wird, also ein Angreifer, welcher auf allen Kommunikationskanälen die Fähigkeiten hat zu lesen, schreiben und zu unterdrücken, dann beschreibt Abbildung 4.24, dass der Angreifer diese Fähigkeiten nur noch auf die in *BuyTicket* und *RetrieveTicket* übertragenen Nachrichten hat. Die Angreiferfähigkeiten können hierbei eine Untermenge von *(read, send, suppress)* sein. Die Anwendung des Stereotypes ≪Threat≫ mit einer leere Menge bedeutet somit, dass der Angreifer keine Fähigkeiten besitzt. Wird der Stereotype gar nicht angewendet, bedeutet dies, dass die Angreiferfähigkeiten die im Deploymentdiagramm definiert sind auch nicht eingeschränkt werden.

Bei dem hier vorgestellten Ansatz handelt es sich um einen feingranularen Ansatz zum Einschränken von Angreiferfähigkeiten. Zudem werden verschiedene Fähigkeiten auf gleichen Aktivitäten zu unterschiedlichen Zeiten unterstützt. Dies ist ähnlich zur Definition von konkreten Testfällen, jedoch mit mehr Freiheitsgraden.

4.5 Integration externer Web Services

Externe Web Services sind Web Services, die unabhängig vom SecureMDD-Ansatz entwickelt wurden. Sie können beliebigen Programmcode ausführen und von Drittanbietern zur Verfügung gestellt werden. Externe Web Services sind essentiell für viele Anwendungen. Zum Beispiel muss ein elektronische Ticketsystem mit verschiedenen externen Web Services interagieren wie z.B. mit einem Bezahlservice, einem Währungsumrechnungsservice oder einem Service mit denen man die Tickets über E-Mail oder MMS (Multimedia Messaging Service) versenden kann. Zudem kann der SecureMDD-Ansatz um anwendungsspezifische Funktionen erweitert werden ohne die Transformationen für Code und formales Modell ändern zu müssen und es können Bibliotheken und Altsysteme durch das Einbetten in einen Web Service integriert werden. Deshalb ist die Integration von solchen Web Services eine wichtige Aufgabe eines modellgetriebenen Ansatzes. Eine naheliegende und einfache Möglichkeit einen externen Web Services aufzurufen ist die manuelle Erweiterung des Programmcodes. Jedoch generiert SecureMDD anhand eines UML-Anwendungsmodells lauffähigen Programmcode sowie formale Spezifikationen zur Verifikation von Sicherheitseigenschaften für diese Anwendung. Die manuelle Erweiterung des generierten Codes würde eine Lücke zwischen dem formalen Modell und dem lauffähigen Programmcode bewirken, sodass die verifizierten Eigenschaften für den generierten Programmcode nicht mehr garantiert werden können. Der hier vorgestellte Ansatz integriert externe Web Services in das Anwendungsmodell, aus dem alles generiert wird, und vermeidet somit diese Lücke. Dafür müssen alle notwendigen Informationen wie die Serviceschnittstelle, die Nachrichtenkonvertierung sowie die Sicherheitsmechanismen abstrakt dargestellt werden, um diese im formal Modell berücksichtigen zu können. Zusätzlich ist es oft notwendig solche Web Services gegen billigere, beliebtere oder effizientere auszutauschen. Die Herausforderung hierbei ist das Austauschen von externen Web Services einfach zu halten und den Verifikationsaufwand zu minimieren.

4.5.1 Kommunikation

Um mit einem Web Service zu kommunizieren benötigt der Client, welcher den Web Service aufruft dessen öffentliche Schnittstelle. SOAP-Web Services definieren diese Schnittstelle in einem WSDL-Dokument, welches die zur Verfügung gestellte Servicefunktionalität mit den verwendeten Nachrichtentypen in maschinenlesbarer Form enthält.

Der SecureMDD-Ansatz nimmt ein solches WSDL-Dokument und transformiert es automatisch in eine UML-Serviceschnittstelle, welches als Modul in das UML-Anwendungsmodell importiert wird (siehe in Abbildung 4.25). Infolgedessen ist der externe Web Service und alle Nachrichtentypen im Anwendungsmodell als Klassen mit Operationen, Attributen und Stereotypen integriert. Das Modell abstrahiert von Informationen wie der Serviceadresse, dem Namensraum sowie den Kodierungsalgorithmen, weil diese irrelevant für die plattformunabhängige Model-

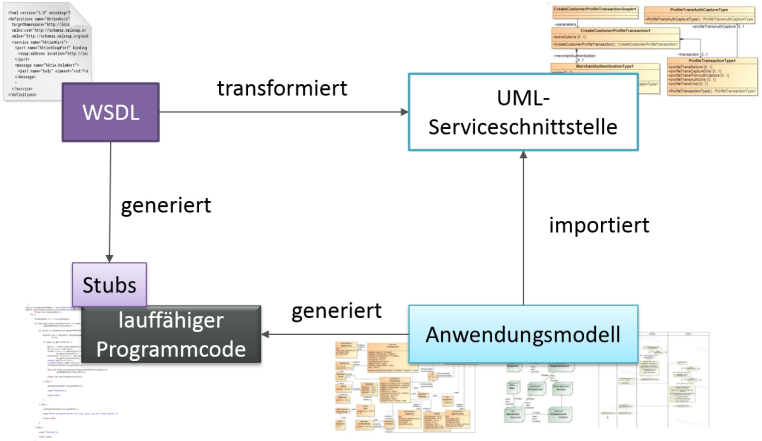

Abbildung 4.25: Übersicht des Transformationsprozesses

lierung sowie für die Verifikation der betrachteten Sicherheitseigenschaften ist. Aufgrund dieser Abstraktionen bleibt das resultierende Metamodell für externe Web Services einfach und kann auch für andere Web Service-Spezifikationssprachen wie WADL verwendet werden. Die Informationen von denen abstrahiert wurde sind jedoch im generierten Programmcode in den Stubs, welche automatisch aus der WSDL generiert wurden vorhanden.

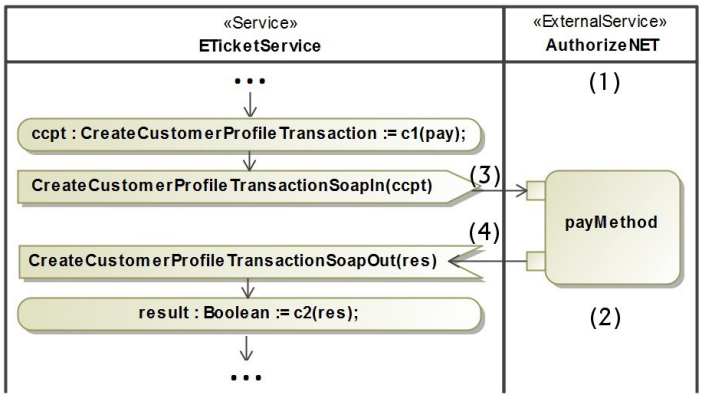

Abbildung 4.26: Aufruf eines externen Web Services

Die Kommunikation mit externen Web Services wird hauptsächlich in Aktivitätsdiagrammen mit der plattformunabhängigen und domänenspezifischen Sprache MEL (siehe Abbildung 4.26) beschrieben. Der externe Web Service wird durch eine Klasse mit dem Stereotyp ≪ExternalService≫ repräsentiert. Diese Klasse wird einer oder mehreren Partitionen (1) zugewiesen und die externen Webmethoden werden durch Operationen der Klasse und CallBehaviorActions ohne das Verhalten zu modellieren dargestellt (2). Der Aufruf einer Webmethode wird mit SendSignal-

Actions (3) und AcceptEventActions (4), die mit einer CallBehaviorAction verbunden sind, modelliert. Für die Konvertierung der Nachrichtendatentypen zwischen der modellierten Anwendung und dem externen Web Service werden Konvertierungsmethoden benötigt. Diese werden in Subaktivitätsdiagrammen mit Hilfe von MEL definiert und müssen dann vor dem Senden an einen externen Web Service und nach dem Empfangen der Antwortnachricht aufgerufen werden. Abbildung 4.26 zeigt wie ein modellierter Web Service (*ETicketService*) den existierenden Bezahldienst *AuthorizeNET* aufruft. Dazu werden die Konvertierungsmethoden *c1* und *c2* verwendet. *c1* konvertiert die Nachricht von der modellierten Anwendung in die von *AuthorizeNET* erwartete Nachrichtenstruktur und *c2* konvertiert das Ergebnis von *AuthorizeNET* wieder zurück. Das bedeutet, dass die Nutzdaten *pay* vom Typ *Pay* und *result* vom Typ *Boolean* zur modellierten Anwendung gehören und *CreateCustomerProfileTransactionSoapIn* und *CreateCustomerProfileTransactionSoapOut* von *AuthorizeNET* genutzten Datentypen sind.

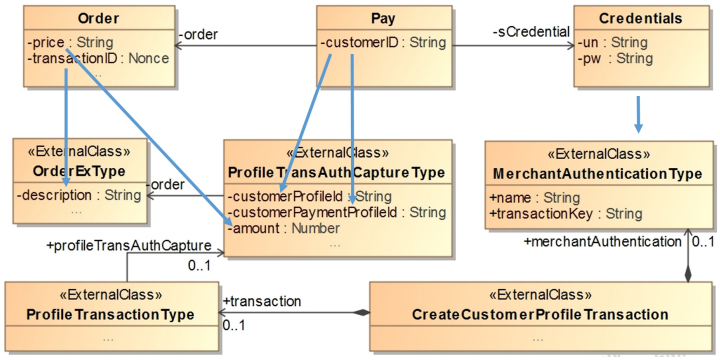

Abbildung 4.27: Konvertierte Nachrichtenklassen

Abbildung 4.28 zeigt die Definition der Konvertierungsmethode *c1*, die das *pay* Objekt vom Datentypen *Pay* in den Datentypen *CreateCustomerProfileTransaction* übersetzt und Abbildung 4.27 illustriert die Abbildung zwischen konvertieren Klassen. Einige Klassen können eins zu eins aufeinander Abgebildet werden (z.B. *Credentials* und *MerchantAuthenticationType*) andere Klassen entstehen aus Informationen, die von verschiedenen Klassen stammen (z.B. *ProfileTransAuthCaptureType* enthält Attribute von den Klassen *Pay* und *Order*) und wenn die Ausgabeklasse mehr Attribute als die Eingabeklasse hat, müssen die fehlenden Informationen aus den existierenden, durch Duplikation oder Transformation, entstehen (z.B. *ProfileTransAuthCaptureType* benötigt eine *customerProfileId* und eine *customerPaymentProfileId* die beide aus der *customerID* extrahiert werden können). Dafür werden generische und vordefinierte Methoden verwendet (z.B. substring für Stringmanipulation). Die Konvertierung in Abbildung 4.28 ist minimal gewählt, aber die Ausgabeklasse *CreateCustomerProfileTransation* hat grob 100 optionale Attribute, die für zusätzliche Leistungen wie z.B. das Berücksichtigen von Rabatten benötigt werden. Somit kann eine solche Konvertierungsmethode sehr groß und

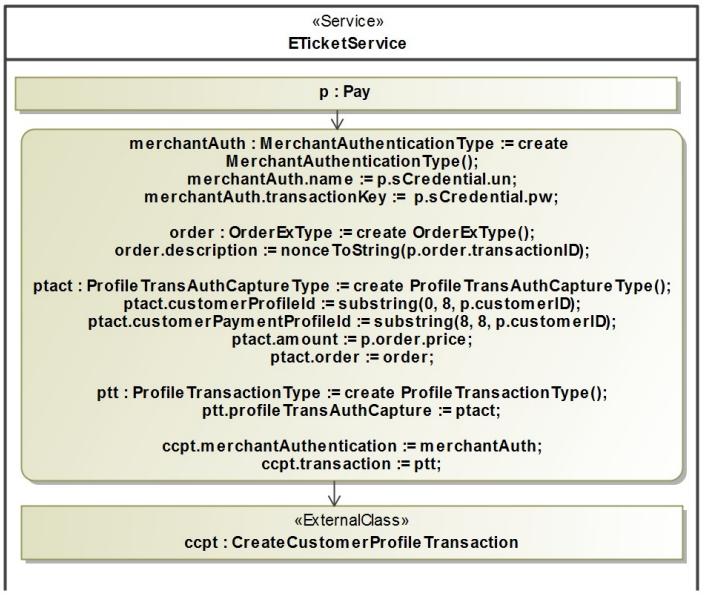

Abbildung 4.28: Konvertierungsmethode *c1*

fehleranfällig werden. Aber da diese Methode modelliert, automatisch ins formale Modell transformiert und bei der Verifikation mitberücksichtigt wird, werden Fehler, die die spezifizierten Sicherheitseigenschaften verletzen im Gegensatz zu einer manuell programmierten Konvertierungsmethode gefunden.

Austauschbarkeit

Sollte es notwendig sein den externen Bezahlservice *AuthorizeNet* aus Abbildung 4.26 gegen einen anderen zu ersetzen, müssten die Protokolldiagramme auch geändert werden und die Sicherheitseigenschaften nochmal bewiesen werden. Dies kann aber durch eine strikte Trennung zwischen den sicherheitskritischen Protokollen und dem Aufruf des externen Web Services vermieden werden. Um das zu erreichen, unterstützt der SecureMDD-Ansatz das Proxy-Entwurfsmuster (siehe Abbildung 4.29). Hierfür muss eine Proxy-Schnittstelle, die vom externen Bezahldienst unabhängig ist, modelliert werden und statt dem Bezahldienst verwendet werden. Für jeden externen Web Service muss ein Proxy modelliert werden, der diese Schnittstelle implementiert und den externen Web Service aufruft. Dann kann der externe Web Services ganz einfach durch das Auswechseln des Proxies im Klassendiagramm ausgetauscht werden. In der betrachteten Fallstudie muss der *ETicketService* eine *PayService* Proxy-Schnittstelle aufrufen. Um einen konkreten Bezahldienst wie *AuthorizeNET* zu verwenden, muss ein neuer Proxy (z.B. *AuthorizeNETProxy*) erstellt werden. Dieser muss von *PayService* erben und definiert das Verhalten der Bezahlmethode *pay* sowie die Konvertierungsmethoden.

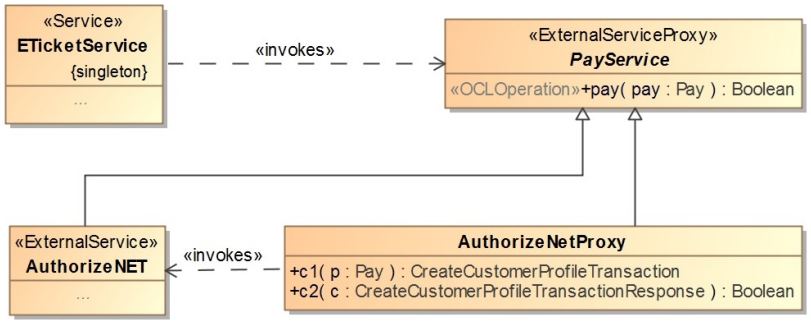

Abbildung 4.29: Proxy-Entwurfsmuster für die Austauschbarkeit von AuthorizeNET

4.5.2 Nachrichtensicherheit

Abhängig von den Sicherheitseigenschaften muss die Kommunikation zwischen dem Client und dem externen Web Service gesichert werden. Da die Sicherheitsmechanismen modelliert werden, können sie ins formale Modell transformiert und bei der Verifikation berücksichtigt werden. Es gibt verschiedene Möglichkeiten eine Kommunikation zu Web Services durch Kryptographie abzusichern. Die einfachste und am häufigsten verwendete Möglichkeit ist die Verwendung von TLS. Dabei handelt es sich um ein Standardprotokoll, welches unabhängig von einem bestimmten Web Service ist. TLS erfüllt aber nicht alle möglichen Anforderungen wie z.B. Ende-zu-Ende-Verschlüsselung. WS-SecurityPolicy ist eine Sprache um individuelle kryptographische Protokolle für Web Services zu beschreiben. Die Entwicklung von anwendungsspezifischen Sicherheitsprotokollen ist aber fehleranfällig und benötigt Verifikation. Zusätzlich ist es wahrscheinlich, dass verschiedene Web Services unterschiedliche WS-SecurityPolicies besitzen, was die Austauschbarkeit von Web Services beeinflusst. Der SecureMDD-Ansatz unterstützt drei unterschiedliche Arten, um die modellierte Funktionalität durch Kryptographie zu sichern.

Transport Layer Security

Die erste Möglichkeit ist die Anwendung von TLS, um die Kommunikation zum externen Service zu sichern. Dies wird durch einen Stereotypen ≪TLS≫ auf den externen Bezahldienst modelliert (siehe Abbildung 4.30). Der Stereotyp hat zwei Eigenschaften um zwischen serverseitiger Authentifizierung (server side authentication *SSA*) und gegenseitiger Authentifizierung (mutual authentication *MA*) zu unterscheiden. Dieser Sicherheitsmechanismus schränkt die Angreiferfähigkeiten wie in Abschnitt 7.2.5 beschrieben ein.

Eigene Protokolle mit vordefinierten Sicherheitsdatentypen

Die zweite Möglichkeit ist die Verwendung von vordefinierten Sicherheitsdatentypen für Verschlüsslung, Signaturen, MACs, Hashes, Nonces, Schlüssel und vorde-

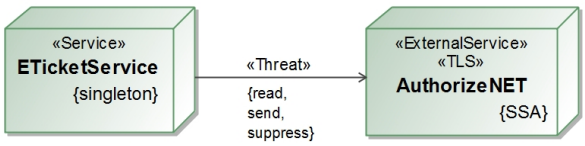

Abbildung 4.30: TLS für die sichere Kommunikation mit einem externen Service

finierte Operationen um diese Datentypen zu erstellen. Diese Sicherheitsmechanismen werden zur Modellierung der Sicherheitsprotokolle zwischen den Systemteilnehmer in der betrachteten Fallstudie verwendet. Da in der Vergangenheit der Fokus von SecureMDD nicht auf der Austauschbarkeit, sondern auf dem Sicherstellen von anwendungsspezifischen Sicherheitseigenschaften war, gibt es keine strikte Trennung zwischen Anwendungslogik und Kryptographie. Deshalb ist diese Möglichkeit unpassend, wenn die Web Services ausgetauscht werden sollen.

WS-SecurityPolicy

Die dritte Möglichkeit die unterstützt wird, um eine modellierte Anwendung abzusichern ist die Verwendung von WS-SecurityPolicy. Hierbei wird Kryptographie direkt vor dem Senden einer Nachricht und direkt nach dem Empfangen einer Nachricht und somit unabhängig von der Anwendungslogik angewendet. WS-SecurityPolicy ist in WSDL integriert, sodass die Policy automatisch aus der WSDL extrahiert und in eine abstrakte UML-Repräsentation transformiert wird, die Stereotypen, Klassen und Attribute verwendet. Zusätzlich abstrahiert die UML-Repräsentation von WS-SecurityPolicy-Assertions wie *AlgorithmSuite*, da diese für die formale Verifikation nicht benötigt werden. Eine WS-SecurityPolicy Spezifikation eines Web Services kann mehrere alternative Policies enthalten, weswegen der Anwendungsentwickler eine auswählen muss, die vom Client verwendet werden soll. Dies wird durch ein Attribut des Aufrufers oder des Proxies modelliert.

Weil Wiederverwendbarkeit Software übersichtlicher und wartbarer macht sowie Fehler reduziert, werden im SecureMDD-Ansatz WS-SecurityPolicies auf die MEL-Notation abgebildet und deren Transformation ins formale Modell wiederverwendet. Die MEL-Ausdrücke, deren Verhalten äquivalent zu den Policies sind werden in das modellierte Aktivitätsdiagramm in dem der Client den externen Web Service aufruft hinein generiert. Dies wird mit Model-zu-Model-Transformationen in QVTo[69] gemacht und das resultierende Modell wird mit dem bereits existierenden Generator in eine formale Spezifikation transformiert.

Abbildung 4.31 und 4.32 zeigen einen Ausschnitt eines Protokolls mit dazu generierter Policy. In dem ursprüngliche Protokoll (ohne die dazu generierte Policy) sammelt der Client die Bezahlinformationen (erster Aktivitätsknoten in Abbildung 4.31) und sendet diese zum Proxy, der die pay-Methode aufruft (letzter Aktivitätsknoten in Abbildung 4.32). Die pay-Methode führt die Konvertierung und den Aufruf des externen Web Services aus. Die dazugehörige WS-SecurityPolicy

beschreibt ein einfaches Sicherheitsprotokoll mit *symmetric binding* und *body encryption*. Das *symmetric binding* benutzt ein X.509 Zertifikat als *protection token*, welches schon ausgetauscht wurde und wird in den Nachrichten durch seine *thumbprint*-Referenz adressiert. Der dazu generierte Teil in Abbildung 4.31 (zweiter Aktivitätsknoten) generiert einen symmetrischen Schlüssel, speichert ihn im *keystore*, um eine optionale Antwort entschlüsseln zu können, verschlüsselt den symmetrischen Schlüssel mit dem öffentlichen Schlüssel aus dem X.509 Zertifikat, welches zum externen Web Services gehört, erstellt einen SOAP-Header der den verschlüsselten symmetrischen Schlüssel enthält, verschlüsselt die Bezahlinformationen mit dem symmetrischen Schlüssel und fügt das alles in ein SOAP-Body Objekt. Der dazu generierte Teil in Abbildung 4.32 (zweiter Aktivitätsknoten) entschlüsselt den symmetrischen Schlüssel aus dem Header und benutzt den symmetrischen Schlüssel um die Bezahlinformationen zu entschlüsseln. Diese werden dann verwendet um die Bezahlmethode *pay* aufzurufen. Der Sende- und Empfangsknoten sind modifiziert, da die ursprünglich modellierten Nachrichten durch die SOAP-Nachrichten beim Transformieren ersetzt werden mussten. Die dazu generierten Abschnitte einschließlich der dazugehörigen Klassen werden automatisch erstellt. Neben dem verschlüsselten symmetrischen Schlüssel enthält der echte SOAP-Header auch Informationen über Algorithmen, die weggelassen werden können und Verweise auf Tokens wie dem *thumbprint* (Fingerabdruck) des öffentlichen Schlüssels, der nicht notwendig ist, wenn die formale Representation des externen Web Services nur ein Schlüsselpaar besitzt. Da das Verhalten der Policy vor der Bezahlmethode aufgerufen wird, können zwei externe Web Services, die beide WS-SecurityPolicies verwenden ohne zusätzlichen Verifikationsaufwand ersetzt werden, wenn die Policy

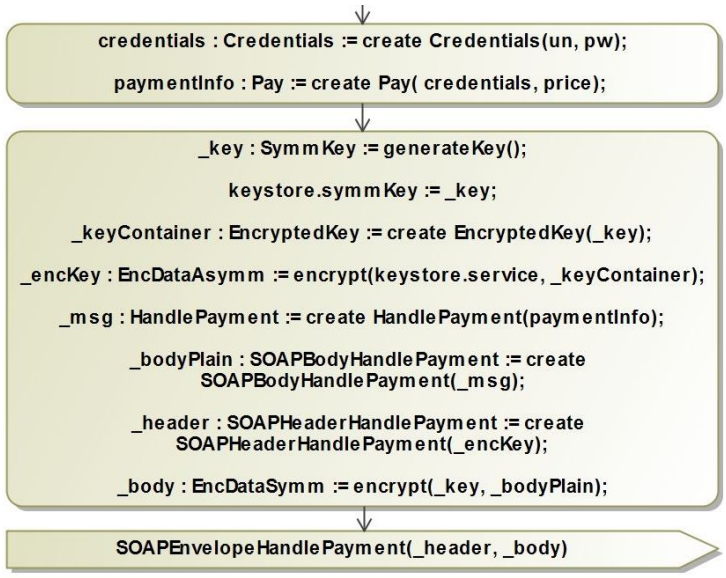

Abbildung 4.31: Übertragung von HandlePayment mit hinzugeneriertem Policy-Verhalten

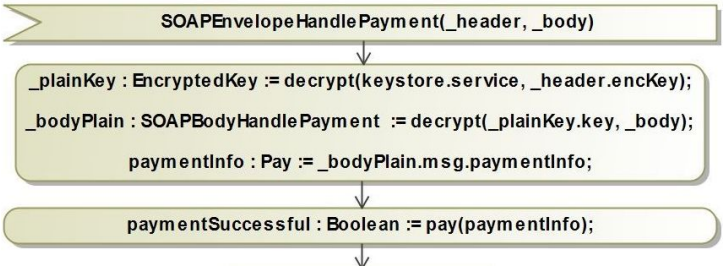

Abbildung 4.32: Empfang von HandlePayment mit hinzugeneriertem Policy-Verhalten

des ersetzten externen Web Services eine Untermenge der Policy des neuen externen Web Services ist. Ob es sich um eine Untermenge handelt kann sehr schnell und automatisch während der QVTo-Transformationen überprüft werden.

4.5.3 Annahmen über den externen Web Service

Für die Verifikation einer Anwendung, die externe Web Services aufruft, sind Annahmen über den externen Web Service notwendig. Ein Sicherheitseigenschaft für ein elektronisches Ticketsystem könnte sein, *Nur bezahlte Tickets werden vom Schaffner gestempelt.* Offensichtlich sind einige Informationen über die externe Bezahlmethode notwendig. Zum Beispiel, dass *wenn der Rückgabewert der Bezahlmethode wahr ist, dann war die Bezahlung erfolgreich oder wird noch erfolgen.* Diese Annahme wird für den Proxy spezifiziert. Dieser ist eine Abstraktion von dem externen Web Service und ist für die Konvertierung zwischen den unterschiedlichen Nachrichtentypen und dem Aufruf des externen Web Services zuständig. Aufgrund dessen ist die Sicherheitseigenschaft unabhängig vom externen Web Service.

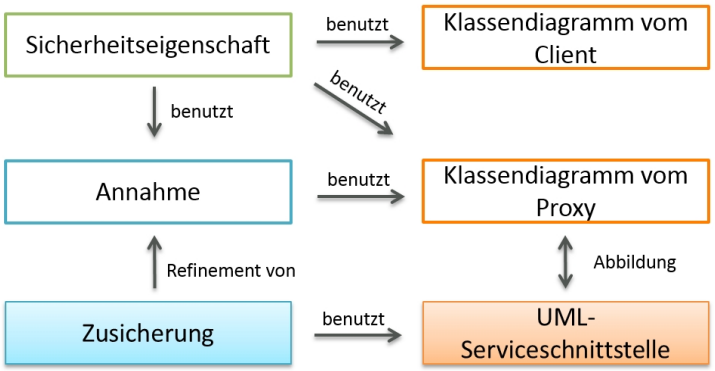

Abbildung 4.33: Zusammenhang zwischen Sicherheitseigenschaften, Annahmen und Klassendiagrammen

Abbildung 4.33 zeigt, dass für die Sicherheitseigenschaft die Klassen vom Client und die vom Proxy sowie die Annahmen des Proxies relevant sind. Die Zusicherung des externen Web Services benutzt Klassen der UML-Serviceschnittstelle, welche

aus der WSDL generiert wurden und die Zusicherung muss ein Refinement der Annahme sein. Die Sicherheitseigenschaft der Anwendung, die Zusicherung des externen Web Services und die Annahmen des Proxies sind mit OCL auf Klassen formal definiert. Diese Klassen beschreiben die internen Zustände und die Nachrichten der Systemteilnehmer. Die Abbildung zwischen den ausgetauschten Nachrichten wir durch die modellierten Konvertierungsmethoden beschrieben. Diese Konvertierungsmethoden werden automatisch in ausführbaren Java-Programmcode sowie eine formale Spezifikation transformiert. Die Beziehung zwischen den internen Zuständen ist nur für die Verifikation notwendig und wird daher nicht modelliert, sondern direkt im formalen Modell spezifiziert. Diese zwei Abbildungen ermöglichen das Refinement zwischen dem externen Web Service und dem Proxy. Wenn der externe Web Service ein Refinement des Proxies ist, dann gilt die Sicherheitseigenschaft die für den Proxy bewiesen wurden auch für den externen Web Service. Falls die Zusicherung eines neuen externen Web Services ebenfalls ein Refinement der Annahme des Proxies ist, dann kann der alte externe Web Service ohne Einfluss auf die Sicherheitseigenschaft durch den neuen ausgetauscht werden. Mehr über die formale Betrachtung von externen Services steht in Abschnitt 7.5.

Annahme

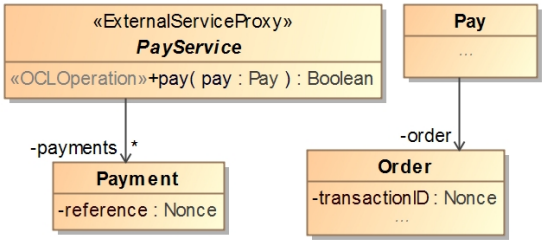

Abbildung 4.34: Ausschnitt des Klassendiagramms von *ExternalServiceProxy*

```
result implies PayService.payments->
  exists(p| p.reference = pay.order.transactionID)
```

Listing 4.4: OCL-Nachbedingung der Bezahloperation *pay*

Abbildung 4.34 stellt einen Ausschnitt des Proxy Klassendiagramms des elektronischen Ticketsystems dar. Um die Annahme zu definieren werden einige interne Zustände benötigt. Deshalb erhält die Proxyschnittstelle *PayService* eine Liste von erfolgreichen Bezahlungen (*payments*). Ein *payment* besteht aus einer Referenz, die eine zufällige und eindeutige Nummer zur Identifizierung der *payments* enthält. Die internen Zustände der Proxy-Klasse sind minimal und enthalten nur die Information die notwendig sind um die Annahme zu definieren. Um die Annahme *wenn der Rückgabewert der Bezahlmethode war ist, dann war die Bezahlung erfolgreich oder wird noch erfolgen* zu definieren, ist es wichtig den Bezahlvorgang zu

abstrahieren. Jedes Ticket, das vom *ETicketService* ausgegeben wurde enthält eine *ticketID*. Diese *ticketID* wird als *transactionID* im *Pay*-Objekt, welches als Eingabeparameter der Bezahlmethode *pay* empfangen wird gespeichert. Die Annahme wird als OCL-Postcondition auf der Bezahlmethode *pay* der Proxyschnittstelle definiert und beschreibt, dass *wenn das Ergebnis wahr ist, dann enthält die Liste der payments einen Eintrag mit der transactionID als Referenz* (siehe Listing 4.4).

4.6 Verwandte Arbeiten

Modellierung von Web Services

Für die Modellierung von Services und serviceorientierten Architekturen existiert bereits der Standard SoaML[1]. Der Fokus liegt auf dem Identifizieren von Services, Abhängigkeiten zwischen Services und deren Anforderungen, dem Definieren von Konsumenten und Verbrauchern sowie dem Definieren von Richtlinien, die beschreiben wie die Services verwendet werden sollen. Hierbei wird jedoch nicht das konkrete Verhalten von Web Services modelliert, aus dem lauffähiger Code generierte werden kann. Zwar ist das Spezifizieren von funktionalen Fähigkeiten, sowie Protokollen und *Message Exchange Patterns* möglich, jedoch auf einer zu abstrakten Ebene. Zudem berücksichtigt diese Modellierungssprache nicht die Sicherheit der modellierten Anwendung gegenüber einem Angreifer.

Ansätze wie [58, 36, 7, 14] beschreiben ebenfalls die Modellierung von Web Services, jedoch betrachten sie ebenfalls keine kryptographischen Protokolle, keinen Angreifer und modellieren nicht das gesamte Verhalten der Services. Deren Fokus liegt auf der Komposition von Services.

Andere Ansätze wie [59, 27, 67], die sich mit der Sicherheit von Services befassen, modellieren ebenfalls nicht das gesamte Verhalten, aus dem lauffähiger Code generiert werden kann.

Die Modellierung mit SecureML unterscheidet sich erheblich von den bestehenden Ansätzen, da aus dem SecureML-Anwendungsmodellen lauffähiger Code generiert wird. Zudem baut diese Arbeit auf der Arbeit von Moebius [62] auf und unterstützt somit die Modellierung von Web Service-Anwendungen die mit Chipkarten über kryptographische Protokolle kommunizieren können. Die Entwicklung von Web Services und Chipkarten in verschiedenen Modellierungssprachen, wäre für den Entwickler unnötig umständlich. Zudem müsste eine Schnittstelle zwischen diesen Sprachen definiert werden.

Modellierung von Sicherheitseigenschaften

Sicherheitsaspekte werden durch Standards wie WS-Security [66] oder WS-SecurityPolicy [65] und Standardsicherheitsprotokollen wie TLS [30] sichergestellt.

[1] http://www.omg.org/spec/SoaML/

Jedoch ist die Verwendung von anwendungsunabhängigen Standards und Protokollen nicht ausreichend um die Sicherheit einer Anwendung zu garantieren. Mit SecureMDD wird die gesamte sicherheitskritische Anwendung, einschließlich des kompletten funktionalen Verhaltens, in UML modelliert. Dadurch können nicht nur Standardeigenschaften wie Geheimhaltung und Integrität, sondern auch anwendungsspezifische Sicherheitseigenschaften wie "es geht kein Geld verloren" verifiziert werden. Diese Eigenschaften müssen somit auch im Anwendungsmodell modelliert werden können.

SecureSOA [60] ist ein modellgetriebener Ansatz zum Modellieren und Sicherstellen von Sicherheitsanforderungen. Der Ansatz verwendet UML-Stereotypen wie ≪User Authentication≫, ≪Non-Repudiation≫ und ≪Trust≫ um Sicherheitsanforderungen zu beschreiben.

Weitere Ansätze wie [78, 67] verwenden ebenfalls standardisierte Schlüsselwörter (wie z.B. Stereotypen) und können somit nur anwendungsunabhängige Sicherheitseigenschaften ausdrücken.

Es existieren jedoch auch andere Ansätze, die nicht auf anwendungsunabhängige Sicherheitseigenschaften eingeschränkt sind.

Pavlidis et al. [72] beschreibt einen solchen Ansatz zur Modellierung von Sicherheitsanforderungen. Der Fokus liegt jedoch nicht auf verifizierbaren Eingenschaften, sondern auf der systematischen Entwicklung von Sicherheitsanforderungen. Die generierten Dokumente sind somit zu ungenau um sie zur Verifikation von Sicherheitseigenschaften zu verwenden.

Ein weiterer Ansatz wurde von Haley et al. [38] vorgestellt. Dieser beschreibt ein Framework für die Auswahl und Analyse von Sicherheitsanforderungen. Für eine existierende Anwendung muss ein Kontext händisch spezifiziert werden und die Sicherheitseigenschaften formal anhand dem Kontext formalisiert. Der SecureMDD-Ansatz spezifiziert keinen Kontext, sondern unterstützt die Modellierung der gesamten Anwendung einschließlich des plattformunabhängigen Verhaltens der Systemteilnehmer.

OCL-Constraints werden jedoch auch in nicht modellgetriebenen Ansätzen verwendet. KeY [11] verwendet OCL zum Spezifizieren von Sicherheitseigenschaften für die Verifikation von objektorientiertem Code. Mit diesem Ansatz ist es möglich anwendungsspezifische Sicherheitseigenschaften mit OCL zu spezifizieren und zu verifizieren. Dieser Ansatz verwendet jedoch objektorientierten Quellcode als Grundlage für die OCL-Constraints und klärt nicht die Frage wie anwendungsspezifische Sicherheitseigenschaften in UML-Modellen beschrieben werden können.

Modellierung von Abstraktionen

Einige modellgetriebene Ansätze [29, 10, 6, 50] verwenden Model Checking zum Finden von Sicherheitslücken in einer modellierten Anwendung. Diese Ansätze modellieren jedoch nicht das gesamte Anwendungsverhalten. Daher besitzen sie viel kleinere Modelle für die selbe Anwendung, jedoch muss der generierte Code durch Anwendungslogik, die bei anwendungsspezifischen Sicherheitseigenschaften

gewöhnlich auch sicherheitsrelevant ist, erweitert werden. Model Checking von großen sicherheitskritischen Anwendungen, in denen das Verhalten der Systemteilnehmer vollständig spezifiziert wurde kann immer zu einer Zustandsexplosion des Suchraums führen. Deshalb sind zusätzliche Modelle zum Reduzieren der Komplexität beim Model Checking notwendig. [5] beschreibt eine abstrahierte sicherheitskritische Anwendung, jedoch wurde die Anwendung händisch in der textuellen Spezifikation des Model-Checkers abstrahiert. In [41] wird ein Ansatz beschrieben, wodurch eine Untermenge von möglichen Testfällen, anhand Ressourcen- und Zeitrestriktionen ausgewählt werden kann. Dieser Ansatz betrachtet jedoch keine Sicherheit gegenüber einen Angreifer.

Integration externer Web Services

Mir ist keine Arbeit bekannt, die sich mit der sicheren Integration und Austauschbarkeit von existierenden Web Services in einem modellgetriebenen Ansatz im Kontext von sicherheitskritischen Anwendung befasst. Die am Ehesten verwandte Arbeit ist jedoch die von Pironti et al. [74]. In dieser beschreiben sie die modellgetriebene Entwicklung eines TLS-Clients, der verifiziert wurde und mit einem existierenden TLS-Service interagieren konnte. Hierbei wurde jedoch die Konvertierung der Nachrichten manuelle und ohne Sicherheitsgarantien hinzugefügt. Zudem wurde auch nicht die Austauschbarkeit sowie die sichere Kommunikation anhand WS-SecurityPolicies betrachtet.

WS-SecurityPolicies wurden jedoch von Nakamura et al. [67] betrachtet. Sie beschreiben Standardsicherheitseigenschaften mit Stereotypen, die verwendet werden um vordefinierte *Security-Patterns* aus eine Bibliothek auf dem Anwendungsmodell anzuwenden. Aus dem Modell werden Konfigurationsdateien für den IBM WebSphere Application Server (WAS) sowie WS-SecurityPolicy Spezifikationen generiert. Menzel et al. [59] beschreiben ebenfalls einen modellgetriebenen Ansatz, der abstrakte *Security-Patterns* verwendet und aus diesen XML-basierte Konfigurationsdateien generiert. Im Gegensatz dazu wird in SecureMDD ein abstraktes Model der WS-SecurityPolicy eines existierenden Web Services generiert und dieses für die formale Verifikation verwendet. Beide Ansätze befassen sich nicht mit der Integration von existierenden Web Services.

KAPITEL 5

Generierung einer lauffähigen Anwendung anhand eines Anwendungsmodells

Aus einem Anwendungsmodell wird automatisch lauffähiger Programmcode erzeugt. Hierzu wird das plattformunabhängige Modell zuerst auf die Einhaltung der Modellierungsrichtlinien überprüft und der Entwickler über vorhandene Modellierungsfehler benachrichtigt. Falls das Anwendungsmodell fehlerfrei ist, wird dieses für jeden Komponententyp (Service, Terminal und Smartcard) mittels der Modell-zu-Modell-Transformationssprache QVTo [70] in ein plattformspezifisches Modell transformiert. Hierbei werden unter anderem die plattformunabhängigen Datentypen und MEL-Ausdrücke in plattformspezifische transformiert sowie plattformspezifische Klassen hinzugefügt. Aus einem plattformspezifischen Modell wird anschließend für jede Komponente (Serviceklasse, Terminalklasse, Smartcardklasse) mittels der Modell-zu-Text-Transformationssprache XPand[1] lauffähiger Programmcode generiert. Der Ansatz verwendet das Eclipse Modeling Framework und ist als Eclipse-Plugin realisiert.

Dieses Kapitel beschreibt in Abschnitt 5.1 den generierten Servicecode und geht in Abschnitt 5.2 auf die Implementierung der externen Services ein. Abschnitt 5.3 beschreibt die beim Deployment einer generierten Serviceanwendung relevante Aspekte. Ein Teil des generierten Servicecodes ist in [17] und [16] publiziert.

5.1 Web Services

UML-Klassen mit dem Stereotyp ≪Service≫ werden im generierten Code durch Java Web Services realisiert, die zum strukturierten Nachrichtenaustausch das Netzwerkprotokoll SOAP [61] verwenden. Zur Implementierung wurde JAX-WS [77] verwendet. Dabei handelt es sich um eine Java API für XML Web Services, die im Web Service Stack Metro[2] enthalten ist. JAX-WS ist ein Standard und unterstützt sowohl Services als auch Clients und kann mit annotierten Java Objekten, die aus den SecureMDD-Transformationen entstehen verwendet werden.

[1] http://projects.eclipse.org/projects/modeling.m2t.xpand
[2] http://metro.java.net/

5.1.1 Stateful Web Services

Services werden als stateful Web Services realisiert. Dadurch erhält jeder Aufrufer eine eigene Serviceinstanz (Sitzungsinstanz), wodurch Sitzungsdaten in den Instanzattributen gespeichert werden können. Für die Realisierung von stateful Web Services mit JAX-WS werden für jede modellierte Serviceklasse zwei Java-Klassen generiert: ein StatefulManager und die modellierte Serviceklasse. Der StatefulManager ist ein Service mit genau einer Serviceoperation, die eine Sitzungsinstanz der modellierten Serviceklasse erstellt und anschließend eine Adresse zu der Sitzungsinstanz (Service-Endpunkt) zurück gibt. Anhand dieser Adresse kann der Client die für ihn erstellte Sitzungsinstanz aufrufen.

```
1  @WebService
2  public class BankStatefulManager {
3
4    private PublicKey verifyKey;
5    private AuthToken bankAuthToken;
6    ...
7
8    public BankStatefulManager(){...}
9
10   ...
11
12   @WebMethod
13   public synchronized W3CEndpointReference getReference()
14       throws java.lang.Exception {
15     Bank service = new Bank();
16
17     service.setVerifyKey(verifyKey);
18     service.setBankAuthToken(bankAuthToken);
19     ...
20
21     W3CEndpointReference reference = Bank.manager.export(
22         W3CEndpointReference.class, service);
23
24     return reference;
25   }
26 }
```

Listing 5.1: StatefulManager am Beispiel Onlinebanking

Listing 5.1 illustriert den StatefulManager für die Onlinebanking-Anwendung. Die generierte Serviceklasse wird mit *@WebService* annotiert (Zeile 1) und die Serviceoperation mit *@WebMethod* (Zeile 12). Dabei handelt es sich um Annotationen die von JAX-WS unterstützt werden. Der StatefulManager besitzt alle Klassenattribute der modellierten Serviceklasse, die den Stereotypen ≪Initialize≫ anwenden (Zeile 4-6), um eine Sitzungsinstanz mit diesen Werten zu initialisieren. Beim Aufrufen der Serviceoperation *getReference* wird eine neue Sitzungsinstanz der modellierten Serviceklasse erstellt (Zeile 15), diese mit den initialen Attribu-

ten befüllt (Zeile 17-19) und anschließend wird eine *W3CEndpointReference* anhand dieser Instanz erstellt und dem Aufrufer zurückgegeben (Zeile 21-22). Eine *W3CEndpointReference* wird ebenfalls von JAX-WS unterstützt und ermöglicht einen Fernzugriff auf die neue Instanz. Die Initialisierung des StatefulManagers geschieht über den Konstruktor.

```
1  @Stateful
2  @Addressing
3  @WebService
4  public class Bank extends AgentComm{
5
6    public static StatefulWebServiceManager<Bank> manager;
7
8    private PublicKey verifyKey;
9    private AuthToken bankAuthToken;
10   ...
11
12   private Account account;
13   private UniqueTransaction ut;
14   ...
15 }
```

Listing 5.2: Serviceklasse am Beispiel Onlinebanking

Listing 5.2 zeigt einen Ausschnitt des Codes, der für die modellierte Serviceklasse *Bank* generierte wird. Diese Klasse wird ebenfalls mit *@WebService* annotiert aber auch mit *@Stateful* und *@Addressing*. Durch *@Stateful* können mehrere Instanzen eines Service-Endpunkts verwaltetet werden, wodurch für jeden Aufrufer eine Sitzungsinstanz erzeugt werden kann und durch *@Addressing* kann ein solcher Service-Endpunkt anhand einer *W3CEndpointReference* aufgerufen werden. Zudem erhält die generierte Serviceklasse ein statisches Klassenattribut *manager* vom Typ *StatefulWebServiceManager* (Zeile 6). Dieses wird vom generierten StatefulManager verwendet um eine *W3CEndpointReference* für eine Instanz der Servicklasse zu erstellen (Zeile 21-22 in Listing 5.1). Da auf das *manager*-Attribut nur innerhalb der Serviceoperation *getReference* zugegriffen wird und diese das Schlüsselwort *synchronized* verwendet, wird ein gleichzeitiger Zugriff verhindert. Zudem besitzt die generierte Serviceklasse alle initialen Attribute (Zeile 8-10) und nicht initialen Attribute (Zeile 12-14) der modellierten Serviceklasse und der Service erbt von der abstrakten Klasse *AgentComm*, welche für die Kommunikation mit anderen Services zuständig ist.

5.1.2 Serviceoperationen

In Abschnitt 4.2.2 wurde gezeigt wie das dynamische Verhalten von Services modelliert wird. Dies kann entweder zusammen mit dem Serviceaufruf in einem Aktivitätsdiagramm stattfinden (siehe Abbildung 4.6) oder durch die Modellierung einer Serviceoperation (siehe Abbildung 4.7). Im ersten Fall muss während den

Transformationen analysiert werden, wo ein Serviceaufruf beginnt und wo er endet. Dies bedeutet, welche Nachricht einen Serviceaufruf darstellt und bei welcher Nachricht es sich um die Rückantwort zu diesem Serviceaufruf handelt. Diese Informationen ist implizit durch den Nachrichtenverlauf vorhanden. Im zweiten Fall ist dies explizit modelliert. Falls ein Service, während er aufgerufen wird, keine weiteren Services aufruft, dann besteht der modellierte Aufruf aus genau einem Protokollschritt und aus diesem wird eine Serviceoperation mit der Annotation *@WebMethod* generiert. Ein Beispiel hierfür ist der Protokollschritt *BuyTicket* aus der ETicket-Anwendung (siehe Abbildung 4.16 für das Modell und Listing 9.1 im Anhang für den generierten Code). Falls während eines Serviceaufrufs wiederum andere Services aufgerufen werden, dann wird ebenfalls eine Serviceoperation generiert, aber auch private Operationen für die anderen Protokollschritte, die während dem Serviceaufruf ausgeführt werden.

```
1  @WebMethod
2  public synchronized
3      TransferResult processTransfer(ProcessTransfer t)
4        throws ServiceException {
5    ...
6    if (accounts.containsKey(t.getAccountNumber())) {
7      ...
8      Message transactionResult = sendMsg(...);
9
10     TransferResult tr = processTransactionResult(
11                 (TransactionResult) transactionResult)
12
13     return tr;
14
15     ...
16   } else {
17     stop("authentication_error");
18     return null;
19   }
20   ...
21 }
```

Listing 5.3: Ausschnitt aus der generierten Serviceoperation zur internen Ausführung einer Überweisung

Listing 5.3 veranschaulicht die aus Abbildung 4.12 generierte Serviceoperation zur internen Ausführung einer Überweisung beim Onlinebanking. Hierbei handelt es sich um eine modellierte Serviceoperation. Der Methodenname ergibt sich aus dem Aktivitätsnamen (bei nicht modellierten Serviceoperationen aus dem Schlüsselwort *process* gefolgt von dem Typ der ersten Nachricht) und der Rückgabetyp ist der Typ der Rückantwort (*TransferResult*). Die empfangene Nachricht wird als Parameter übergeben und die Serviceoperation kann eine generierte *ServiceException* werfen, die an den Aufrufer als SOAP-Nachricht weitergeleitet wird. Zudem erhält die Serviceoperation das Schlüsselwort *synchronized*, damit bei einem potentiellen

Aufruf durch den Angreifer kein gleichzeitiger Zugriff auf die Serviceattribute stattfindet. Die hier betrachtete Serviceoperation besteht aus zwei Protokollschritten. Die Anweisungen des ersten Protokollschritts (inklusive der Nachrichtenübertragung) sind direkt in der Serviceoperation enthalten (Zeile 6-8). Der zweite Protokollschritt wird durch eine Operation (*processTransactionResult*) gekapselt, die nur intern aufgerufen werden kann. Als erstes wird überprüft ob die übergebene Kontonummer in der Key-Value-Liste *accounts* vorhanden ist (Zeile 6). Falls nicht, dann wird eine Exception mit dem String *authentication error* geworfen (Zeile 17). Der erste Protokollschritt gilt als beendet, wenn ein neuer Serviceaufruf stattfindet. Dies geschieht mit der Methode *sendMsg* (Zeile 8). Da es sich hierbei um eine allgemeine Methode für alle Nachrichtentypen handelt, muss das Ergebnis auf den richtigen Typ gecastet werden und kann anschließend an den nächsten Protokollschritt übergeben werden. Zum Schluss wird der Rückgabewert der aufgerufenen Operation zurückgegeben (Zeile 13). Die vollständige Code der Serviceoperation ist im Anhang in Listing 9.2.

5.1.3 Serviceaufruf incl. Verbindungsaufbau und -abbau

Die Kommunikation zu Services erfolgt anhand Stubs, die automatisch mit dem in JAX-WS enthaltenen Tool *wsimport* generiert werden. Somit können anhand der Stubs die Serviceoperationen als lokale Operationsaufrufe erfolgen. Die Stubs regeln somit den Aufruf der entfernten Methoden. Der Aufruf der Stubs sowie der Verbindungsaufbau und -abbau erfolgt durch die Methode *sendMsg*.

```
1  protected Message sendMsg(Message msg, int port,
2        String address, Boolean openSessionBeforeSend,
3        Boolean closeSessionAfterReceive)
4        throws ServiceException {
5
6      Message response = null;
7      Object receiver = getReceiver(address, port,
8                        openSessionBeforeSend);
9
10     switch (port) {
11       case Ports.Bank2Bank_default : {
12         try {
13           bankStubs.Bank s =(bankStubs.Bank) receiver;
14           setTimeout((BindingProvider) s);
15
16           switch (msg.getCode()) {
17             case Code.TRANSACTION : {
18               response = s.creditAccount((Transaction) msg);
19               break;
20             }
21             default : {
22               throw new ServiceException(...);
23             }
```

```
24            }
25
26            if (closeSessionAfterReceive) {
27              statefulServices.remove(Integer.toString(port));
28              s.closeService();
29            }
30
31          } catch (java.lang.Exception e) {
32            throw new ServiceException(...);
33          }
34          return response;
35        }
36        default : {
37          throw new ServiceException(...);
38        }
39      }
40    }
```

Listing 5.4: *sendMsg*-Methode des Bankservice in der Onlinebanking Fallstudie

Listing 5.4 zeigt die Methode *sendMsg* des Bankservices in der Onlinebanking Fallstudie. Es wird die zu übertragende Nachricht übergeben sowie der Port über den die Nachricht übertragen werden soll, die Adresse des Empfängers und die Information ob eine neue Sitzung vor dem Senden erstellt und nach dem Empfangen beendet werden soll. Ports stellen die Enden eines Kommunikationspfades zwischen zwei Knoten in einem Deploymentdiagramm dar und beschreiben somit über welchen Kommunikationskanal und zu welchen Empfängertyp eine Nachricht übertragen wird. Da die Adresse und der Port übergeben wird, muss die Adresse mit dem durch den Port definierten Empfängertyp übereinstimmen. Mithilfe der *getReceiver*-Methode wird der Stub erstellt (Zeile 7-8). Hierzu wird die Adresse, der Port und die Information ob eine neue Sitzung erstellt oder eine bestehende verwendet werden soll benötigt. Der in der *getReceiver*-Methode erstellte Stub wird in dem Klassenattribut *statefulServices* gespeichert, um bei einer weiteren Nachricht innerhalb der selben Sitzung auf die selbe Instanz zugreifen zu können. Ein Aufrufer muss jedoch in der Lagen sein Sitzungen zu verschiedenen Service zu verwalten, weshalb *statefulServices* als Map realisiert ist. Da die Methode *sendMsg* empfängerunabhängig ist, wird anschließend der Empfängertyp anhand des Ports unterschieden (Zeile 10) und das Empfängerobjekt *receiver* auf den richtigen Typ gecastet sowie ein Timeout für diesen gesetzt (Zeile 13-14). Falls der Cast fehlschlägt, da der Empfängertyp und somit auch die Adresse nicht zum Port passt, wird eine *ClassCastException* geworfen, aufgefangen und eine generierte *ServiceException* weitergereicht. Somit wird eine vom Angreifer manipulierte Serviceadresse zu einer anderen Serviceklasse erkannt. Nach einem erfolgreichen Cast wird anhand einem eindeutigen Nachrichtencode die richtige Serviceoperation des Stubs mit der zu übertragenden Nachricht aufgerufen (Zeile 16-24). Der Rückgabewert wird in der lokalen Variable *response* gespeichert. Falls die Sitzung geschlossen werden

soll, wird zusätzlich der Stub aus *statefulServices* gelöscht sowie die Serviceoperation *closeService*[3] aufgerufen (Zeile 26-29). Zum Schluss wird der Rückgabewert in *response* zurückgeben (Zeile 34). Falls die Serviceoperation *closeService* aufgrund des Angreifers nicht vollständig ausgeführt werden kann, wird die Verbindung zum Empfänger durch den Timeout geschlossen.

```
1  protected Object getReceiver(String address, int port,
2     Boolean openSessionBeforeSend) throws ServiceException {
3    try {
4     if (openSessionBeforeSend) {
5      if (getReceiverType(port).equals("Bank")) {
6       bankStatefulManagerStubs.BankStatefulManager manager =
7        new BankStatefulManagerService(
8         new java.net.URL(address),
9         new javax.xml.namespace.QName(
10          "http://bank.service.onlinebanking.generated/",
11          "BankStatefulManagerService")
12       ).getBankStatefulManagerPort();
13      setTimeout((BindingProvider) manager);
14      bankStubs.Bank statefulService =
15       new BankService().getPort(manager.getReference(),
16              bankStubs.Bank.class);
17      setTimeout((BindingProvider) statefulService);
18      statefulServices.put(Integer.toString(port),
19                statefulService);
20     }
21    }
22
23    return statefulServices.get(Integer.toString(port));
24
25   } catch (java.lang.Exception e) {
26    throw new ServiceException(...);
27   }
28   throw new ServiceException(...);
29  }
```

Listing 5.5: *getReceiver*-Methode des Bankservice in der Onlinebanking-Fallstudie

Listing 5.5 zeigt die Methode *getReceiver*. Falls eine neue Sitzung erstellt werden soll (Zeile 4) wird anhand des Ports die Empfängerklasse bestimmt (Zeile 5) und anschließend eine Sitzungsinstanz der richtigen Klasse erstellt. Hierzu wird ein Stub des StatefulManagers (*manager*) anhand der übergebenen Adresse und des qualifizierten Namens erstellt (Zeile 6-12), ein Timeout für diesen gesetzt (Zeile 13) und anschließend die Serviceoperation *getReference* des StatefulManagers aufgerufen (Zeile 15) und anhand der empfangenen *W3CEndpointReference* ein Stub für die neue Sitzungsinstanz erstellt (Zeile 14-16). Zum Schluss wird noch ein

[3]Die generierten Serviceklassen besitzt die Serviceoperation *closeService*, wodurch eine Sitzungsinstanz aus dem *manager*-Attribut gelöscht wird und somit nicht mehr aufgerufen werden kann.

Timeout für diesen Stub gesetzt (Zeile 17) und er wird in der *Map statefulServices* in Abhängigkeit des Ports gespeichert (Zeile 18-19). Somit kann bei einer weiteren Nachricht innerhalb der selben Sitzung anhand des Ports auf diese Instanz zugegriffen werden (Zeile 22). Falls der Sitzungsaufbau fehlschlägt oder versucht wird auf einen Stub zuzugreifen, der noch nicht erstellt worden ist, wird eine *ServiceException* geworfen, die bis zum Benutzer durchgereicht wird und somit der gesamte Protokollablauf beendet wird. Falls der Aufrufer nur genau eine Instanz einer Serviceklasse kennt (wie z.B. bei der ETicket-Anwendung), dann wird die Adresse des StatefulManagers nicht benötigt, da es nur eine gibt die im generierten Stub gespeichert ist.

5.1.4 Gemeinsame Attribute

Durch stateful Web Services erhält jeder Aufrufer eine Sitzungsinstanz in dessen Attributen sitzungsabhängige Daten (wie z.B. Nonces oder Sitzungsschlüssel) gespeichert werden können. Gemeinsam genutzte Daten, also Daten die von allen Sitzungsinstanzen eines Services verändert werden können, müssen speziell behandelt werden. Beispiele hierfür sind die Kontostände bei der Onlinebanking-Anwendung oder die ausgestellten Tickets bei der ETicket-Anwendung. Hierfür unterstützt SecureML den Stereotyp ≪static≫. Alle Attribute einer modellierten Serviceklasse, die diesen Stereotyp anwenden, werden in der generierten Serviceklasse als *static*-Variablen realisiert und können somit von alle Sitzungsinstanzen eines Services gelesen und verändert werden. Hierdurch ist aber auch ein gleichzeitiger Zugriff durch die Sitzungsinstanzen möglich. Deshalb muss der Zugriff auf solche statischen Variablen synchronisiert werden. Insbesondere die Bereiche in denen lesender und schreibender Zugriff stattfindet, müssen atomar ausgeführt werden. Dies wird durch die Synchronisation der Protokollschritte anhand des statischen *manager*-Attributes sichergestellt.

5.1.5 Transport Layer Security

TLS wird durch eine externe Java-Bibliothek realisiert und kann deaktiviert werden, falls eine andere TLS Implementierung verwendet werden soll. TLS verwendet Schlüssel und Zertifikate um eine sichere Verbindung aufzubauen. Diese werden in speziellen *key stores* und *trust stores* gespeichert. Die asymmetrischen Schlüsselpaare werden in *key stores* gespeichert und die signierten Zertifikate in den *trust stores*. Falls ein Zertifikat eine Zertifizierungsstelle darstellt, werden alle Zertifikate die von dieser Zertifizierungsstelle ausgestellt wurden ebenfalls akzeptiert. Es wird davon ausgegangen, dass die Public-Key-Infrastruktur nicht gebrochen ist und ein Angreifer sich somit keine gültigen Zertifikate für andere Serviceadressen ausstellen kann. Zudem wird bei TLS mit gegenseitiger Authentifizierung davon ausgegangen, dass der Angreifer die privaten Schlüssel der Clients nicht kennt und sich auch keine gültigen Zertifikate für seinen öffentlichen Schlüssel ausstellen kann.

5.1.6 Chipkartenkommunikation

Für die kryptografische Kommunikation mit Chipkarten konnte der in der Arbeit von Moebius [62] vorgestellte Codierungsmechanismus verwendet werden. Dieser übernimmt die Serialisierung und Deserialisierung zwischen Web Service-Klassen und Byte-Arrays, die an eine Java Card übertragen werden können. Ein Web Service, der z.B. eine verschlüsselte Nachricht an eine Chipkarte eines Kunden übertragen möchten, verwendet die genannte Serialisierung und überträgt anschließend das zurückgegebene Byte-Array in einer SOAP-Nachricht an den Computer des Kunden, der die Nachricht an dessen Chipkarte weiterleitet. Hierdurch wird sichergestellt, dass selbst wenn Malware auf dem Computer installiert ist, ein Angreifer die Klartextnachrichten nicht lesen kann.

5.1.7 Allgemeine Erweiterungen

Wiederholende Ereignisse

Wiederholende Ereignisse sind Serviceoperationen, die nicht von außerhalb aufgerufen werden, sondern durch einen zeitabhängigen Trigger. Beim Starten eines Services, der ein wiederholendes Ereignis besitzt, wird gleichzeitig ein neuer Thread gestartet, der die entsprechende Operation im modellierten Zeitintervall aufruft.

*-Listen und Key-Value-Listen

*-Listen und Key-Value-Listen werden durch die Java-Klasse ArrayList implementiert. Key-Value-Listen besitzen zusätzlich zu Listen die Methoden *get*, *set*, *removeKey* und *containsKey* (siehe Abschnitt 4.2.2). Die *get*-Methode iteriert solange durch die Liste bis ein Element gefunden wurde, dessen *key*-Attribut mit dem übergebenen Wert übereinstimmt. Anschließend wird eine Kopie erstellt und zurückgeben. Falls es kein solches Element gibt wird eine *ServiceException* geworfen. Bei der *removeKey*-Methode wird ebenfalls die Liste durchlaufen und wenn ein Element mit dem passenden *key* gefunden wurde, dann wird dieses gelöscht und die Methode beendet. Falls keins gefunden wurde wird nichts gemacht. Bei der *set*-Methode wird zuerst ein bereits in der Liste vorhandenes Element mit dem gleichen *key* durch die *removeKey*-Methode gelöscht und anschließend wird das übergebene Element der Liste hinten angehängt. Die *containsKey*-Methode wurde durch die *get*-Methode implementiert und gibt *true* zurück falls ein Element gefunden wurde. Falls kein Element gefunden wurde, wird die *ServiceException* abgefangen und *false* zurückgegeben.

Nachrichtenreihenfolge

Bei Smartcards kann eine Nachricht zu jeder Zeit empfangen und der entsprechende Protokollschritt ausgeführt werden, falls dies durch die modellierten Protokolle nicht verhindert wird. Serviceaufrufe bzw. modellierte Serviceoperationen können

aus mehreren Protokollschritten bestehen. Da diese jedoch im Code als Operation realisiert werden, wird hierdurch eine feste Nachrichtenreihenfolge innerhalb einer Operation sichergestellt. Zusätzlich dazu unterstützt SecureML mit der Property *fixMessageOrder*, dass alle Protokollschritte innerhalb einer Swimlane in einem Aktivitätsdiagramm die implizit vorhandene Reihenfolge sicherstellen (siehe Abschnitt 4.1). Hierzu wird vor der eigentlichen Transformation ins plattformspezifische Modell das plattformunabhängige Modell verändert indem dynamische Zustandsüberprüfungen in die Protokollschritte hinzugefügt werden und somit bei den darauffolgenden Transformationen mitberücksichtigt werden.

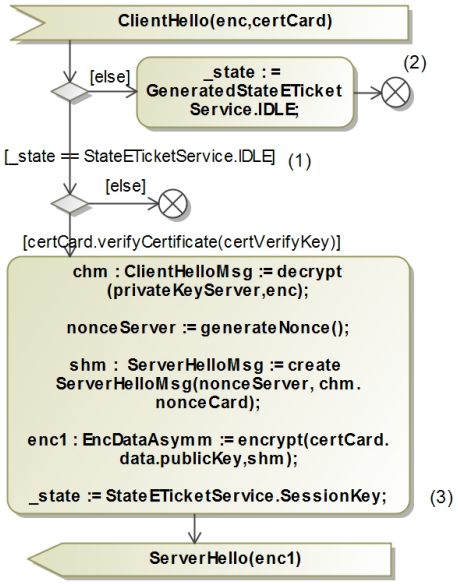

Abbildung 5.1: Hinzugenerierte Zustandsüberprüfung

Abbildung 5.1 zeigt den transformierten Protokollschritt *ClientHello* aus Abbildung 4.14. Direkt nach dem Empfangen der Nachricht wird überprüft ob sich der Empfänger im richtigen Zustand befindet (1). Falls dies nicht der Fall ist, wird der Zustand auf *IDLE* gesetzt (2). Am Ende des Protokollschritts wird der Zustand verändert, sodass nur noch der nächste Protokollschritt ausgeführt werden kann (3). Da ein Protokollabbruch durch einen modellierten *FlowFinalNode*, aber auch durch vordefinierte Operationen stattfinden kann, erhalten die Serviceoperationen im generierten Code zudem einen *try-catch*-Block. Hierdurch kann der hinzugenerierte Zustand bei einer *Exception* ebenfalls zurückgesetzt werden, aber auch eine modellierte Ausnahmebehandlungsroutine ausgeführt werden.

5.2 Kommunikation mit externen Web Services

Der grobe Ablauf zur Kommunikation mit externen Web Services wurde im Abschnitt 4.5.1 beschrieben. Hierzu muss einerseits ein WSDL-Dokument in eine UML-Serviceschnittstelle transformiert werden und andererseits müssen externe Serviceaufrufe generiert werden. Die Transformation von WSDL nach UML wurde durch das hyperModel[4] realisiert. Dieses verwendet generische XML-Schemas als Eingabe. XML-Schema ist eine Schemasprache zum Spezifizieren der Struktur und des Inhalts eines XML-Dokuments. Somit werden auch WSDL-Spezifikationen unterstützt. Da SecureML ein erweitertes UML-Profil mit eigenen Datentypen besitzt, musste dieses bei den Transformationen berücksichtigt werden. Zum Aufrufen der externen Services werden Stubs verwendet. Zum Generieren der Stubs wird WSDL2Java von Apache Axis2[5] verwendet sowie JiBX[6], um beliebige Klassenstrukturen auf XML-Dokumente abbilden zu können. Dies ist wichtig, da die Datentypen von externen Web Servcies von den vordefinierten Datentypen in SecureML abweichen.

5.3 Deployment

Wie ein Anwendung veröffentlicht werden muss ist hauptsächlich im Deploymentdiagramm spezifiziert. *User* repräsentiert echte Menschen, die über ein *Terminal* mit einem Service bzw. mit einer Smartcard kommunizieren dürfen. Es wird zwischen sicheren und unsicheren Terminals unterschieden. Dies beeinflusst nicht den generierten Programmcode sondern nur, welche Eigenschaften ein Gerät erfüllen muss auf dem der generierte Code veröffentlicht werden darf. In beiden Fällen ist eine installierte Java Virtual Maschine (JVM) notwendig. Bei einem unsicheren Terminal darf jedoch auch Malware installiert sein, ohne dass dadurch die Sicherheitseigenschaften verletzt werden. Ein Service muss auf einem Server veröffentlicht werden, zu dem nur der Betreiber des Services Zugriff hat. Für andere soll die Kommunikation mit einem Service nur über die im Modell definierte Schnittstelle möglich sein und nur über Verbindungen auf denen der Angreifer nicht mehr Fähigkeiten als die modellierten besitzt. Jede Komponente kann beliebig oft instanziiert werden. Bei Services kann dies jedoch durch das Schlüsselwort *singleton* eingeschränkt werden. Wie viele Instanzen eines Services einem Client bekannt sind, kann zusätzlich durch die Angabe von Multiplizität an den Enden der Kommunikationspfade eingeschränkt worden sein. Die Angreiferfähigkeiten stehen auf den Leitungen. Eingeschränkte Angreiferfähigkeiten müssen extern sichergestellt werden. Die Angreiferfähigkeiten *read, send, suppress* bedeuten, dass die Kommunikation über ein unsicheres Netzwerk wie z.B. das Internet stattfinden darf. Die Kommunikation mit einem Service kann zusätzlich durch TLS gesichert worden

[4] http://www.xmlmodeling.com/hypermodel/
[5] http://www.axis.apache.org/axis2
[6] http://www.jibx.sourceforge.net

sein. Hierzu wird automatisch Code generiert, der TLS anwendet. Die *key stores* und *trust stores* sind separate Dateien und sollten beim Veröffentlichen durch eigene ersetzt werden. Im Klassendiagramm können Klassenattribute mit dem Stereotyp *Initialize* annotiert werden. Diese müssen beim Veröffentlichen den Komponenten übergeben werden. Dies kann über die Konstruktoren der Service- und Terminalklassen erfolgen. Beim Service erfolgt dies über den StatefulManager. Es wird angenommen, dass der Betreiber den ununterbrochenen Betrieb des Services garantiert. Zudem besitzen die Key-Value-Listen, die zum Speichern von Kontoständen oder ausgestellten Tickets in den vorgestellten Anwendungen verwendet wurden eine prototypische Implementierung. Diese kann durch andere Implementierung oder Datenbanken, die das selbe Interface implementieren ersetzt werden. Der generierte Code enthält für jede Serviceklasse ein separates und von anderen Serviceklassen unabhängiges Package, wodurch ein Service einfach veröffentlicht werden kann.

Da sich die Anforderungen während der Softwareentwicklung oft ändern, ist es nützlich dass der Code für die modellierte Anwendung automatisch generiert und getestet werden kann. Zum Testen muss ein Testfall geschrieben werden, der alle Instanzen initialisiert, alle Services veröffentlicht und die Benutzernachrichten aufruft, um die Protokolle auszuführen. Wenn ein solcher Testfall existiert, wird der gesamte Prozess, angefangen mit der Codegenerierung, über das Veröffentlichen der Services, bis hin zur Ausführung des Testcodes durch einen Klick in Eclipse ausgeführt. Das Testframework veröffentlicht die Services auf einem leichtgewichtigen Server, der in Java integriert ist. Aber natürlich es es auch möglich die Services auf anderen Servern wie Glassfish[7] oder Tomcat[8] zu veröffentlichen.

5.4 Verwandte Arbeiten

Es existieren einige Codegeneratoren. Die meisten generieren jedoch anhand allgemeinen UML-Modellen nur Codefragmente wie Klassen und Methodensignaturen und werden an dieser Stelle nicht weiter betrachtet.

Es gibt jedoch auch Ansätze die auf Web Services zugeschnitten sind. SoaML[9] kann z.B. automatisch nach BPEL, WSDL oder Java transformiert werden. Für alle generierten Artefakte gilt jedoch, dass diese nur die Web Services Schnittstelle beinhalten und nicht das gesamte Verhalten des Web Services. MDD4SOA von Mayer et al. [56] ist eine Erweiterung von SoaML die auf Serviceorchestrierung zugeschnitten ist und generiert z.B. aus UML-Aktivitätsdiagrammen ebenfalls Code für Java, welcher jedoch nicht das vollständige Verhalten der Web Services enthält. Andere Ansätze wie [54] und [7] die sich mit Serviceorchestrierung befassen generieren ebenfalls nur die Web Service-Schnittstellen.

[7] http://www.glassfish.java.net/
[8] http://www.tomcat.apache.org/
[9] http://www.omg.org/spec/SoaML/

Die bisher erwähnten Ansätze haben zudem keinen Bezug zu sicherheitskritischen Anwendungen. Menzel [59] und Foster [33] betrachten zwar kryptographische Sicherheit jedoch generieren diese keinen lauffähigen Web Service-Code, sondern WS-Policies. Ein weiterer Ansatz der kryptographische Sicherheit betrachtet ist Spi2Java [75]. Die Codegenerierung erfolgt jedoch nicht anhand UML sondern anhand einer textuellen Spezifikationssprache genannt spi-calculus oder einer graphischen Visualisierung davon. Zudem wird lediglich Java-Code, jedoch kein Web Service-Code generiert.

Mir ist kein Codegenerator bekannt, der das vollständige Verhalten von Web Services generiert und kryptographische Operationen sowie die Kommunikation mit Chipkarten unterstützt.

KAPITEL 6

Finden von Sicherheitslücken mithilfe von Model Checking

Ein wichtiger Aspekt des SecureMDD-Ansatzes ist die Sicherheit der entwickelten Anwendungen. Welche Eigenschaften sichergestellt werden sollen und wie diese in SecureMDD definiert werden, wurde in Abschnitt 4.3 vorgestellt. Dieses Kapitel befasst sich mit Model Checking, um in den entwickelten Anwendungen Sicherheitslücken, welche diese Sicherheitseigenschaften verletzen, schon in der Designphase zu finden. Es existieren einige Model Checker (z.B. NuSMV[28], SPIN[43], PRISM[42]), jedoch nur wenige die kryptographische Protokolle unterstützen. AVANTSSAR[4] ist ein Tool zum automatischen Validieren der Sicherheit von serviceorientierten Architekturen. Als Eingabesprache dient hierbei ASLan++[80]. Diese Sprache unterstützt die Spezifikation von kryptographischen Protokollen sowie die Spezifikation von anwendungsspezifischen Sicherheitseigenschaften. Bei kryptographischen Protokollen werden Nachrichten über Kommunikationskanäle übertragen, welche von einem Angreifer beeinflusst werden. Die Spezifikation von einem solchen System, dessen Verhalten zudem vollständig spezifiziert werden muss, damit anwendungsspezifische Sicherheitseigenschaften betrachtet werden können, kann aufgrund von beschränkten Ressourcen sehr schnell zu groß und zu komplex für Model Checking werden. Deshalb ist es oft notwendig die Spezifikationen zu abstrahieren.

In Abschnitt 6.1 wird die Übersetzung von SecureML nach ASLan++ detailliert erläutert. Des Weiteren werden in Abschnitt 6.2 automatische Abstraktionen vorgestellt, um auch große Anwendungen Model-Checken zu können. Zudem werden die vorgestellten automatischen Abstraktionen, sowie die im Abschnitt 4.4 vorgestellten manuellen Abstraktionen anhand der ETicket-Fallstudie evaluiert. In Abschnitt 6.3 wird ein Angreifer-Cockpit vorgestellt, wodurch der Entwickler im Modellierungstool (MagicDraw) die Anwendung auf Sicherheitslücken überprüfen kann und ihm das Ergebnis in einer für ihn verständlichen Notation (SecureMDD-Notation) präsentiert wird. Zum Schluss werden in Abschnitt 6.4 die hier vorgestellten Ergebnisse mit verwandten Arbeiten verglichen. Die Übersetzung von SecureML nach ASLan++ sowie die Evaluierung des Model Checkers mit und

ohne Abstraktionen wurde in [18, 20] publiziert.

6.1 Übersetzung von SecureML nach ASLan++

Um ein SecureML-Anwendungsmodell mit AVANTSSAR auf Sicherheitslücken überprüfen zu können, muss dieses Modell zuerst nach ASLan++ übersetzt werden. Damit der Entwickler dies nicht für jede SecureML-Anwendung von Hand machen muss sind Transformationen sinnvoll, die eine solche Übersetzung automatisch durchführen. Während dieser Arbeit wurden Übersetzungsregeln von SecureML nach ASLan++ definiert und Transformationen für die automatische Übersetzung des Systemverhaltens nach ASLan++ erstellt. Mithilfe dieser Transformationen wurde für die ETicket-Anwendung eine entsprechende ASLan++-Spezifikation automatisch erstellt. Hierbei wurden die Komponenten auf jeweils eine Instanz eingeschränkt, weshalb Aspekte wie Mehrfachinstanziierung oder Sitzungsverwaltung nicht berücksichtigt wurden. Trotz dieser Einschränkungen war der Suchraum zu groß, weshalb selbst nach zwei Wochen keine Sicherheitslücken gefunden wurden, obwohl mehrere vorhanden waren. Deshalb wurden nicht alle Besonderheiten von SecureML betrachtet, sondern Mechanismen gesucht mit denen der Suchraum eingeschränkt werden kann, um in komplexen Anwendungen wie ETicket mit Model Checking Sicherheitslücken zu finden.

6.1.1 ASLan++-Grundlagen

ASLan++ ist eine textuelle Sprache, welche von AVANTSSAR zum Spezifizieren von sicherheitskritischen und serviceorientierten Architekturen verwendet wird. ASLan++ besitzt einige vordefinierte Elemente. Im folgenden wird auf einige Elemente eingegangen, welche wichtig sind um die Übersetzung nach ASLan++ nachvollziehen zu können.

Die Hauptbausteine in ASLan++ sind *entities*. Innerhalb einer *entity* können wiederum weitere *entities* aber auch *types*, *symbols*, ein *body* und *goals* definiert werden. Die äußerste *entity* wird gewöhnlich *Enviroment* genannt.

```
1  entity Environment
2  {
3      types
4          ...
5      symbols
6          ...
7      entity Session(...){
8          entity A(...){
9              ...
10         }
11         entity B(...){
12             ...
13         }
14         ...
15         body{
16             new A(...);
17             new B(...);
```

```
18              }
19          }
20          body{
21              ...
22              new Session(...);
23          }
24          goals
25              GoalName: ...
26  }
```

Listing 6.1: ASLan++ Environment

Listing 6.1 zeigt eine solche *Environment*. In dieser werden die Datentypen (*types*) definiert sowie die globalen Variablen und Funktionen (*symbols*) und alle Sicherheitseigenschaften (*goals*), welche Aussagen über das gesamte System machen. Eine solche *Environment* spezifiziert somit die Umgebung in der das System laufen soll. Innerhalb einer *Environment* wird gewöhnlich eine *Session* definiert, um Angriffe zu finden, welche mehrere Sitzungen benötigen. Eine solche *Session* wird als *entity* spezifiziert und enthält alle Kommunikationsteilnehmer (Zeile 8-10 und 11-13), die ebenfalls als *entities* spezifiziert sind. Diese werden auch Agenten genannt. Jede *entity* enthält einen *body*. Der *body* der *Environment* initialisiert das System, indem eine feste Anzahl an *Sessions* instanziiert wird (Zeile 22). Der *body* der *Session* instanziiert wiederum die Agenten (Zeile 15-18). Die Instanziierung einer *entity* erfolgt durch *new* gefolgt von dem Namen der *entity* und deren Parametern. In dem *body* eines Agenten wird dessen Verhalten spezifiziert. Um das Verhalten zu spezifizieren werden folgende vordefinierten Typen und Symbole verwendet (siehe Listing 6.2 und 6.3).

```
types
    fact
    message
    nat < message;
    text <    message;
    agent < message;
    symmetric_key < message;
    private_key < message;
    public_key < message;
```

Listing 6.2: Vordefinierte ASLan++-Typen

```
symbols
    i: agent
    iknows(message): fact
    true, false: fact;
    0,1,2,3, ... : nat;
    succ(nat): nat;
    noninvertible hash ( message ): message ;
    noninvertible scrypt ( symmetric_key , message ): message ;
    noninvertible crypt ( public_key , message ): message ;
    noninvertible sign ( private_key , message ): message ;
    inv ( public_key ): private_key ;
```

Listing 6.3: Vordefinierte ASLan++-Symbole

Einer der vordefinierten Typen ist *fact*. Atomare Propositionen werden durch Prädikate vom Typ *fact* spezifiziert. Ob eine atomare Proposition gilt oder nicht hängt von der Existenz des dazugehörigen Prädikatterms im globalen *fact space* ab. Ein weiterer Typ ist *message*. Alle Datentypen erben von *message*. Dies liegt daran, dass die Angreiferfähigkeiten auf dem Typ *message* definiert sind. Somit kann alles was übertragen wird vom Angreifer abhängig von seinen Fähigkeiten analysiert und manipuliert werden. Da sich Typen vererben lassen, kann somit zwischen verschiedenen Untertypen von *message* unterschieden werden. Hierzu zählen unter anderem *nat*, *text*, *agent*, *symmetric_key*, *private_key* und *public_key*. *nat* repräsentiert natürliche Zahlen. Das Bilden von natürlichen Zahlen geschieht über die Nachfolgerfunktion *succ(nat): nat*. Diese nimmt ein Symbol vom Typ *nat* entgegen und gibt den Nachfolger vom Typ *nat* wieder zurück. Zudem gibt es für jede natürliche Zahl ein vordefiniertes Symbol. Dieses Symbol wird auf einen Term aus verschachtelten *succ*-Operatoren abgebildet. Ein Beispiel hierfür ist, dass die Zahl 2 auf den Term *succ(succ(0))* abgebildet wird. Der Typ *text* repräsentiert hingegen Zeichenfolgen und *agent* beschreibt die Agenten. Ein Agent ist ein Systemteilnehmer welcher mit anderen Systemteilnehmern Nachrichten austauscht. Somit lassen sich die Identitäten der Agenten ebenfalls über die Kommunikationskanäle übertragen. Beim Angreifer handelt es sich um einen vordefinierten Agenten, welcher durch das Symbol *i* repräsentiert wird. Zusätzlich gibt es das Prädikat *iknows(message): fact*. Dadurch kann das Wissen des Angreifers erweitert und abgefragt werden. Zum Schluss gibt es noch *symmetric_key* für symmetrische Schlüssel, *private_key* für private Schlüssel und *public_key* für öffentliche Schlüssel. Um ein Schlüsselpaar zu spezifizieren, wird die Funktion *inv(public_key): private_key* verwendet. Diese gibt zu einem bestehenden öffentlichen Schlüssel einen privaten Schlüssel zurück. Konkret wird hierdurch nichts berechnet. Der private Schlüssel ist dann einfach der Term bestehend aus dem *inv*-Operator und dem öffentlichen Schlüssel. Diese Schlüssel können zum Verschlüsseln und Signieren verwendet werden. Hierzu gibt es die vordefinierten Funktionen *scrypt(symmetric_key, message): message*, *crypt(public_key, message): message* und *sign(private_key, message): message*. *scrypt* wird für die symmetrische Verschlüsselung verwendet, *crypt* für die asymetrische Verschlüsselung und *sign* zum Signieren. Alle drei Funktionen nehmen jeweils einen Schlüssel und eine Nachricht entgegen und geben eine Nachricht zurück. Da es sich hierbei um ideale Kryptographie handelt wird nicht tatsächlich verschlüsselt, sondern in dem Rückgabewert steht lediglich der Operator sowie der Schlüssel und die unverschlüsselte bzw. unsignierte Nachricht. Zum Entschlüsseln muss somit keine Operation aufgerufen werden, sondern es muss überprüft werden ob der Agent, welcher versucht die Nachricht zu entschlüsseln den passenden Schlüssel besitzt. Falls dies der Fall ist, kann der Agent auf die unverschlüsselte Nachricht zugreifen. Beim Verwenden der Funktion *hash(message): message* wird ebenfalls kein Hash erstellt, sondern ein Term mit dem Operator und den zu haschenden Daten. Der Angreifer kann nicht auf die *message* in einem Hashterm zugreifen und die generierten Agenten sind so spezifiziert, dass sie es nicht tun.

6.1.2 Allgemeine Übersetzungsregeln

Datentypen

Die vordefinierten SecureML-Datentypen aus Abschnitt 3.1 wurden nach ASLan++ übersetzt. Für einige Datentypen wurden neue ASLan++-Typen definiert und andere wurden in bestehende ASLan++-Typen übersetzt.

SecureML	**UML-Typ**	**ASLan++**	**ASLan++-Typ**
Number	primitive	number	nat
String	primitive	string	text
Boolean	primitive	boolean	nat
Secret	Class	secret	text
Nonce	Class	nonce	nat

Tabelle 6.1: Übersetzungstabelle von SecureML nach ASLan++

Tabelle 6.1 veranschaulicht die Übersetzung von SecureML-Datentypen auf neu erstellte ASLan++-Typen. Für die primitiven SecureML-Datentypen *Number*, *String* und *Boolean* wurden die äquivalente ASLan++-Typen *number*, *string* und *boolean* erstellt. Für die SecureML-Datentypen *Secret* und *Nonce*, welche als Klasse vordefiniert sind, wurden die ASLan++-Typen *secret* und *nonce* definiert. *number*, *boolean*[1] und *nonce* sind vom Typ *nat* und *string* und *secret* vom Typ *text*. Durch die Abbildung der Typen werden *false positives* vermieden. Ohne diese Typisierung könnten Angriffe in der ASLan++-Sepzifikation aufgrund einer fehlenden Typisierung gefunden werden, die jedoch im generierten Code aufgrund der verwendeten Typen nicht möglich sind.

Die restlichen vordefinierten SecureML-Datentypen werden auf bestehende ASLan++-Typen abgebildet. Hierbei handelt es sich um Datentypen, die für kryptographische Operationen notwendig sind.

SecureML	**ASLan++**
PublicKey	public_key
PrivateKey	private_key
SymmKey	symmetric_key
EncDataSymm	message
EncDataAsymm	message
SignedData	message
HashedData	message

Tabelle 6.2: Übersetzungstabelle von SecureML nach ASLan++

Die Tabelle 6.2 zeigt die Übersetzung der Datentypen, die für kryptographi-

[1] Da *true* und *false* nicht von Typ message erben und somit nicht übertragen werden können, erbt boolean von *nat* und erhält die Symbole t für *true* und f für *false*.

sche Operationen notwendig sind. Die öffentlichen, privaten und symmetrischen Schlüssel aus SecureML (*PublicKey, PrivateKey, SymmKey*) werden auf die äquivalenten ASLan++-Typen abgebildet (*public_key, private_key, symmetric_key*). Die symmetrisch und asymmetrisch verschlüsselten Daten (*EncDataSymm* und *EncDataAsymm*), sowie Signaturen und Hashes (*SignedData* und *HashedData*) werden auf den ASLan++-Typ *message* abgebildet. Dies ist notwendig, da die vorhandenen kryptographischen Operationen in ASLan++ für diese Typen definiert sind.

Komponenten

Jede Komponente in SecureML wird in eine ASLan++-*entity* übersetzt. Die Instanzen einer *entity* nennt man Agenten. Die lokalen Variablen sowie die Klassenattribute werden in *symbols* innerhalb der zuständigen *entity* transformiert. Primitive werden in ASLan++ als *type* definiert. Komplexe Klassenattribute werden in ihre Primitive zerlegt und in Variablen transformiert. Der *body* enthält den dynamischen Teil der Anwendung sowie die Kommunikationsstruktur und die Angreiferfähigkeiten auf den einzelnen Kommunikationskanälen.

Klassen

ASLan++ besitzt keine Klassen, weshalb Klassenattribute und deren Zugehörigkeit zu einer Klasse eindeutig abgebildet werden müssen. Deshalb wurde der ASLan++-Typ *class* spezifiziert. Zudem wird für jede Klasse mit Attributen eine gleichnamige Konstante vom Typ *class* definiert und alle primitiven Attribute einer Klasse werden als ASLan++-Variablen spezifiziert. Dazu gehören auch die primitiven Attribute, die wiederum in komplexen Klassenattributen enthalten sind. Um die Klassenstruktur der einzelnen Variablen abzubilden, wird diese in die Variablennamen codiert.

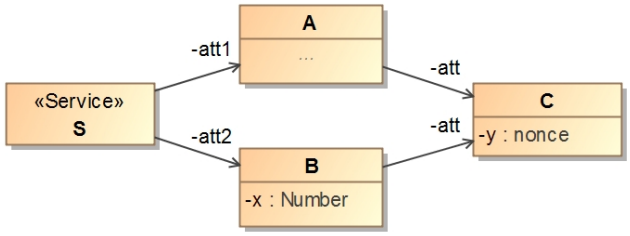

Abbildung 6.1: Klassen übersetzen nach ASLan++

Abbildung 6.1 zeigt einen Service S, welcher zwei komplexe Attribute *att1* und *att2* besitzt. *att1* ist vom Typ A und *att2* vom Typ B. Beide besitzen ein Klassenattribut *att* vom Typ C, mit dem Attribut y. Zudem besitzt *att2* ein Attribut x. Nun erhält der Service S in ASLan++ die folgenden Variablen:

- att1_att_y : nonce;
- att2_x : number;
- att2_att_y : nonce;

Zuweisungen von Primitiven werden in ASLan++ genauso wie in SecureMDD durch den Zuweisungsoperator := durchgeführt. Beim Zuweisen eines Klassenobjekts muss jedoch jede Variable dieser Klasse separat zugewiesen werden. Da es keine Klassen gibt, müssen auch keine Klassenobjekte erzeugt werden.

Senden und Empfangen

Das Verschicken und Empfangen von Nachrichten geschieht über Kommunikationskanäle. Diese sind in SecureML in einem Deploymentdiagramm durch Kommunikationspfade zwischen den Komponenten definiert. In ASLan++ werden diese implizit beim Senden und Empfangen definiert. *Actor -> B : M* spezifiziert das Versenden einer Nachricht *M* an den Agenten *B*. Will man nun das in Abbildung 6.1 betrachtete Klassenattribut *att2* verschicken, dann muss dieses aus den entsprechenden Variablen zusammengesetzt werden. Damit die Information über die Klassenstruktur erhalten bleibt und zwischen verschiedenen Klassen unterschieden werden kann, werden die Klassennamen als Präfixe hinzugefügt. Somit sieht das Versenden des Klassenattributes *att2* an den Agenten *B* wie folgt aus:

- Actor -> B : b.(att2_x.c.(att2_att_y))

Da ein *agent* vom Typ *message* erbt kann dieser ebenfalls verschickt werden. Falls wie bei der OAuth-Anwendung (siehe Abschnitt 9.1 im Anhang) Adressen über Leitungen verschickt werden (Adressen werden in SecureML durch den vordefinierten Datentyp *Address* modelliert), kann dies durch das Übertragen eines *agents* spezifiziert werden. Dieser kann anschließend in einer lokalen Variable gespeichert werden und beim Versenden einer Nachricht an diesen in dem oben gezeigten Konstrukt verwendet werden.

In SecureMDD können Nachrichten die z.B. Serviceoperation aufrufen beliebig oft und in einer beliebigen Reihenfolge empfangen werden. In ASLan++ muss dieses Verhalten im *body*-Element von jedem Agenten durch eine Endlosschleife und eine nichtdeterministische Wahl der aufzurufenden Operationen (zu empfangenden Nachrichten) spezifiziert werden (siehe Listing 6.4). Eine nichtdeterministische Wahl wird durch das *select*-Statement definiert, welches bedingte Verzweigungen ohne *else*-Fall enthält (*(on(...)*). *A → Actor : M1* bedeutet das ein Teilnehmer (Actor) die Nachricht *M1* vom Agenten *A* empfangen kann. Wie oft eine solche Nachricht während des Model Checkings empfangen werden kann, wird erst beim Starten des Model Checkers bestimmt.

```
while(true)
{
  select
```

```
  {
    on(A->Actor : M1):{...}
    on(A->Actor : M2):{...}
    ...
    on(B->Actor : Mn):{...}
} }
```

Listing 6.4: Empfangen von Nachrichten

Bedingte Verzweigung

In SecureML werden bedingte Verzweigungen durch *DecisionNodes* mit *Guards* an den ausgehenden Kontrollflüssen (*ControlFlows*) modelliert. In SecureML darf es nur zwei ausgehende Kontrollflüsse pro *DecisionNode* geben und einer der Kontrollflüsse muss der *else*-Fall sein. Deshalb lassen sich solche bedingten Verzweigungen in ASLan++ durch das *if/else* Konstrukt abbilden. Ein *Guard* kann ein beliebiger boolescher Ausdruck sein. Hierfür werden die logische Operationen *and*, *or* und *not*, sowie die Vergleichsfunktion == auf die entsprechenden ASLan++ Notationen (&, |, !, =) abgebildet.

Listen und Key-Value-Listen

SecureML unterstützt Listen und Key-Value-Listen, wobei ASLan++ nur Mengen unterstützt. Deshalb mussten SecureML-Listen durch ASLan++-Mengen nachgebildet werden. Jedes ASLan++-Listenelement wird durch ein Tupel definiert, welches aus dem originalen Listenelement und einem eindeutigen Index besteht. Zudem wird für jede Liste ein Indexzähler definiert. Somit kann *add* auf eine einfache und effiziente Weise nach ASLan++ übersetzt werden. Während der *add*-Operation wird der Indexzähler erhöht und durch die ASLan++-Operation *add* ein neues Tupel der Menge hinzugefügt. Dieses Tupel besteht aus dem originalen Element und dem neuen Zählerindex. Dadurch ist es möglich Duplikate in einer Menge zu speichern. Für die korrekte Abbildung der *remove*-Operation muss über den Indexzähler iteriert werden bis zum erste Vorkommen des Elements und anschließend kann dieses mit der ASLan++-Operation *remove* gelöscht werden.

Key-Value-Listen werden ebenfalls durch ASLan++-Mengen nachgebildet. Die *containsKey*-Operation kann auf das ASLan++-Prädikat *contains* abgebildet werden. Da *contains* nicht nur den Key erwartet, sondern ein Element der Menge, werden die nicht bekannten Attribute des Elements als frei wählbar definiert. Analog dazu kann die *removeKey*-Operation durch die ASLan++-Operation *remove* abgebildet werden. Da *remove* jedoch blockiert, falls ein solches Element nicht vorhanden ist, wird zusätzlich mit *contains* überprüft ob ein solches Element vorhanden ist und nur in diesem Fall *remove* ausgeführt. Um die *set*-Operation abzubilden, wird ein bestehendes Element mit dem selben Key zuerst gelöscht und anschließend das Element hinzugefügt. Die Überprüfung ob ein solches Element vorhanden ist sowie das Löschen des Elements wird durch *contains* und *remove* realisiert und das anschließende hinzufügen durch die ASLan++-Operation *add*. Die *get*-Operation

wird durch *contains* spezifiziert, indem die nicht bekannten Attribute des Objekts durch "?" gefolgt von einer lokalen Variable in diese Variablen gespeichert werden.

Angreifer

SecureML unterstützt für die Kommunikation mit Smartcards beliebige Angreiferfähigkeiten, die eine Untermenge von *(read, send, suppress)* sind. Für Services wurde dies auf *(read, send, suppress)* und die leere Menge eingeschränkt. Die Angreiferfähigkeiten werden auf Kommunikationspfaden definiert und beschreiben unabhängig vom modellierten Protokoll die Sicherheit der Kommunikationskanäle. In ASLan++ hingegen wird die Sicherheit der Kanäle durch Sicherheitseigenschaften wie ungesicherte, authentische oder vertrauliche Übertragung beschrieben. ASLan++ unterstützt ein *Ideal Channel Model (ICM)*, welches die Eigenschaften der Kanäle durch die Einschränkung der Angreiferfähigkeiten realisiert. Dies entspricht somit dem Konzept von SecureMDD.

read, send, suppress wird auf ungesicherte Übertragungen abgebildet. *read, suppress* wird auf eine authentische Übertragung mit Replay-Schutz abgebildet. Der ASLan++-Angreifer hat bei einer authentischen Übertragung trotzdem die Möglichkeit authentische Nachrichten mehrfach zu versenden bzw. deren Reihenfolge zu verändern. Der ASLan++-Replay-Schutz stellt sicher, dass die gleiche Nachricht nicht mehrfach akzeptiert wird. Um die Nachrichtenreihenfolge sicherzustellen muss zusätzlich ein Nachrichtenzähler spezifiziert werden. *send, suppress* wird auf eine vertrauliche Übertragung abgebildet, wodurch der Angreifer den übertragenen Inhalt nicht lesen kann. *suppress* sowie die leere Menge kann auf eine sichere Übertragung mit Replay-Schutz und Nachrichtenzähler abgebildet. Dadurch ist die Übertragung authentisch und vertraulich und es können keine Nachrichten wiederholt eingespielt werden bzw. deren Nachrichtenreihenfolge manipuliert werden.

Im Gegensatz zu SecureML werden in ASLan++ die Kommunikationskanäle durch die individuelle Übertragung von Nachrichten über einen Angreifer realisiert. Dies bedeutet, dass die Angreiferfähigkeiten für eine übertragene Nachricht beim Senden und beim Empfangen gleich sein müssen. Dies wird durch die Transformationen sichergestellt.

Kryptographie

In SecureMDD spielt Kryptographie eine besonders wichtige Rolle. Es werden symmetrische und asymmetrische Verschlüsselung, Signaturen, Zertifikate, Hashwerte, sowie Nonces unterstützt (siehe Abschnitt 3.2). Die symmetrische Verschlüsselung erfolgt in ASLan++ durch die Funktion *scrypt*. Diese erhält einen symmetrischen Schlüssel und die zu verschlüsselnde Nachricht. *scrypt* ist jedoch nur ein Symbol, wodurch die Nachricht nicht verschlüsselt wird. Es entsteht lediglich ein Term mit dem Symbol *scrypt* sowie dem Schlüssel und der Nachricht. Soll die Nachricht entschlüsselt werden, dann kann mithilfe des richtigen Schlüssels auf die Nach-

richt zugegriffen werden. Hierzu muss eine Gleichung aufgestellt werden, in der der Term gleich einem neuen Term ist, bei dem jedoch nur das Symbol *scrypt* und der Schlüssel bekannt sind. Der Rest (d.h. die Klartextdaten) wird so gewählt, dass die Gleichung stimmt. Deshalb kann die Entschlüsselung nur innerhalb einer *Guard*[2] beim Empfangen einer Nachricht oder bei einer bedingten Verzweigung stattfinden. Bei der asymmetrischen Verschlüsselung werden für das Verschlüsseln und Entschlüsseln verschiedene Schlüssel benötigt. Zum Verschlüsseln wird die Funktion *crypt* mit dem öffentlichen Schlüssel und der zu verschlüsselnden Nachricht aufgerufen. Zum Entschlüsseln wird jedoch wieder der öffentliche Schlüssel benötigt, da nur überprüft wird ob der Schlüssel der Selbe wie beim Verschlüsseln ist. Der Empfänger kennt natürlich seinen öffentlichen Schlüssel, jedoch wird in SecureML der private Schlüssel der Entschlüsslungsfunktion übergeben. Hierfür wurde das Prädikat *isKeyPair* definiert, welches die Zuordnung zwischen privatem und öffentlichem Schlüssel speichert. Mit diesem Prädikat kann innerhalb einer *Guard* der zu einem privaten Schlüssel gehörende öffentliche Schlüssel in einer lokalen Variable gespeichert werden und anschließend hierdurch auf die Nachricht zugegriffen werden. Signaturen und Zertifikate werden ähnlich behandelt. Hierfür wird ein Term erstellt, welcher das Symbol *sign*, einen privaten Schlüssel und die zu signierende Nachricht enthält. Zum Verifizieren wird somit wieder der private Schlüssel benötigt. Der Empfänger kennt jedoch nur den öffentlichen Schlüssel. Mithilfe der inversen-Funktion *inv* wird zu einem öffentlichen Schlüssel der private Schlüssel zurückgegeben. Da diese Funktion in SecureML nicht vorhanden ist, sondern nur an den geeigneten Stellen hinzutransformiert wird und dem Angreifer nicht bekannt ist, ist die Verwendung dieser Funktion legitim. Das Verifizieren kann wie das Entschlüsseln nur innerhalb einer *Guard* stattfinden. Der Unterschied zwischen einer Signatur und einem Zertifikat ist lediglich, dass ein Zertifikat zusätzlich zur Signatur noch die Daten, welche signiert wurden enthält. Ein Hashwert einer *message* wird durch die Funktion *hash* erstellt. Die Überprüfung geschieht ebenfalls in einer *Guard*, indem überprüft wird, ob die Nachricht mit der im Hashterm übereinstimmt. Nonces werden durch die Funktion *fresh* erstellt und sind vom Typ *message*. Diese können verschickt, gespeichert und mit anderen Nonces verglichen werden.

Initiale Eigenschaften und Sicherheitseigenschaften

Die Initialisierung erfolgt im *body* der Environment. Zum Initialisieren der vordefinierten Datentypen kann die Funktion *fresh* verwendet werden. Hierzu zählen *number*, *string*, *boolean*, *secret*, *nonce* und öffentliche Schlüssel. Private Schlüssel werden durch die inverse-Funktion *inv* erstellt und anschließend die Zuordnung von privaten Schlüssel auf öffentlichen Schlüssel durch das Prädikat *isKeyPair* hergestellt. Initiale Signaturen, Zertifikate, Hashes und verschlüsselte Daten werden

[2] Eine *Guard* ist ein Wächterausdruck der erfüllt sein muss, damit der durch ihn geschützte Bereich ausgeführt werden kann.

durch die dazugehörigen Funktionen *sign*, *hash*, *scrypt* und *crypt* erstellt. Ein initiales Angreiferwissen kann durch das Prädikat *iknows* spezifiziert werden. Da der Angreifer immer alle öffentlichen Schlüssel sowie alle Zertifikate kennt, kann dies automatisch erfolgen. Auch das Initialisieren der vordefinierten Datentypen mit *fresh* kann automatisch erfolgen. Anwendungsspezifische Initialisierungen werden in SecureML als initiale Eigenschaften mit OCL spezifiziert, die bei der Initialisierung in ASLan++ berücksichtigt werden können. Beispiele hierfür sind die Zuordnung der Schlüssel, die initialen Signaturen, Zertifikate und verschlüsselte Daten sowie anwendungsspezifisches initiales Angreiferwissen (siehe 6.1.3).

Sicherheitseigenschaften werden in SecureML ebenfalls in OCL spezifiziert. ASLan++ unterstützt die Spezifikation von prädikatenlogischen Aussagen, die immer gelten sollen. Hierbei werden folgende Operatoren unterstützt:

- [](f) (gilt immer)
- !f (Negation)
- f1 = f2 (Gleichheit)
- f1 & f2 (Konjunktion)
- f1 | f2 (Disjunktion)
- f1 => f2 (Implikation)
- forall V1 ... Vn. f (Allquantor)
- exists V1 ... Vn. f (Existenzquantor)

f steht hierbei für eine Formel. Eine Formel kann wieder aus Formeln bestehen und Terme miteinander vergleichen sowie Funktionen aufrufen.

Die initialen Eigenschaften und die Sicherheitseigenschaften werden derzeit jedoch nicht automatisch transformiert. Die initialen Eigenschaften und Sicherheitseigenschaften wurden für die ETicket-Fallstudie händisch übersetzt (siehe nächster Abschnitt).

6.1.3 Übersetzung anhand ETicket

Das hier betrachtete ETicket-Modell entspricht dem aus Abschnitt 4.2.4. Für die ETicket-Fallstudie wurde jede Komponente genau einmal instanziiert.

Abbildung 6.2 zeigt einen Teil des *ETicketService*-Verhaltens, welches im Kontext des Kaufprotokolls modelliert wurde. Das gesamte Protokoll zum Kaufen von Tickets ist in Abbildung 4.16 abgebildet. Nach dem Empfangen der Nachricht *BuyTicket* (3) wird zuerst überprüft, ob der Zustand *AUTHENTICATED* ist (5). Ist dies nicht der Fall, dann wird eine Exception-Aktivität, welche den Zustand auf *IDLE* setzt aufgerufen (siehe Abbildung 4.15 in Abschnitt 4.2.4) und der Protokollschritt beendet. Ist die Zustandsüberprüfung jedoch erfolgreich, dann werden

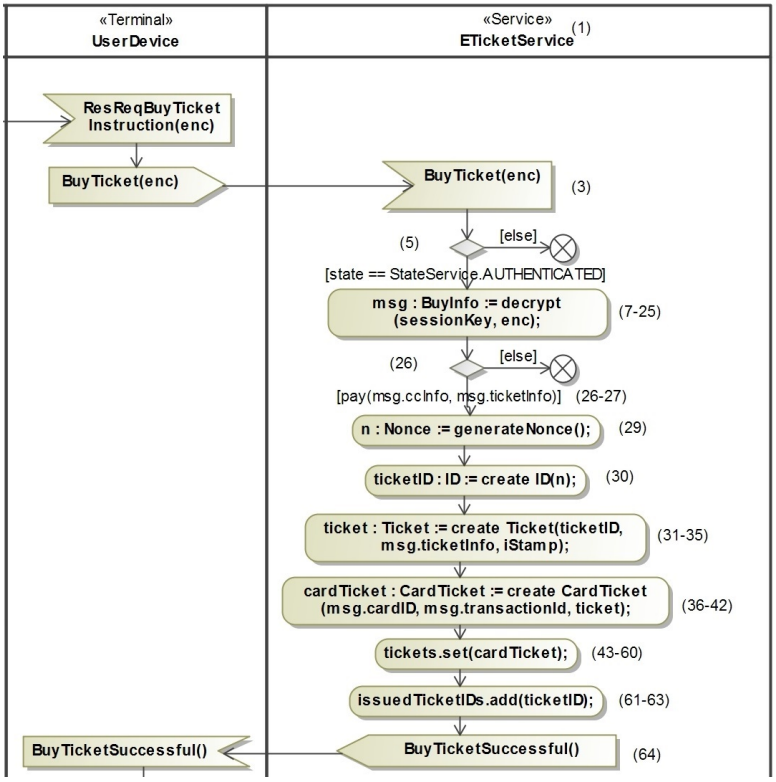

Abbildung 6.2: Ausschnitt aus dem ETicket Kaufprotokoll in SecureML

die verschlüsselten Daten in *enc* mit einem zuvor ausgetauschten symmetrischen Schlüssel entschlüsselt (7-25) und die Bezahlfunktion *pay* mit den zuvor entschlüsselten Kreditkartendaten (*ccInfo*) und Ticketinformationen (*ticketInfo*) aufgerufen (26-27). Bei einer erfolgreichen Bezahlung wird eine neue Nonce generiert (29), eine neue lokale Variable *ticketID* vom Typ *ID* erstellt und dieser die generierte Nonce zugewiesen (30). Anschließend wird eine lokale Variable *ticket* mit der zuvor erstellten *ticketID*, den entschlüsselten Ticketinformationen und einem vorinitialisierten Stempel (*iStamp*) erstellt (31-35). Dieses Ticket wird zusammen mit der übertragenen und entschlüsselten *cardID* und *transactionId* zum Erstellen eines *cardTickets* (36-42) verwendet. Dieses *cardTicket* wird zur Key-Value-Liste *tickets* hinzugefügt (43-60) und die *ticketID* wird der Liste *issuedTicketIDs* hinzugefügt (61-63). Zum Schluss wird die Nachricht *BuyTicketSuccessful* zurück an den Aufrufer geschickt (64).

```
1  entity ETicketService(...){
2  ...
3    on(UserDevice -> Actor : buyTicket.(?M_buyTicket_enc)) :
4    {
5      if (State = stateservice_authenticated)
6      {
7        if (M_buyTicket_enc = scrypt(SessionKey,
8          buyInfo.(
```

```
9           cCInfo.(
10            ?L_buyTicket_msg_ccInfo_cCHolderName.
11            ?L_buyTicket_msg_ccInfo_cCSecNumber.
12            ?L_buyTicket_msg_ccInfo_cCNumber.
13            ?L_buyTicket_msg_ccInfo_cCType.
14            ?L_buyTicket_msg_ccInfo_cCExpDate
15          ).
16          ticketInfo.(
17            ?L_buyTicket_msg_ticketInfo_departure.
18            ?L_buyTicket_msg_ticketInfo_destination
19          ).
20          iD.(
21            ?L_buyTicket_msg_cardID_id
22          ).
23          ?L_buyTicket_msg_transactionId)
24        )
25      ){
26        if (pay(L_buyTicket_msg_ccInfo_cCHolderName,...,
27                L_buyTicket_msg_ticketInfo_departure,...))
28        {
29          L_buyTicket_n := fresh();
30          L_buyTicket_ticketID_id := L_buyTicket_n;
31          L_buyTicket_ticket_id_id := L_buyTicket_ticketID_id;
32          L_buyTicket_ticket_ticketInfo_departure :=
33              L_buyTicket_msg_ticketInfo_departure;
34          ...
35          L_buyTicket_ticket_stamp_stamped := IStamp_stamped;
36          ...
37          L_buyTicket_cardTicket_id_id := L_buyTicket_msg_cardID_id;
38          L_buyTicket_cardTicket_transactionId :=
39              L_buyTicket_msg_transactionId;
40          L_buyTicket_cardTicket_ticket_id_id :=
41              L_buyTicket_ticket_id_id;
42          ...
43          if (contains( eTicketService_tickets,
44              (L_buyTicket_cardTicket_id_id, ?, ?, ?, ?, ?, ?)))
45          {
46            remove(eTicketService_tickets,
47                (L_buyTicket_cardTicket_id_id, ?, ?, ?, ?, ?, ?)
48            );
49          }
50          add(eTicketService_tickets,
51            (
52              L_buyTicket_cardTicket_id_id,
53              L_buyTicket_cardTicket_transactionId,
54              L_buyTicket_cardTicket_ticket_id_id,
55              L_buyTicket_cardTicket_ticket_ticketInfo_departure,
56              L_buyTicket_cardTicket_ticket_ticketInfo_destination,
57              L_buyTicket_cardTicket_ticket_stamp_stamped,
58              L_buyTicket_cardTicket_ticket_stamp_stampTime_dateTime
59            )
60          );
61          add(eTicketService_issuedTicketIDs),
62              (IssuedTicketIDsCounter, L_buyTicket_ticketID_id));
63          IssuedTicketIDsCounter := succ(IssuedTicketIDsCounter);
64          Actor -> UserDevice : buyTicketSuccessful;
65        }
66        else{
67          State := stateservice_idle;
68        }
69      }
70      else{
```

```
71          State := stateservice_idle;
72        }
73      }
74      else
75      {
76        State := stateservice_idle;
77      }
78    }
79  }
```

Listing 6.5: Ausschnitt aus dem ETicket Kaufprotokoll in ASLan++

Listing 6.5 zeigt die ASLan++-Spezifikation zu dem modellierten Verhalten. Der Systemteilnehmer *ETicketService* wird auf eine gleichnamige *entity* abgebildet (Zeile 1). Das Empfangen der Nachricht *BuyTicket* ist in (Zeile 3) spezifiziert. Der *Actor* ist in diesem Kontext der *ETicketService* und kann zu jedem Zeitpunkt vom *UserDevice* über einen unsicheren Kommunikationskanal die Nachricht *BuyTicket* erhalten. Konkret wird jede Nachricht akzeptiert, welche aus zwei konkatenierten *message*-Typen besteht, wobei es sich beim Ersten um die Konstante *buyTicket* handeln muss. Der *?*-Operator vor einer Variable bezeichnet, dass der empfangene Inhalt nicht mit der Variable verglichen wird, sondern in die Variable geschrieben wird. Da es sich um eine ungesicherte Übertragung handelt, kann auch der Angreifer eigene Nachrichten senden, welche akzeptiert werden. Nach dem Empfangen der Nachricht wird überprüft ob der Zustand *authenticated* ist (Zeile 5). Ist dies nicht der Fall, dann wird in Zeile 74-77 der Zustand durch die modellierte Exception-Aktivität auf *idle* gesetzt und die Nachrichtenbearbeitung ist abgeschlossen. Ansonsten wird die Abarbeitung des Protokollschritts fortgesetzt und die Nachricht entschlüsselt. Dies geschieht durch eine Gleichung, welche anhand der verschlüsselten Daten und dem richtigen Schlüssel die Unbekannten (den Klartext) bestimmt (Zeile 7-25). Da es keine Klassen gibt, werden direkt die Primitiven in die lokalen Variablen gespeichert. Dies geschieht indem die bekannte Klassenstruktur und für jedes primitive Attribut eine Variable angegeben wird. Falls der Schlüssel, die Klassenstruktur, die primitiven Datentypen oder die Anzahl der Datentypen nicht stimmt, dann schlägt die Entschlüsselung fehl und der Zustand wird durch die modellierte Exception-Aktivität wieder auf *idle* gesetzt (Zeile 70-72). Ansonsten war die Entschlüsselung erfolgreich und die Bezahlfunktion wird ausgeführt (Zeile 26-27). Die Bezahlfunktion *pay* gibt einen Wahrheitswert zurück und kann somit als *fact* (Prädikat) spezifiziert werden. In der ETicket-Fallstudie wurde für das Model-Checking das Verhalten der Bezahlfunktion vereinfacht indem der Defaultrückgabewert *true* im Modell definiert wurde. Da Model-Checking nur zum Finden von Fehlern verwendet wird und die Sicherheitseigenschaften unabhängig von einem konkreten Verhalten der Bezahlfunktion definiert sind, ist diese Vereinfachung legitim. Das bedeutet, dass durch diese Vereinfachung keine neuen Angriffe hinzu kommen. Dennoch könnte es Angriffe geben, welche nur gefunden werden können, wenn der Rückgabewert der Funktion *false* ist. Dies muss bei der Vergabe von Defaultwerten bei der Modellierung berücksichtigt werden. Falls die Bezahlung mitberücksichtigt werden soll, dann muss im SecureMDD-Modell das

Verhalten modelliert oder eine OCL-Constraint für diese Operation definiert werden. OCL-Constraints könnten auf Horn-Klauseln in ASLan++ abgebildet werden. In dem Fall, dass die Bezahlfunktion erfolgreich ist, wird eine neue Nonce durch die Operation *fresh* generiert (Zeile 29) und mit dieser Nonce die lokale Variable *ticketID* erstellt (30). Das Erstellen der Variable *ticket* (Zeile 31-35) und *cardTicket* (Zeile 36-42) geschieht durch die Zuweisung aller Primitive dieser Klasse. Da dies die Spezifikation sehr groß macht, wurde die gezeigte Spezifikation gekürzt. Nach dem Erstellen der zwei Variablen wird die zweite Variable (*CardTicket*) einer Key-Value-Liste hinzugefügt (Zeile 43-60). Key-Value-Listen werden durch Mengen realisiert. Damit ein bestehendes Element mit dem selben *key* aktualisiert wird und nicht einfach als neues Element hinzugefügt wird muss das alte Element gelöscht werden. Deshalb wird mittels dem *contains*-Prädikat überprüft, ob ein Element mit dem selben *key* (*L_ buyTicket_ cardTicket_ id_ id*) vorhanden ist (Zeile 43-44) und gegebenenfalls mit der *remove*-Operation gelöscht (Zeile 46-47). Der *?*-Operator steht hierbei als Platzhalter, da das Element *cardTicket* aus sieben Primitiven besteht. Anschließend wird das Element *cardTicket* zur Key-Value-Liste *tickets* hinzugefügt (Zeile 50-60). Auch hier müssen wieder alle Primitive übergeben werden. Beim Hinzufügen der *ticketID* zu der Liste *issuedTicketIDs* wird das Element der Liste mit der *add*-Operation für Mengen hinzugefügt (Zeile 61-62). Ein solches Listenelement besteht jedoch aus einem Zähler und dem eigentlichen Element. Der Zähler ermöglicht das auch Duplikate gespeichert werden können. Nach dem Hinzufügen des Elements zu der Liste wird noch der Zähler erhöht (Zeile 63). Das Versenden der Rückantwort an *UserDevice* geschieht in Zeile 64. Da in SecureML bei der Rückantwort immer der selbe Kommunikationskanal wie beim Aufrufen verwendet wird, wird ebenfalls eine unsichere Übertragung verwendet. Somit kann der Angreifer auch hier die Nachrichten empfangen, manipulieren und unterdrücken.

Initiale Eigenschaften und Sicherheitseigenschaften

Für die ETicket-Fallstudie wurden die initialen Eigenschaften sowie die Sicherheitseigenschaften nach ASLan++ übersetzt. Die Zuordnung der Schlüssel, die initialen Signaturen, Zertifikate und verschlüsselte Daten sowie das initiale Angreiferwissen kann in SecureML durch initiale Eigenschaften spezifiziert und anschließend nach ASLan++ übersetzt werden.

- *ETicketCard.allInstances().id->forAll(id|*
  Attacker.getInstance().knowledge->incudes(id))

Diese initiale Eigenschaft spezifiziert, dass der Angreifer initial die *ids* alle Karten kennt. Dies wurde nach *iknows(ETicketCard_ id)* übersetzt. In SecureML kann eine Komponente standardmäßig beliebig aber endlich oft instanziiert werden. Der Zugriff auf alle Instanzen geschieht durch folgenden Ausdruck:

- *'Komponentenname'.allInstances()*

Da in diesem Beispiel jedoch nur jeweils eine Instanz pro Komponente betrachtet wird, kann hiervon abstrahiert werden. Bei mehreren Instanzen könnte die Eigenschaft dupliziert und die Variable angepasst werden.

Die nächste initiale Eigenschafte definiert, dass der öffentlicher Schlüssel *c.certCard. data.publicKey* und der private Schlüssel *c.privateKeyCard* ein Schlüsselpaar sind.

- *ETicketCard.allInstances()->forAll(c| isKeyPair(c.certCard.data.publicKey, c.privateKeyCard)*

Diese Eigenschaft wurde nach *isKeyPair(ETicketCard_certCard_data_publicKey, ETicketCard_privateKeyCard)* übersetzt und bei der Initialisierung aufgerufen.

Listing 6.6 zeigt die Sicherheitseigenschaft, die aussagt, dass nur ausgestellte Tickets gestempelt werden. Listing 6.7 zeigt die Übersetzung nach ASLan++.

```
inv:
InspectorDevice.allInstances().stampedTicketIDs->
forAll(id | ETicketServer.allInstances().
  issuedTicketIDs->includes(id)
)
```

Listing 6.6: "Nur ausgestellte Tickets werden gestempelt" in SecureML

```
inv: [] ( forall ID .
      contains(inspectorDevice_stampedTicketIDs, (?,ID))
        => contains(eTicketService_issuedTicketIDs, (?,ID))
    );
```

Listing 6.7: "Nur ausgestellte Tickets werden gestempelt" in ASLan++

Der []-Operator spezifiziert, dass es sich hierbei um eine Invariante handelt. Die folgende Aussage soll somit während der gesamten Laufzeit des Systems gelten. Mit *InspectorDevice.allInstances().stampedTicketIDs* werden alle *stampedTicketIDs* von allen *InspectorDevices* aufgesammelt. Der Zugriff auf die Liste *stampedTicketIDs* vom *InspectorDevice* geschieht über die Variable *inspectorDevice_stampedTicketIDs*. Da die Liste neben den eigentlichen Elementen zusätzlich jeweils einen Zähler enthält, wird durch *contains(inspectorDevice_stampedTicketIDs, (?,ID))* das gewünschte Element selektiert. Der Zähler wird durch *?* ignoriert, da er auch nicht modelliert wurde. In ASLan++ wird *forAll* nicht als Listenoperation unterstützt. Somit wird die *forAll*-Operation auf Listen wie folgt übersetzt:

- *l->forAll(e| function(e)* nach *forall e. contains(l,e) => function(e)*

Falls die Elemente der Liste *l* mehrere modellierte Attribute besitzen, dann müssen alle Attribute als quantifizierte Variablen vorkommen. Hierduch können auch Attribute für alle Elemente einer Liste aufgesammelt werden und die *forAll*-Operation nur auf diesen Attributen ausgeführt werden. Die *includes*-Operation im OCL-Ausdruck kann auf die *contains*-Operation in ASLan++ abgebildet werden.

6.2 Abstraktionen

Model Checking von großen Anwendungen benötigt viele Ressourcen. Die in Abschnitt 4.2.4 vorgestellte ETicket-Fallstudie besitzt 52 unterschiedliche Protokollschritte, die wiederum zu mehreren Transitionregeln transformiert werden. Eine automatische 1:1-Übersetzung ergab eine Spezifikation mit 162 Transitionsregeln. Die gängigen AVANTSSAR-Beispiele besitzen jedoch durchschnittlich nur 20 Transitionsregeln. Im Vergleich hierzu sind 162 Transitionsregeln ein enormer Unterschied, da die Komplexität exponentiell mit der Anzahl der Transitionsregeln wächst. Dies führte dazu, dass die ETicket-Fallstudie selbst nach zwei Wochen nicht vollständig überprüft werden konnte. Die Laufzeit für das Model Checking ist somit stark abhängig von der Anzahl der Transitionsregeln, aber auch von deren Abhängigkeiten zueinander. Die Transitionsregeln werden durch die Anzahl der modellierten Systemteilnehmer sowie deren Verhalten beschrieben. Des Weiteren wird die Laufzeit auch von den Angreiferfähigkeiten beeinflusst. Bei jeder Transition wird überprüft, ob der Angreifer durch das Erstellen und Einschleusen von Nachrichten die Sicherheitseigenschaften verletzen kann. Ein ebenfalls wichtiger Aspekt ist das Vorkommen von Schleifen und die Instanziierbarkeit der Systemteilnehmer. Abhängig von der möglichen Ausführungshäufigkeit einer Transitionsregel sowie der maximalen Pfadlänge, kann hierdurch die Ausführungszeit erheblich beeinflusst werden. Ein Pfad besteht hierbei aus mehreren Transitionsregeln und beschreibt eine mögliche Ausführungsreihenfolge. Somit ist die Ausführungszeit beim Model Checking abhängig von vielen Faktoren. Eine gängige Praxis um die Komplexität beim Model Checking zu reduzieren ist das Abstrahieren der Modelle [5]. In diesem Abschnitt werden einerseits automatische und andererseits manuelle Abstraktionen betrachtet. Bei den automatischen Abstraktionen handelt es sich um semantisch korrekte Ersetzungen, wodurch keine Sicherheitslücken verschwinden und keine neuen hinzukommen. Im Gegensatz dazu werden die manuellen Abstraktionen vom Anwendungsentwickler modelliert, wodurch das Verhalten und die damit verbundene Komplexität stark reduziert wird, aber einige Angriffe unter Umständen nicht gefunden werden.

6.2.1 Automatische Abstraktionen

Im Folgenden werden automatische Abstraktionen vorgestellt, wodurch die generierte Spezifikation von ETicket von ursprünglich 162 auf 65 Transitionsregeln reduziert werden konnte. Die Anzahl der Transitionsregeln der so entstandenen Spezifikation kann ohne das Weglassen von Anwendungsfunktionalität nicht erheblich mehr verringert werden. Die hierbei verwendeten Abstraktionen lassen sich in die vier folgenden Bereiche unterteilen.

1. Entfernen von Teilnehmern, die Nachrichten nur weiterleiten

2. Zusammenfassen von bedingten Verzweigungen

3. Statische Nachrichtenreihenfolge

4. Entfernen von unbenutzten Attributen

Das Entfernen von Teilnehmern, die Nachrichten nur weiterleiten kommt besonders bei Anwendungen vor, bei denen Chipkarten und Web Services miteinander kommunizieren. Hierbei wird ein Terminal als Intermediär verwendet. Ein solches Terminal, welches nur Nachrichten zwischen zwei anderen Systemteilnehmer weiterleitet, kann in der abstrakten Spezifikation entfernt werden. Um die Kommunikationsmöglichkeiten nicht zu verändern müssen neue Kommunikationspfade erstellt werden. Die Angreiferfähigkeiten auf einem neuen Kommunikationspfad entstehen durch die Vereinigung der Angreiferfähigkeiten auf den Kommunikationspfaden, welche durch den neuen Kommunikationspfad ersetzt wurden. In der ETicket-Fallstudie kann der Systemteilnehmer *UserDevice* entfernt werden. Hierzu muss jeweils ein neuer Kommunikationspfad zwischen *User* und *ETicketService*, zwischen *User* und *ETicketCard* und zwischen *ETicketCard* und *ETicketService* erstellt werden. Die Angreiferfähigkeiten auf den jeweiligen Kommunikationspfaden sind *read, send, suppress*. Das Anwenden dieser Abstraktion hat die Anzahl der Transitionsregeln für die ETicket-Fallstudie um 10% reduziert.

Ein weiterer Punkt zum Reduzieren der Transitionsregeln ist das Zusammenfassen von bedingten Verzweigungen. In SecureML werden solche Verzweigungen in Protokollschritten modelliert. Ein Protokollschritt wird als atomarer Bereich zwischen dem Empfangen einer Nachrichten und dem Senden der nächsten Nachricht betrachtet. Wenn alle Protokollschritte in ASLan++ als atomar spezifiziert werden könnten, dann könnte jeder Protokollschritt in genau eine Transitionsregel übersetzt werden. Jedoch werden in ASLan++ Protokollschritte, welche Verzweigungen enthalten, bei der Übersetzung nach ASLan in mehrere Transitionsregeln übersetzt, wodurch die Komplexität der Spezifikation steigt. Es ist jedoch möglich mehrere verschachtelte bedingte Verzweigungen in eine zusammenzufassen, wenn diese den selben *else*-Fall haben und nur die innerste Verzweigung auszuführende Anweisungen enthält. In diesem Fall können die Bedingungen der Verzweigungen mit einem logischen UND verknüpft werden. Dieser Fall kann in SecureMDD-Modellen ziemlich oft vorkommen, wodurch viele Transitionsregeln eingespart werden können. Des weiteren kann eine bedingte Verzweigung, welche direkt nach dem Empfangen einer Nachricht vorkommt in manchen Fällen schon beim Empfangen der Nachricht überprüft werden. Dies ist bei der dynamischen Zustandsüberprüfung der Fall. Diese wird verwendet, um das Ausführen von Protokollschritten nur in einem bestimmten Zustand zu erlauben. Beim Empfangen einer Nachricht können innerhalb der *on(...)* Anweisung Bedingungen überprüft werden, wann eine Nachricht empfangen werden kann, jedoch existiert hierbei kein *else*-Fall. Da üblicherweise in SecureML der *else*-Fall für alle Zustandsüberprüfungen gleich ist (z.B. wird nur der Zustand zurückgesetzt), ist es möglich das gewünschte Verhalten durch das Hinzufügen einer *on(true)*-Anweisung, welche den Zustand zurücksetzt zu spezifizieren. Mithilfe dieser Abstraktionen konnte die Anzahl der Transitions-

regeln bei der ETicket-Fallstudie um 50% verringert werden.

Die nächste Abstraktion ist die statische Spezifikation der Nachrichtenreihenfolge. Das bedeutet, dass die dynamische Zustandsüberprüfung während dem Empfangen einer Nachricht in kaskadierende ASLan++ Empfangsblöcke übersetzt wird. Dabei kann der innere Empfangsblock nur eine Nachricht empfangen, wenn der äußere eine Nachricht empfangen und bearbeitet hat. Die statische Nachrichtenreihenfolge ist im verwendeten Model Checker Cl-AtSe sehr effizient implementiert und führt zu einer enormen Beschleunigung, welche es ermöglicht hat überhaupt eine Sicherheitslücke in der ETicket-Fallstudie zu finden.

Die letzte Abstraktion entfernt unbenutzte Attribute. Einige Klassenattribute sind für die lauffähige Anwendung notwendig, jedoch wird in den Protokollen niemals auf diese zugegriffen. Ein Beispiel hierfür sind die Attribute von *TicketInfo (departure, destination, expiration, ...)*. Diese sind relevant für den tatsächlichen Benutzer und die Schaffner, jedoch nicht für die formale Spezifikation. Wenn die Attribute einer Klasse in den Protokollen niemals verwendet werden (insbesondere kein Konstruktoraufruf der Klasse) und sich die Sicherheitseigenschaften auf diese Klassenattribute nicht beziehen, dann müssen diese Attribute nicht in der ASLan++ Spezifikation spezifiziert werden. Für eine solche Klasse mit unbenutzten Attributen wird somit ein ASLan++ Datentyp spezifiziert. Das Model Checking einer ETicket-Version ist mit dieser Abstraktion fünf mal schneller als ohne.

Die vorgestellten Abstraktionen werden automatisch während den Transformationen angewendet und führen zu einer signifikanten Reduzierung der Systemkomplexität und machen das Model Checking von größeren Anwendungen wie der ETicket-Fallstudie mit ASLan++ und einem modelgetriebenen Ansatz erst möglich. Jedoch wächst die Ausführungszeit beim Model Checking exponentiell mit der Anzahl der Transitionregeln. Deswegen sind noch größere Anwendungen, wie die elektronische Gesundheitskarte [63], welche 105 verschiedene Protokollschritte besitzt (die in ungefähr 130 Transitionsregeln übersetzt werden) zu groß für das Model Checking von Sicherheitseigenschaften für die gesamte Anwendung.

6.2.2 Evaluation automatischer Abstraktionen

AVANTSSAR integriert drei Model Checker (Cl-AtSe, OFMC und SATMC). Alle drei benutzen als Eingabesprache ASLan++. Da nur Cl-AtSe die volle Syntax von ASLan++ unterstützt und da in Vergleichstests alle Model Checker bezüglicher der Geschwindigkeit vergleichbar schnell sind [79] wurde für die Evaluation nur der Model Checker Cl-AtSe verwendet. Cl-AtSe ist ein "Constraint Logic based Attack Searcher" für Sicherheitsprotokolle. Um Angriffe zu finden werden *rewriting* und *constraint solving*-Techniken sowie verschiedene Arten von *backward*-Strategien verwendet. Cl-AtSe unterstützt eine *split* Funktion um eine Spezifikation in viele Teilaufgaben zu zerlegen, welche gleichzeitig ausgeführt werden können. Die Tests wurden auf einem 3GHz quad core Computer ausgeführt. Ohne die *split*-Funktion lag die CPU-Auslastung bei ca. 13% und mit der *split*-Funktion konnte eine vollständige Auslastung erreicht werden. Für die ETicket-Fallstudie wurden

in Abschnitt 4.3 drei Sicherheitseigenschaften definiert und die Übersetzung nach ASLan++ im Abschnitt 6.1.3 skizziert. Die Evaluation wurde anhand der ETicket-Fallstudie sowie diesen drei Sicherheitseigenschaften durchgeführt. Hierbei wurde getestet welche Sicherheitslücken gefunden werden, welche Annahmen hierzu notwendig sind und wie lange es dauert um eine Sicherheitslücke zu finden.

Die erste Sicherheitseigenschaft lautet, dass **nur ausgestellte Tickets gestempelt werden**. Um diese Eigenschaft sicherzustellen ist es notwendig, dass nur Tickets, die auf einer gültigen Karte gespeichert sind, gestempelt werden. Deshalb muss sich die Karte gegenüber dem Schaffnergerät authentifizieren bevor das Ticket, welches sich auf der Karte befindet, gestempelt werden kann. Dies wird mithilfe von Zertifikaten gelöst. Anschließend wird sichergestellt, dass alle eingehenden Nachrichten von dieser authentifizierten Karte stammen. Dies wird durch Nonces sichergestellt. Das Schaffnergerät schickt hierzu an die Karte eine Nonce verschlüsselt mit dem zertifizierten öffentlichen Schlüssel der Karte. Diese antwortet mit der empfangenen Nonce verschlüsselt mit dem öffentlichen Schlüssel des Schaffnergereätes. Das konkrete Protokoll kann im Abschnitt 4.2.4 nachgelesen werden. Falls keine Nonce verwendet wird, dann kann ein Angreifer eine Antwortnachricht einschleusen und ein Ticket, welches nicht auf einer gültigen Karte gespeichert ist wird gestempelt. Dies würde die hier betrachtete Sicherheitseigenschaft jedoch verletzen. Um den Ansatz zu testen wurde die Nonce entfernt. Die Sicherheitslücke wurde mit der automatisch abstrahierten Version innerhalb weniger Sekunden gefunden. Die *split*-Funktion musste nicht verwendet werden und die Annahme war, dass jede Transitionsregel nur einmal in einem Angriffspfad vorkommen darf.

Eine weitere Sicherheitseigenschaft ist, dass **ausgestellte Tickets aufgrund von Unterbrechungen nicht verloren gehen können**. Bevor ein Ticket gekauft werden kann muss eine Authentifizierung stattfinden und eine gültige PIN verwendet werden. Wenn ein Benutzer ein Ticket kauft, dieses jedoch nicht erhält, weil ein Angreifer die Nachricht mit dem enthaltenen Ticket unterdrückt hat, dann ist das Ticket zwar bezahlt jedoch nicht auf der Karte des Käufers gespeichert. Deshalb wurde das Protokoll so entwickelt, dass ein Ticket solange abgeholt werden kann bis es auf einer Karte erfolgreich gespeichert wurde. Nachdem ein Ticket gekauft, jedoch noch nicht empfangen wurde kann der Kartenbesitzer ein neues Ticket kaufen. In diesem Fall wird das erste Ticket, welches auf dem Server gespeichert ist mit dem neuen Ticket überschrieben und das alte Ticket ist somit verloren. Dies verletzt jedoch die betrachtete Sicherheitseigenschaft. Um den Fehler zu finden ist es notwendig, dass ein Ticket zweimal gekauft wird. Daher muss beim Cl-AtSe Model Checker angegeben werden, das jede Transitionsregel mindestens zweimal in einem Angriffspfad vorkommen darf. Diese führt zu einer höheren Komplexität und längeren Ausführungszeit. Anhand der ETicket-Fallstudie mit automatischen Abstraktionen und der Verwendung der *split*-Funktion, welche eine volle CPU-Auslastung erlaubt, konnte der Angriff selbst nach einer Woche Laufzeit nicht gefunden werden. Um sicherzugehen, dass der Angriff in der ASLan++

Spezifikation tatsächlich vorhanden ist, wurde der Angriffspfad manuell angegeben, wodurch der Angriff gefunden wurde. Diese Sicherheitslücke wurde durch das Attribut *waitingForTicket* geschlossen, welches ausdrückt ob ein Ticket gekauft wurde, jedoch noch nicht abgeholt wurde. *waitingForTicket* wird auf *true* gesetzt bevor die Nachricht zum Kaufen eines Tickets gesendet wird (siehe Abbildung 4.16) und wird wieder auf *false* gesetzt nachdem das Ticket empfangen wurde (siehe Abbildung 4.17).

Die letzte Sicherheitseigenschaft besagt, dass **Tickets nicht mehrfach gestempelt werden**. Eine Sicherheitslücke, die diese Eigenschaft verletzt wurde mittels Model Checking trotz der automatischen Abstraktionen und der Verwendung der *split*-Funktion nicht gefunden. Der Model Checker hat jedoch selbst nach mehreren Wochen Laufzeit immer noch nicht terminiert. Da sich diese Eigenschaft mithilfe der interaktiven Verifikation jedoch nicht beweisen ließ, wurde während der Verifikation eine Sicherheitslücke gefunden, die sehr komplex war und die nur dann auftreten konnte, wenn fast alle Protokollschritte in Kombination betrachtet wurden. Für diesen Angriff musste ein Sitzungsschlüssel zwischen Karte und Service ausgetauscht werden, ein Ticket musste gekauft und auf der Karte gespeichert werden und die Bestätigungsnachricht, um das Ticket vom Service zu löschen, musste unterdrückt werden. Anschließend musste das Ticket vom Schaffnergerät gestempelt werden. Hierfür mussten die öffentlichen Schlüssel zwischen Karte uns Schaffnergerät ausgetauscht, ihre Zertifikate verifiziert und das Ticket ausgewählt werden. Danach musste ein neues Ticket gekauft werden, damit das Klassenattribut *waitingForTicket* auf *true* gesetzt wurde. Hierfür musste jedoch zuvor ein neuer Sitzungsschlüssel ausgetauscht werden. Beim neuen Ticketkauf musste jedoch die Kaufanfrage, welche von der Karte an der Service geschickt wurde unterdrückt werden. Danach konnte das ursprüngliche Ticket durch die Abholfunktion *(retrieveTicket)* vom Service auf die Karte wieder übertragen werden. Dieses Ticket ersetzt dann das gestempelte Ticket (selbe TicketId), jedoch ist dieses Ticket nicht gestempelt. Somit kann es Ticket nochmal gestempelt werden. Dieser Angriff benötigt 73 Schritte, was sehr viel ist.

6.2.3 Evaluation manueller Abstraktionen

Trotz einiger automatischer Abstraktionen ist die generierte Spezifikation zu groß, da selbst nach mehreren Wochen der Model Checker nicht terminiert und einige Sicherheitslücken nicht gefunden werden. Wenn eine zu überprüfende Spezifikation zu groß ist, ist es gängige Praxis die Spezifikation von Hand zu abstrahieren [5]. Hierbei wird jedoch das Verhalten der Anwendung verändert, wodurch Sicherheitslücken unter Umständen nicht mehr gefunden werden können. Somit wird hierfür Expertenwissen benötigt. Trotzdem können hierdurch Teile großer Anwendungen mithilfe von Model Checking gezielt auf Sicherheitslücken überprüft werden. In Abschnitt 4.4 wurde bereits beschrieben wie solche manuellen Abstraktionen in einem SecureMDD-Anwendungsmodell modelliert werden. Textuelle Spezifikationen von großen Anwendungen können sehr verwirrend sein und manuelle Änderungen

sind zeitraubend und fehleranfällig. Ein großer Vorteil von einem modellgetriebenen Ansatz ist, das eine Anwendung unterschiedliche Abstraktionssichten hat. Es ist um einiges einfacher eine große Anwendung auf einer abstrakteren Ebene einzuschränken. Eine einzige Änderung auf einer abstrakten Ebene (Anwendungsmodell) resultiert in vielen Änderungen auf einer darunter liegenden Ebene (ASLan++ Spezifikation).

Ein Beispiel hierfür ist Abbildung 4.23 in Abschnitt 4.4. Dieses besteht nur aus drei UML-Elementen und reduziert die ursprüngliche Spezifikation um 250 Codezeilen. Hierbei handelt es sich um nicht zusammenhängende Zeilen, wodurch eine manuelle Änderung viel mehr Aufwand und Expertenwissen benötigen würde. Durch diese manuelle Abstraktion konnte die zweite Sicherheitseigenschaft, dass *ausgestellte Tickets aufgrund von Unterbrechungen nicht verloren gehen können* überprüft werden und der Fehler wurde innerhalb von wenigen Sekunden gefunden. Hierbei war es offensichtlich welche Aktivitäten für diese Sicherheitseigenschaft relevant sind und die nicht benötigten Aktivitäten wurden entfernt. Die Angreiferfähigkeiten wurden nicht verändert.

Durch die manuellen Abstraktionen in Abbildung 4.24 in Abschnitt 4.4 konnte auch der Angriffe gefunden werden, welcher die dritte Sicherheitseigenschaft verletzt. Nämlich, dass *Tickets nicht mehrfach gestempelt werden.* Durch die manuelle Abstraktion wurde die originale Spezifikation zwar um 200 Zeilen vergrößert, jedoch konnten die Aktivitäten nur innerhalb einer eingeschränkten Reihenfolge ausgeführt werden. Zudem wurden die Angreiferfähigkeiten auf eine feingranulare Weise eingeschränkt und nur bestimmte Aktivitäten konnten mehrfach ausgeführt werden. Dies führte zu einer enormen Einschränkung der möglichen Kombinationen, wodurch die Sicherheitslücke in 28 Minuten gefunden werden konnte. Die Annahme war, dass es eine Sicherheitslücke beim Kaufen und Empfangen eines Tickets geben könnte. Somit hat der Angreifer volle Fähigkeiten bei diesen zwei Aktivitäten erhalten.

Durch die vorgestellten Abstraktionen konnten somit die drei existierenden Sicherheitslücken in der ETicket-Fallstudie in wenigen Minuten gefunden werden ohne den genauen Angriff zu spezifizieren.

6.3 ASLan nach SecureML

Die Transformationen von SecureML nach ASLan++, sowie der Model Checker Cl-AtSe wurden in MagicDraw als Plugin integriert. Ein Entwickler, welcher eine sicherheitskritische Anwendung mit SecureML modelliert, kann sein Anwendungsmodell automatisch nach ASLan++ transformieren lassen und mit dem Model Checker Cl-AtSe auf Sicherheitslücken prüfen. Es können jedoch auch nur die Teile des System betrachtet werden, die durch manuelle Abstraktionen (vorgestellt in Abschnitt 4.4) definiert wurden und es kann ausgewählt werden, wie oft eine Transitionsregel in einem Angriffspfad vorkommen darf. Wenn eine Sicherheitslücke gefunden wurde, dann erhält der Entwickler als Ausgabe den Angriffspfad und kann

den Angriff nachvollziehen. Ein Problem hierbei ist jedoch, dass der Angriffspfad sich auf die ASLan-Spezifikation bezieht. Nun müsste der Anwendungsentwickler den Angriffspfad mit der ASLan-Spezifikation vergleichen und diese mit dem SecureML-Modell. Die ASLan-Spezifikation wird jedoch generiert und sollte somit transparent für den Anwendungsentwickler sein. Um dies zu erreichen, wird der Angriffspfad in eine für den Anwendungsentwickler bekannte und lesbare Notation transformiert. Hierzu wurde eine textuelle Notation von SecureML verwendet, welche die gleichen MEL-Ausdrücke nutzt.

Ein Beispiel für einen Angriffspfad, der automatisch nach SecureML übersetzt wurde ist in Listing 6.8 zu sehen.

```
Intruder Knowledge:
_____________________________

ETicketCard
    certCard_data_publicKey_1
_____________________________


STEP 1:
Intruder -> InspectorDevice : uStampTicket()
InspectorDevice -> Intruder : stampTicket(...)

STEP 2:
Intruder -> InspectorDevice : stampTicketSuccessful(
                                encrypt(certCard_data_publicKey_1,
                                    stampTicketSuccessfulMsg(nonce_1175)
                                )
                              )
InspectorDevice -> Intruder : uStampTicketSuccessful()
```

Listing 6.8: Angriffspfad in SecureMDD

Der Angriffspfad verletzt die Sicherheitseigenschaft, dass **nur ausgestellte Tickets gestempelt werden**. Im folgenden Angriff kann die Bestätigungsnachricht, welche beschreibt, dass der Stempelvorgang erfolgreich war, gefälscht werden und somit kann dem Schaffnergerät (*InspectorDevice*) ein erfolgreicher Stempelvorgang vorgetäuscht werden. Hierzu sendet der Angreifer (Intruder) im ersten Schritt die Nachricht *uStampTicket* ohne Parameter an das *InspectorDevice*. Dieser antwortet mit der Nachricht *stampTicket*. Diese Nachricht besitzt Parameter, die jedoch für den Angriff irrelevant sind. Damit der Angriffspfad übersichtlich bleibt werden irrelevante Informationen automatisch weggelassen. Im zweiten Schritt überspringt der Angreifer das Weiterleiten der Nachricht *stampTicket* an die Karte und sendet stattdessen eine selbst erstellte Antwortnachricht vom Typ *stampTicketSuccessful* an das *InspectorDevice*. Die erstellte Nachricht enthält eine Nonce (nonce_1175), welche mit dem öffentlichen Schlüssel *certCard_data_publicKey_1* verschlüsselt ist. Dieser Schlüssel ist dem Angreifer initial bekannt und steht somit in seinem initialen Angreiferwissen. Das initiale Angreiferwissen steht über dem Angriffspfad. Die übertragene Nonce wurde vom Angreifer erstellt. Da es besonders wichtig ist zu unterscheiden welche Daten vom Angreifer erstellt wurden und welche von einem validen Agenten stammen, werden alle vom Angreifer erstellten Daten rot markiert. Zudem wird die Klassenstruktur wieder zurücküber-

setzt. *stampTicketSuccessful* bezeichnet ein Objekt vom Typ *StampTicketSuccessful(...)* und *stampTicketSuccessfulMsg(...)* ein Objekt vom Typ *StampTicketSuccessfulMsg*. Beim nicht transformierten Attacktrace wird die Verschlüsselung in der ASLan Notation durch *plainText_key* dargestellt. Dies wird wieder zurück nach *encrypt(key,plainText)* übersetzt. Zudem wird die Initialisierung der Environment, der Session und der Agenten sowie die Zustandsübergänge der Agenten weggelassen.

6.4 Verwandte Arbeiten

In diesem Kapitel wurde ein existierender Model-Checker in den SecureMDD-Ansatz integriert. Es existieren einige Model Checker (z.B. NuSMV[28], SPIN[43], PRISM[42]), jedoch nur wenige die für sicherheitskritische Anwendungen zugeschnitten sind.

Mit UMLSec [51] ist es möglich kryptographische Protokolle zu modellieren und diese mit SPIN Model-Checken zu lassen. UMLSec unterstützt jedoch nicht die Modellierung der Protokollschritte, sondern nur die der Kommunikation. Zudem werden nur Standardsicherheitseigenschaften unterstützt.

In [81] beschreiben Wimmel und Wisspeintner eine Erweiterung von AUTOFOCUS [44] für die Modellierung von sicherheitskritischen Systemen. Durch den Anschluss an SMV [15] können diese Modelle auf Sicherheitslücken überprüft werden. Die Sicherheitseigenschaften werden jedoch durch generische *Security Patterns* definiert. Zudem ist keine Version von AUTOFOCUS, die diese Erweiterung beinhaltet öffentlich verfügbar.

In diesem Kapitel wurde AVANTSSAR[4] in den SecureMDD-Ansatz integriert. Dabei handelt es sich um ein Tool zum automatischen Validieren der Sicherheit von serviceorientierten Architekturen. Es werden kryptographische Operationen sowie verschiedene Angreifermodell unterstützt. Zudem können anwendungsspezifische Sicherheitseigenschaften definiert werden und dass Tool ist öffentlich zugänglich und gut dokumentiert.

KAPITEL 7

Interaktive Verifikation der entwickelten Web Service-Anwendungen

Um die Sicherheit einer mit SecureMDD entwickelten Anwendung zu garantieren ist formale Verifikation notwendig. Hierzu muss die gesamte modellierte Anwendung einschließlich der modellierten Sicherheitseigenschaften in ein formales Modell transformiert werden. Anschließend kann anhand dieses formalen Modells die Anwendung verifiziert werden. Damit die Sicherheitseigenschaften, welche für das formale Modell gezeigt wurden auch für die generierte Anwendung gelten, muss der generierte Programmcode eine Verfeinerung (Refinement) des formalen Modells sein. Dies bedeutet, dass das formale Modell abstrakter sein darf, jedoch nicht konkreter. Im Gegensatz zum Model Checking welches im Kapitel 6 erläutert wurde, hat die interaktive Verifikation einige Vorteile. Einerseits sind selbst große Anwendungen als ganzes verifizierbar und andererseits kann eine beliebige aber feste Anzahl an Agenteninstanzen sowie Protokollschrittausführen betrachtet werden. Somit können alle Abläufe die zur Laufzeit des Systems mit einer Vielzahl an Teilnehmern stattfinden können berücksichtigt werden. Um Web Service-Anwendungen formal verifizieren zu können, mussten Web Services und deren Eigenschaften formal spezifiziert werden. Hierzu gehört die Formalisierung von Verbindungen und Serviceaufrufen sowie die Spezifikation einer Sitzungsverwaltung oder Standardprotokollen wie Transport Layer Security. Da das Einsatzgebiet von Web Services sehr groß ist, wurde zudem die Integration von existierende (externe) Services betrachtet. Damit sich die Sicherheitseigenschaften entlang des Refinements vererben, durfte der Programmcode nicht manuell erweitert werden, sondern die Integration der externen Services musste in allen Phasen des Entwicklungsprozesses betrachtet werden. Ein wichtiger Punkt hierbei war, dass die externen Services mit wenig Verifikationsaufwand und somit unabhängig von den zu zeigenden Sicherheitseigenschaften ausgetauscht werden können. Ein ebenfalls wichtiger und keinesfalls trivialer Punkt war die Transformation der modellierten anwendungsspezifischen Sicherheitseigenschaften in das formale Modell. In allen erwähnte Bereichen spielt der Angreifer eine wichtige Rolle und musste somit immer betrachtet werden. In Zusammenhang mit dieser Arbeit wurden

die gezeigten Sicherheitseigenschaften für die ETicket-Anwendung sowie für die Debitcard-Anwendung verifiziert.

Dieses Kapitel ist wie folgt gegliedert. In Abschnitt 7.1 werden die wichtigsten Grundlagen anhand der ETicket-Anwendung vorgestellt. Im darauf folgenden Abschnitt 7.2 werden servicespezifische Herausforderungen wie die Spezifikation einer Sitzungsverwaltung sowie die Formalisierung von TLS und einer neuen Kommunikationsstruktur für kaskadierende Serviceaufrufe erörtert. Abschnitt 7.4 zeigt, wie anwendungsspezifische Sicherheitseigenschaften, die mit erweitertem OCL definiert wurden in das formale Modell übersetzt werden. In Abschnitt 7.5 wird die Integration von externen Web Services in das formale Modell beschrieben sowie die Austauschbarkeit der externen Web Services ohne erneute Verifikation. Abschnitt 7.6 erläutert die Verifikation einer mit SecureMDD erstellten Anwendung und Abschnitt 7.7 diskutiert verwandte Arbeiten. Die formale Spezifikation von Services sowie externen Services und deren Verifikation anhand modellierter Sicherheitseigenschaften ist in [16, 22, 19] publiziert.

7.1 Grundlagen des formalen Modells

Für Smart Card-Anwendungen wurde das formale Modell in der Arbeit von Moebius [62] vorgestellt. In diesem Abschnitt werden die wichtigsten Grundlagen des generierten formalen Modells anhand der ETicket-Anwendung vorgestellt. In Abschnitt 7.1.1 wird ein Überblick über den Transformationsprozess gegeben. Abschnitt 7.1.2 beschreibt den statischen Teil des formalen Modells, z.B. Datentypen, Komponenten der Anwendung und Nachrichtentypen und in Abschnitt 7.1.3 wird die Spezifikation der dynamischen Aspekte eines Systems mithilfe von abstrakten Zustandsmaschinen (ASMs) gezeigt.

7.1.1 Übersicht über den Transformationsprozess

Das formale Modell, welches aus UML-Diagrammen generiert wird, beschreibt eine abstrakte Sicht auf das gesamte UML-Modell (welches eine Repräsentation des gesamten mit SecureMDD entwickelten Systems ist). Es enthält eine beliebige, aber endliche Anzahl an Komponenteninstanzen (Chipkarten, Terminals und Services) und hat eine beliebige Anzahl an verzahnten Protokollschritten.

Die statischen Aspekte einer Anwendung, wie z.B. die Komponenten, die Datentypen, die Kommunikationsinfrastruktur und der Angreifer, werden durch algebraische Spezifikationen definiert. Der dynamische Teil einer Anwendung, wie z.B. die kryptographischen Protokolle, werden durch abstrakte Zustandsmaschinen (ASM) beschrieben. Die ASM besteht aus zwei Arten von Regeln: Regeln, die das Verhalten des Angreifers beschreiben und Regeln, die das Verhalten der Komponenten (*agenten* im formalen Modell) beschreiben. Die Ausführung von Regeln erzeugt einen Pfad von Zuständen. Da anwendbare Regeln nichtdeterministisch ausgewählt werden, wird eine Menge von Pfaden erzeugt, die alle sequen-

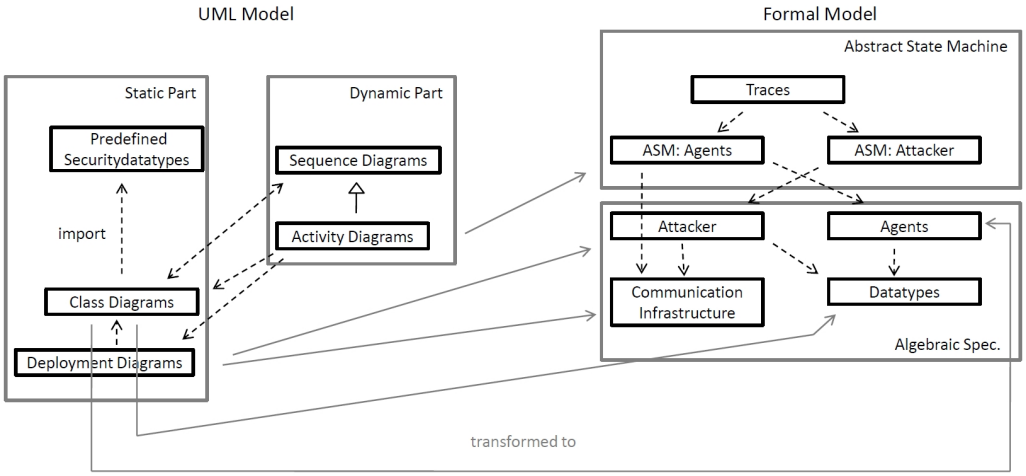

Abbildung 7.1: Abhängigkeiten zwischen UML-Modell und formalem Modell

ziellen Kombinationsmöglichkeiten enthält. Abbildung 7.1 zeigt die Komponenten des formalen Modells und aus welchem Teil des UML-Models sie generiert werden. Aus technischer Sicht wird ein UML-Modell mit dem Eclipse Modeling Framework (EMF) nach Eclipse geladen und die Spezifikation für den interaktiven Theorembeweiser KIV [8] wird mit XPand[1], ebenfalls Teil vom EMF, generiert. Andere Beweiser könnten durch das Implementieren von weiteren XPand-Transformationen unterstützt werden. Das generierte formale Modell wird anschießend verwendet um die Sicherheit der modellierten Anwendung mit KIV zu beweisen.

7.1.2 Statischer Teil

Datentypen

Die Daten einer Anwendung, wie z.B. die Nachrichten, vordefinierte Sicherheitsdatentypen und Datentypen, die im Klassendiagramm definiert wurden, werden in algebraische Spezifikationen übersetzt. Um die Lücke zwischen dem UML-Model und dem formalen Model zu reduzieren, werden dieselben Datentypen wie im Klassendiagramm verwendet und kein generisches Datenformat wie in einigen anderen Ansätzen (z.B. [71, 39]). Dies vereinfacht die Verifikation enorm. Zum Beispiel wird in der ETicket-Fallstudie ein Ticket als frei generierter Datentyp spezifiziert

```
Ticket = mkTicket(. .id_ID : ID;
                  . .ticketInfo_TicketInfo : TicketInfo;
                  . .stamp_Stamp : Stamp);
```

[1] http://projects.eclipse.org/projects/modeling.m2t.xpand

und der Datentyp *message*, welcher eine beliebige modellierte Nachricht sein kann besteht aus 40 Nachrichten:

```
message =
 mkBuyTicket(. .msg_EncData : EncData) with isBuyTicket
 | mkBuyTicketSuccessful with isBuyTicketSuccessful
 ...
```

Die Spezifikation hat nicht die Absicht, Instanzen beliebiger Klassenstrukturen getreu zu repräsentieren. Ohne einen Heap oder Referenzen (oder etwas ähnlichem) ist es nicht möglich beliebige Pointerstrukturen wie Zyklen oder geteilte Objekte zu modellieren. Jedoch ist dies nicht notwendig, da die Nachrichten und Daten die in einem kryptographischen Protokollen genutzt werden solche Besonderheiten nicht verwenden (und nicht verwenden sollen).

Dynamische Funktionen

Jede Instanz einer Komponentenklasse (Service, Terminal, Chipkarte oder Benutzer) wird zu einem Agenten in der formalen Spezifikation und besitzt ein Verhalten, welches durch die modellierten Protokollschritte definiert ist. Die Attribute der Komponentenklassen sind als dynamische Funktionen spezifiziert und können durch ASM-Regeln modifiziert (aktualisiert) werden. Für jedes Attribut wird eine dynamische Funktion definiert, die einen Agenten auf seine Attributwerte abbildet. Zum Beispiel ist

```
ETicketService-state : agent  →  StateService;
```

die dynamische Funktion für das *state*-Attribut der Klasse *ETicketService* und *ETicketService-state(ag)* gibt den Wert des Attributes für einen gegebenen *ETicket Service*-Agenten *ag* zurück. Die ETicket-Anwendung enthält 40 solcher dynamischen Funktionen. Erfahrungen haben gezeigt, dass das Zerlegen einer Klasse in seine Attribute, im Gegensatz zu einer dynamischen Funktion, die den gesamten Zustand eines Agenten zurückgibt, die Verifikation vereinfacht. Dies liegt daran, dass eine Aktualisierung eines Attributes und ein lesender Zugriff auf ein anderes Attribut syntaktisch unabhängig voneinander sind.

Kryptographie und der Angreifer

Die vordefinierten Sicherheitsdatentypen sind unabhängig von der konkreten Anwendung und die Transformation ist generisch für alle Anwendungen. Zum Beispiel sind verschlüsselte Daten als frei generierter Datentype *EncData* spezifiziert:

```
EncData = mkEncData( . .key : SymmKey; . .plain :
PlainData);
```

Das bedeutet, die verschlüsselten Daten beinhalten den Schlüssel, der zum Verschlüsseln verwendet wurde. Dieser Ansatz ist ähnlich zu [71]. Hierdurch ist es einfach zu spezifizieren, ob etwas entschlüsselt werden kann oder nicht. Der Schlüssel muss hierfür der Gleiche sein, der zum Verschlüsseln verwendet wurde (private und öffentliche Schlüsselpaare für asymmetrische Verschlüsselung). In dieser Spezifikation des Angreifers, hat er keinen direkten Zugang zum Schlüssel, sodass das Verhalten der kryptographischen Datentypen und Operationen dem in der Realität entspricht.

Der Angreifer ist nur implizit durch den Stereotypen ≪Threat≫ im UML-Modell enthalten. In der formalen Spezifikation ist der Angreifer explizit als eigenständiger Agent spezifiziert. Dieser ist in der Lage mit anderen Komponenten durch das Senden und Empfangen von Nachrichten zu interagieren. Er ist verbunden mit einer Menge von Daten, welche sein Wissen repräsentieren. Wenn der Angreifer an einem Kommunikationspfad mitliest und Daten über diesen übertragen werden, dann werden diese Daten analysiert und zu seinem Angreiferwissen hinzugefügt. Wenn der Angreifer z.B. eine verschlüsselte Nachricht erhält und den Schlüssel kennt, dann kann er die Nachricht entschlüsseln und den Inhalt analysieren um z.B. eine geheime PIN zu erfahren. Oder, falls er einen Schlüssel erhält kann er zuvor erhaltene verschlüsselte Nachrichten entschlüsseln, die wiederum andere Schlüssel beinhalten könnten. Dies ist ebenfalls ähnlich zu [71]. Umgekehrt kann der Angreifer nur Nachrichten senden, die er mit seinem Wissen erzeugen kann. Er kann ihm bekannte Daten mit einem ihm bekannten Schlüssel verschlüsseln und diese verschlüsselten Daten versenden. Er kann ebenfalls Nachrichten die er zu einem früheren Zeitpunkt erhalten hat versenden (Replay-Angriff), sogar wenn er die Nachricht nicht entschlüsseln kann. Der Angreifer ist nicht in der Lage verschlüsselte Nachrichten ohne den Schlüssel zu entschlüsseln, beliebige Schlüssel zu generieren oder beliebige Nonces und Geheimisse zu raten. Dies modelliert die Tatsache, dass es praktisch unmöglich ist einen Schlüssel oder eine Nonce durch Zufall oder brute-force während der Laufzeit der Anwendung zu finden. Dies wird mitunter als "perfect cryptography assumption" oder "symbolic cryptography" bezeichnet. Nicht kryptographische Daten wie Zahlen und Zeichenfolgen sind niemals geheim und können immer vom Angreifer in einer Nachricht verwendet werden. Deshalb muss eine PIN als *Secret* modelliert werden (1234 ist als Zahl jedem bekannt, jedoch als PIN sollte diese geheim bleiben).

Es werden nur die Spezifikationen für die vordefinierten Sicherheitsdatentypen und kryptographischen Operationen generiert die auch tatsächlich verwendet werden. Die ETicket-Anwendung verwendet symmetrische und asymmetrische Verschlüsselung, Digitale Signaturen, Nonces und Secrets (PINs).

7.1.3 Dynamischer Teil

Der dynamische Teil einer Anwendung, wie z.B. die Sicherheitsprotokolle, werden mit UML in Aktivitätsdiagrammen modelliert. Im formalen Modell werden diese in eine ASM übersetzt. Die ASM besteht aus einer Anzahl von Regeln, welche hauptsächlich die individuellen Protokollschritte beschreiben. Eine anwendbare Regel (ein STEP) wird nichtdeterministisch in einem gegebenen Zustand gewählt und führt nach der Ausführung zu einem neuen Zustand. Regeln werden beliebig oft ausgeführt (STEP*). Somit werden alle möglichen (endlichen und unendlichen) Abläufe der spezifizierten "Welt" generiert:

```
ASM = STEP*
```

Die ASM-Regel STEP wählt nichtdeterministisch eine Aktion zum Ausführen aus. Mögliche Schritte sind ein *attacker-step* oder ein Schritt eines Komponententypen. Wenn z.B. der gewählte Protokollschritt *ETicketService-agent-step* ist, wird ein beliebiger Agent *ag* vom Typ *ETicketService* gewählt und die ASM-Regel *ETICKETSERVICESTEP* wird für diesen Agenten ausgeführt.

```
STEP {
let asm-step = [?] in {
 if asm-step = attacker-step
 then ATTACKER(...)
 else if asm-step = ETicketService-agent-step
   then {choose ag with
      exETicketService(ag) in ETICKETSERVICESTEP(...) }
   else ...
```

ETICKETSERVICESTEP enthält alle Protokollschritte, die der ETicket-Service ausführen kann. Der Nachrichtenaustausch wird durch Eingabequeues (Inboxes) modelliert. Ein Protokollschritt kann ausgeführt werden, wenn die Inbox einer Komponente eine Nachricht vom entsprechenden Typ besitzt. Anderenfalls passiert nichts.

```
ETICKETSERVICESTEP {
choose port with is-valid-port(ag, port) and
 inputs(ag)(port) ≠ []
in {
 let inmsg = inputs(ag)(port).first in {
  inputs := rem(ag, port, inputs);
  if (isClientHello(inmsg)) then ClientHello else
  if (isSessionKey(inmsg)) then SESSIONKEY else
  if (isBuyTicket(inmsg)) then BuyTicket else
  if (isRetrieveTicket(inmsg)) then BuyTicketSuccessful else
  ...
```

Die Nachricht wird aus der Inbox entfernt und die entsprechende ASM-Regel ausgeführt. Die Regel bearbeitet die Nachricht, führt Überprüfungen aus und verändert den Zustand des Agenten, wie es im entsprechenden Protokollschritt innerhalb eines Aktivitätsdiagramms modelliert ist. Das Senden einer Nachricht geschieht, indem die Nachricht zur Inbox des Empfängers hinzugefügt wird. Falls der Angreifer auf einem Kommunikationspfad die Fähigkeit besitzt Nachricht mitzulesen (*read*-Fähigkeit im Deploymentdiagramm), dann werden die übertragenen Nachrichten ebenfalls zum Angreiferwissen hinzugefügt.

```
BuyTicket {
let enc = inmsg .msg in
 if (serviceState(ag) = AUTHENTICATED) then
  if (not (can_decrypt(sessionKey(ag), enc) ∧
  isBuyInfo(decrypt(sessionKey(ag), enc)))) then
  STOPSTEP
 else
  let msg = decrypt(sessionKey(ag), enc).buyInfo in ...
```

BuyTicket beschreibt eine ASM-Regel. Abgesehen von einigen syntaktischen Unterschieden ist das abstrakte Programm sehr ähnlich zum modellierten Aktivitätsdiagramm. Es besitzt die gleichen Kontrollstrukturen, Überprüfungen und Zuweisungen. Jedoch gibt es Unterschiede wie z.B. das im Anwendungsmodell die Überprüfung *if(not (can_ decrypt ...)* nicht vorkommt. Diese wurde während der Transformationen hinzugefügt. MEL enthält lediglich die Anweisung

```
msg : BuyInfo := decrypt(sessionKey, enc);
```

decrypt ist eine vordefinierte MEL-Operation, die sich im generierten Code (und somit in der realen Welt) wie folgt verhält: Die Daten werden verschlüsselt (durch die Anwendung eines Algorithmus wie 3DES oder AES) und anschließend wird überprüft ob das Ergebnis vom erwarteten Typ ist (*BuyInfo* im oberen Beispiel). Ist dies nicht der Fall, wird eine Exception geworfen. Der abstrakte Code hat dasselbe Verhalten. Wenn mit einem falschen Schlüssel entschlüsselt wird, dann schlägt *can_ decrypt* fehl und das Resultat der Entschlüsselung ist eine sinnlose Abfolge von bytes (für sichere Algorithmen wie AES). Wenn der Schlüssel korrekt ist, dann ist das Ergebnis sinnvoll, jedoch könnte es von einem anderen Typ sein. Dies wird im zweiten Teil getestet (*isBuyInfo(decrypt(...))*). Da die abstrakte Programmiersprache keine Exception unterstützt, wird dasselbe Verhalten durch if-then-else erreicht.

Eine ASM-Regel wird atomar ausgeführt. Es wird angenommen, dass ein Angreifer das, was in einer Komponente passiert, nicht beeinflussen oder manipulieren kann. Zudem kann er auch nicht den lokalen Zustand der Komponente auslesen.

Ein Angreifer kann lediglich mit den Nachrichten und Inboxes, wie es im Deploymentdiagramm definiert wurde, interagieren (siehe Abbildung 4.13 in Abschnitt 4.2.4). Eine weitere Konsequenz dieser Atomizität ist, dass der Zugriff auf Web Services synchronisiert werden muss (siehe Abschnitt 5.1.4). Dies kann zu Effizienzproblemen führen, ist derzeit jedoch notwendig um das gleiche Verhalten für das Model und den generierten Code zu erzielen.

Wenn anstelle einer Komponente der Angreifer ausgewählt wurde, dann wird die ASM-Regel für den Angreifer aufgerufen. Dann wird nichtdeterministisch gewählt ob der Angreifer eine Nachricht unterdrückt oder versendet. Falls der Angreifer eine Nachricht unterdrücken soll, dann wird eine nichtleere Inbox gewählt, die zu einem Übertragungskanal gehört auf dem Angreifer Nachrichten unterdrücken kann. Anschließend wird eine Nachricht aus der Inbox gelöscht. Falls der Angreifer eine Nachricht senden soll, generiert sich der Angreifer eine Nachricht aus seinem aktuellen Wissen. Die Nachricht wird anschließend zu einer zufällig gewählten Inbox geschickt, auf die der Angreifer Zugriff hat.

7.2 Web Services

Web Services unterscheiden sich erheblich von Chipkarten. Eine Chipkarte kann von einem Terminal aufgerufen werden und kann nach dem Bearbeiten der empfangenen Nachricht entweder den Protokollablauf beenden oder eine Nachricht zurück an das Terminal senden. Im Gegensatz zu Chipkarten kann ein Service wieder andere Services aufrufen, wodurch sich kaskadierende Serviceaufrufe ergeben. Die Kommunikation mit Chipkarten erfolgt über Kartenlesegeräte, die an einem Terminal angeschlossen sind. In ein solches Kartenlesegerät können Chipkarten von Benutzern hineingesteckt und wieder herausgezogen werden. Ein Web Service wird jedoch vom Client über eine URL aufgerufen. Ein ebenfalls wichtiger Unterschied gegenüber Chipkarten ist, dass ein Service von mehreren Clients gleichzeitig aufgerufen werden kann. Dies kann bei Chipkarten nicht geschehen, da ein Chipkarte zur gleichen Zeit nur in einem Kartenleser stecken kann. Ein weiterer Unterschied ist, dass für die Kommunikation mit Web Services etablierte Standardprotokolle wie Transport Layer Security existieren, welche die Nachrichtenübertragung zwischen Services gegenüber einem Angreifer schützen. Alle diese Unterschiede haben großen Einfluss auf die formale Spezifikation und insbesondere auf das Verhalten des Angreifers.

7.2.1 Stateful Web Services

Services wurden in SecureMDD als stateful Web Services implementiert. Ein solcher Service erstellt für jeden Aufrufer eine neue Sitzungsinstanz. Im tatsächlichen Programmcode bedeutet das, dass beim Aufruf eines Services ein Servicemanager kontaktiert wird und dieser eine neue Sitzungsinstanz nur für den Aufrufer erzeugt, veröffentlicht und dem Aufrufer die Adresse übergibt. Dieses Verhalten

wird durch die Verwendung eines Frameworks im generierten Code realisiert (siehe Abschnitt 5.1.1). Im SecureML-Modell sowie im formalen Modell wird hiervon jedoch abstrahiert. Im generierten Programmcode gibt es keine zusätzlichen Sicherheitslücken, die formal nicht betrachtet werden. Es wird angenommen, dass für die reale Anwendung ein Angreifer dieselben Fähigkeiten für die Kommunikation zwischen Aufrufer und Servicemanager sowie zwischen dem Aufrufer und seiner Sitzungsinstanz besitzt. Dies ist realistisch, da der Aufrufer derselbe ist und der Servicemanager sowie die Sitzungsinstanz auf demselben Server liegen. Dies bedeutet auch, dass wenn TLS auf eine Leitung angewendet wird, dann gilt dies für den Servicemanager sowie für die Sitzungsinstanz. Des Weiteren kann ein Service nur in einem unsicheren Netzwerk wie dem Internet veröffentlicht werden oder in einem sicheren Netzwerk wie einem Intranet. Der Servicemanager überträgt nur die Adresse einer neu erstellten Sitzungsinstanz. Falls dies über ein sicheres Netzwerk oder über eine TLS-Verbindung in einem unsicheren Netzwerk geschieht, kann der Angreifer die Adresse durch die eingeschränkten Angreiferfähigkeiten nicht lesen und auch nicht manipulieren. Ist ein Services über ein unsicheres Netzwerk erreichbar und wird kein TLS verwendet, besitzt der Angreifer die Fähigkeiten *read, send, suppress* auf der Leitung zum Service. Somit kann der Angreifer durch das Einschleusen einer selbst gewählten Adresse den Aufrufer dazu bringen mit dem Angreifer zu kommunizieren. Hierdurch kann ein Angreifer jedoch nicht mehr als er durch seine Fähigkeiten *read, send, suppress* auf der Leitung sowieso schon kann. Die vom Angreifer eingeschleuste Adresse kann aber auch die von einer anderen (vom Angreifer mitgehörten) Sitzungsinstanz sein, wodurch der Aufrufer mit dieser kommunizieren würde. Durch die Fähigkeiten *read, send, suppress* kann ein Angreifer jedoch ohnehin alle Nachrichten zu einer beliebigen aber schon erzeugten Sitzungsinstanz dieses Servicemanagers umleiten.

Somit kann vom Servicemanager abstrahiert werden. Ein stateful Web Service wird als Serviceklasse modelliert, welche mehrfach instanziiert werden kann (bei der Onlinebanking-Fallstudie können dadurch beliebig viele konkrete Banken betrachtet werden). Diese Serviceinstanz entspricht einer Instanz des Servicemanagers. Jede dieser Serviceinstanzen erstellt für die Aufrufer neue Sitzungsinstanzen. Alle Sitzungsinstanzen werden als eine beliebige Menge von Serviceagenten formalisiert. Zwar sind alle Agenten die zu einer Serviceinstanz gehören gleich initialisiert, jedoch nicht die Agenten unterschiedlicher Serviceinstanzen. Deshalb muss ein Serviceagent einer Serviceinstanz zugeordnet werden.

7.2.2 Serviceaufrufe

Die modellierten Serviceklassen wie auch Smart Card-Klassen können mehrfach (beliebig, aber endlich oft) instanziiert werden. Dadurch ergeben sich viele Instanzen vom gleichen Typ, zwischen denen bei der Kommunikation unterschieden werden muss. Bei Chipkarten geschieht dies durch das Einstecken einer Chipkarte in ein Kartenlesegerät. Da dies vom Benutzer gemacht wird, müssen alle Kombinationsmöglichkeiten betrachtet werden, was durch einen nichtdeterministischen

Verbindungsaufbau und -abbau zu den verschiedenen Smart Card-Instanzen formalisiert ist. Bei Services hingegen erfolgt die Auswahl der konkreten Serviceinstanz explizit durch die Serviceadresse (URL). In SecureML wird dies durch den Stereotyp ≪sendTo≫ und einer *URL* modelliert (siehe Abschnitt 4.2.2).

Beim Aufbau einer Verbindung zu einer Serviceinstanz wird eine neue Sitzungsinstanz erstellt. In der formalen Spezifikation werden jedoch nur die Sitzungsinstanzen als Serviceagenten formalisiert. Die Serviceinstanz ist nur implizit durch seine Adresse definiert. Deshalb wird für jede Serviceklasse eine Variable spezifiziert. Für die Bankklasse aus der Onlinebanking-Anwendung in Abschnitt 4.2.3 ergibt sich:

```
next-BankAgentName : nat;
```

Diese Variable gibt einen Agentennamen zurück, der beim Erzeugen eines neuen Serviceagenten inkrementiert wird, wodurch die Serviceagenten einer bestimmten Klasse eindeutig beschrieben werden. Hierdurch ist jedoch nicht spezifiziert zu welcher Serviceinstanz der Agent gehört. Deshalb wird im formalen Modell eine dynamische Funktion generiert, die einen Agenten der jeweiligen Serviceklasse auf die Adresse seiner Serviceinstanz abbildet. Für die Bankklasse ergibt sich die folgende dynamische Funktion:

```
BankAddress : (agent  →  string);
```

Da jede Serviceinstanz ihre initialen Attribute anders initialisieren kann, werden diese abhängig von der Serviceinstanzadresse gespeichert. Deshalb ist für jedes initiale Attribut einer Serviceklasse eine zusätzliche dynamische Funktion notwendig. Für das *verifyKey*-Attribut der Bankklasse ergibt sich:

```
Initial-Bank-verifyKey : (string  →  PublicKey);
```

Somit können die initialen Attribute für jede Serviceinstanz separat initialisiert und beim Erstellen den Serviceagenten übergeben werden.

```
1  ...
2  if (attackerAddress = URL) then {
3   CONNECTTOSERVICE(ag, port, ATTACKER; connections, inputs);
4   SEND(msg, port, ag, connections; attacker-known, inputs);
5  } else if (serviceClassName(URL) = Bank) then {
6    let ag0 = Bank(next-BankAgentName) in {
7     BankAddress(ag0) := URL;
8     Bank-verifyKey(ag0) := Initial-Bank-verifyKey(URL);
9     ...
10    next-BankAgentName := next-BankAgentName + 1;
11    CONNECTTOSERVICE(ag, port, ag0; connections, inputs);
12    SEND(msg, port, ag, connections; attacker-known, inputs);
13   }
14 }
```

Listing 7.1: Aufruf des Services *Bank*

Listing 7.1 zeigt den Aufruf des Services *Bank* anhand einer Adresse, die in der Variable *URL* gespeichert ist. Der Inhalt der Variable ist dynamisch und könnte sogar über eine unsichere Leitung übertragen worden sein. Deshalb wird zuerst überprüft ob die Adresse dem Angreifer gehört (Zeile 2). Falls dies der Fall ist, wird eine Verbindung zum Angreifer (der sich als Service ausgibt) aufgebaut und die Nachricht an den Angreifer geschickt (Zeile 3-4). Falls es nicht die Angreiferadresse ist, wird überprüft ob die Adresse zu der Serviceklasse passt, zu der die Nachrichtenübertragung modelliert wurde (Zeile 5). Hierzu wird die dynamische Funktion *serviceClassName* verwendet. Diese bildet eine Adresse auf einen *serviceClassName*[2] ab. Anschließend wird anhand der Variable *next-BankAgentName* ein Agentenname zurückgegeben und mit diesem der Serviceagent erzeugt (Zeile 6). In Zeile 7 wird mithilfe der dynamischen Funktion *BankAddress* spezifiziert, dass dieser Agent zur Serviceinstanz gehört, welche durch diese Adresse repräsentiert wird. Danach werden die initialen Attribute der Serviceinstanz dem Serviceagenten zugewiesen (Zeile 8-9). Anschließend wird der Agentenname für diese Adresse inkrementiert (Zeile 11). Dadurch wird sichergestellt, dass ein weiterer Aufrufer keinen schon verwendete Serviceagenten zugewiesen bekommt. Somit ist nun bekannt mit welchem Serviceagent kommuniziert werden soll und welche initialen Attribute dieser besitzt. Ein Spezialfall hierbei ist, dass der Aufrufer nur eine Serviceinstanz des Services kennt (dies wird durch eine 1-Multiplizität auf dem Kommunikationspfad im Deploymentdiagramm modelliert), oder dass es nur eine Serviceinstanz des Services gibt (dies wird durch den Stereotyp ≪singleton≫ modelliert). In diesem Fall kennt der Aufrufer die Serviceadresse initial. Dies wird formalisiert, indem für jeden Ausgangsport eines Aufrufers, für den dieser Spezialfall gilt, eine dynamische Funktion spezifiziert wird, die abhängig vom Aufruferagenten eine Serviceadresse zurückgibt.

Dies ist der Fall bei der ETicket-Fallstudie in Abschnitt 4.2.4. Der *ETicketService* besitzt den Stereotyp ≪singleton≫. Somit ist für das *UserDevice* (Serviceaufrufer) die folgende Funktion notwendig:

```
UserDevice-UserDevice2ETicketService_default-Address :
(agent  →  string);
```

UserDevice2ETicketService_ default beschreibt den Standard-Port, der automatisch generiert wird, falls keiner modelliert ist. Somit hat jeder Agent, der eine solche Serviceinstanz aufruft Zugriff auf die Adresse der Serviceinstanz und erhält auf dieselbe Weise wie beschrieben (siehe Listing 7.1) einen Serviceagenten zugewiesen. Zum Versenden von Nachrichten wird anschließend eine Verbindung zum Empfänger mittels *CONNECTTOSERVICE* aufgebaut und mit der *SEND*-Prozedur (siehe Listing 9.7 im Anhang)), wie sie auch bei Chipkarten verwendet

[2] *serviceClassName* ist ein Aufzählungstyp von allen Serviceklassen

wird, die Nachricht übertragen. Wenn eine Verbindung zu einem Serviceagenten jedoch schon besteht, wird zum Senden einer weiteren Nachricht nur noch die *SEND*-Prozedur benötigt. Ebenso erfolgt die Rückantwort nur über die *SEND*-Prozedur, da eine Verbindung schon besteht. Der Verbindungsaufbau wird im nächsten Abschnitt detailliert erläutert.

7.2.3 Verbindungsaufbau und -abbau

Kaskadierende Serviceaufrufe sind mit der bisherigen Formalisierung wie sie bei Chipkarten verwendet wird nicht möglich. Bevor eine Nachricht an eine Chipkarte übertragen werden kann, muss eine Verbindung aufgebaut werden. Eine Verbindung (Connection) besteht aus zwei Endpunkten (Endpoints). Ein Endpunkt besteht wiederum aus einem Agent und einem Port[3], und darf zur gleichen Zeit nur an einer Verbindung teilnehmen. Durch eine solche Verbindung wird der Empfänger der gesendeten Nachricht sowie der Rückantwort jeweils anhand des Aufrufers und dessen Port eindeutig bestimmt. Bei einer Nachrichtenübertragung wird anhand des im entsprechenden Endpunkt gespeicherten Agenten die Nachricht in dessen Inbox gelegt. Solange einer der beiden Kommunikationsteilnehmer noch eine Nachricht in seiner Inbox besitzt, kann eine Verbindung nicht beendet werden. Dadurch wird sichergestellt, dass eine Rückantwort, ohne die Interaktion eines Angreifers, wieder beim selben Absender ankommt. Der Verbindungsaufbau und -abbau ist jedoch nichtdeterministisch formalisiert. Bei einem kaskadierenden Serviceaufruf könnte nach dem Verarbeiten der empfangenen Nachricht und dem Aufruf des nächsten Services die Verbindung zum ursprünglichen Aufrufer nichtdeterministisch abgebaut und stattdessen ebenfalls nichtdeterministisch eine Verbindung zu einem anderen Service wieder aufgebaut werden. Hierdurch würde eine spätere Rückantwort des betrachteten Services nicht an den ursprünglichen Aufrufer übertragen. Somit könnten z.B. sensible Daten an den falschen Empfänger geraten ohne das Mitwirken eines Angreifers. Dies ist in der realen Anwendung jedoch nicht möglich.

Dies hat zur Folge, dass der Kommunikationsmechanismus verändert werden musste. Ein wichtiger Bestandteil hierbei ist der Verbindungsaufbau zu Services. Dieser wird nun nicht mehr nichtdeterministisch aufgebaut, sondern vom Aufrufer. Dadurch wird bei jedem Verbindungsaufbau eine neue Sitzungsinstanz erstellt. Dies entspricht auch dem Verhalten des generierten Codes (siehe Listing 7.1).

```
1 procedures
2  CONNECTTOSERVICE agent × ports × agent × port : connectionset,
3  (agent → ports → messagelist) nonfunctional indeterministic;
4
5 declaration
6  CONNECTTOSERVICE
7  : CONNECTTOSERVICE (ag, outport, ag0; connections, inputs)
8        {
```

[3]Ein Port beschreibt ein Ende eines in SecureML modellierten Kommunikationspfades.

```
9   ...
10  choose conn
11   with connected(ag ⊙ outport ,connections)
12   in {
13    connections := connections -- conn ;
14    inputs(ag)(outport) := [] ;
15   } ifnone skip
16  ...
17  choose conn
18   with serviceConn-ok(conn) ∧
19     conn.endpoint1.agent = ag ∧
20     conn.endpoint1.port = outport ∧
21     conn.endpoint2.agent = ag0
22   in {
23     connections := connections ++ conn ;
24   } ifnone skip
25  };
```

Listing 7.2: Spezifikation von CONNECTTOSERVICE

Listing 7.2 beschreibt die Prozedur *CONNECTTOSERVICE* die den Verbindungsaufbau durch einen Aufrufer formalisiert. Hierbei handelt es sich um eine gekürzte Version, da diese Prozedur auch zum Verbindungsaufbau durch einen Angreifer verwendet wird. Der Verbindungsaufbau durch einen Angreifer sowie die fehlenden Zeilen werden im Abschnitt 7.3 betrachtet. Als Parameter wird der Aufrufer *ag*, der Aufruferport *outport* über den die Nachricht verschickt werden soll, der Empfänger *ag0* sowie alle Verbindungen (*connections*) und alle Inboxen (*inputs*) übergeben (Zeile 7). Der Eingangsport des Empfängers muss nicht übergeben werden, da er durch die drei ersten Argumente eindeutig bestimmbar ist. Zuerst wird überprüft ob der Aufrufer an dem Ausgangsport *outport* eine offene Verbindung besitzt (Zeile 11). Ist dies der Fall wird diese geschlossen (Zeile 13) und die Inbox (*inputs*) für den Aufrufer an diesem Ausgangsport geleert (Zeile 14). Das Schließen einer existierenden Verbindung ist wichtig, da jeder Agent zur gleichen Zeit nur eine Verbindung am selben Port besitzen darf, damit eine Nachrichtenübertragung über eine Verbindung eindeutig anhand des Aufrufers und dessen Port stattfinden kann. Dies geschieht auch im realen Code, da eine neue Stubinstanz erstellt wird und die Alte verworfen. Dadurch kann der Aufrufer auch im generierten Code keine Nachrichten mehr über diese Verbindung empfangen und bearbeiten. Deshalb werden alle bis dahin nicht bearbeiteten Nachrichten vom Aufrufer verworfen. Die Sitzungsinstanz am anderen Ende der Verbindung erfährt davon erstmal nichts bis ein Timeout stattfindet. Deshalb darf die Inbox des Empfängers nicht gelöscht werden. Anschließend wird eine Verbindung gewählt, die das Prädikat *serviceConn-ok* erfüllt, am einen Ende der Verbindung der Aufrufer *ag* mit seinem Ausgangsport *outport* sitzt (Zeile 19-20) und am anderen Ende der Agent *ag0* (Zeile 21). Falls eine solche Verbindung möglich ist, wird diese zu der Menge aller Verbindungen hinzugefügt (Zeile 23).

```
1  predicates
2   serviceConn-ok : connection;
3
4  axioms
5   serviceConn-ok :
6   serviceConn-ok(conn) ↔
7    serviceEndpoint-ok(conn.endpoint2) ∧
8    (
9     (conn.endpoint1.port = AccountOwnerPC2Bank_default ∧
10     (conn.endpoint2.port = Bank2AccountOwnerPCDefaultPort ∨
11     conn.endpoint2.port = AttackerPort)) ∨
12    (conn.endpoint1.port = Bank2Bank_default ∧
13     (conn.endpoint2.port = Bank2BankDefaultPort ∨
14     conn.endpoint2.port = AttackerPort)) ∨
15    ...
16   );
```

Listing 7.3: Spezifikation von serviceConn-ok

Listing 7.3 stellt das Prädikat *serviceConn-ok* anhand der Onlinebanking-Fallstudie dar. Es überprüft ob es sich um eine gültige Serviceverbindung handelt. Hierzu wird überprüft, dass der Empfänger ein Service ist bzw. der Angreifer, der sich als Service ausgibt, und dass die Ports den durchs Anwendungsmodell definierten Kommunikationspfaden entsprechen. Als erstes wird anhand dem Prädikat *serviceEndpoint-ok* überprüft ob der Endpunkt des Services, der aufgerufen werden soll, gültig ist. Der Endpunkt des Aufrufers (*conn.endpoint1*) muss nicht überprüft werden, da die Serviceklasse-Port-Kombination beim Aufruf nicht gewählt wird, sondern statisch bekannt ist (siehe Zeile 3 und 11 in Listing 7.1). Anschließend wird überprüft ob es sich um eine gültige (modellierte) Portkombination zu einem Service bzw. zu einem Angreifer (nicht modelliert), der sich als Services ausgibt, handelt (Zeile 9-14).

```
1  predicates
2   serviceEndpoint-ok    : endpoint;
3
4  axioms
5   serviceEndpoint-ok-def :
6   serviceEndpoint-ok(endp) ↔
7   (
8    (isBank(endp.agent) ∧ endp.agent.name < next-BankAgentName ∧
9     (endp.port = Bank2AccountOwnerPCDefaultPort ∨
10    endp.port = Bank2BankDefaultPort)) ∨
11   (isAttacker(endp.agent) ∧ (endp.port = AttackerPort))
12  );
```

Listing 7.4: Spezifikation von serviceEndpoint-ok

Das Prädikat *serviceEndpointok* in Listing 7.4 beschreibt, dass bei einem gültigen Endpunkt eines aufzurufenden Serviceagenten in der Onlinebanking-Fallstudie

entweder der Agent vom Typ *Bank* sein muss, schon erstellt wurde und einen der zwei modellierten Einganspors der Serviceklasse *Bank* besitzt (Zeile 8-10) oder der Angreifer mit dem Port *AttackerPort* ist (Zeile 11).

7.2.4 Sitzungsverwaltung

Durch die Verwendung von stateful Web Services wird für jeden Aufrufer eine Sitzungsinstanz erstellt. Diese Sitzungsinstanz kann Daten wie z.B. einen Sitzungsschlüssel als Instanzattribut speichern. Somit wird die Sitzungsverwaltung durch stateful Web Services übernommen. Die Gültigkeit einer Sitzung wird mit ≪openSession≫ und ≪closeSession≫ modelliert. Der Aufbau einer Sitzung (≪openSession≫) erfolgt durch den Aufbau einer Verbindung wie in Listing 7.2 gezeigt. Solange diese Verbindung besteht können Nachrichten über diese in beide Richtungen übertragen werden.

Gemeinsame Attribute

In SecureML wird der Zugriff auf gemeinsam genutzte Attribute zwischen den Sitzungsinstanzen einer Serviceinstanz unterstützt. Dies wird modelliert indem der Stereotyp ≪static≫ auf die Attribut einer Serviceklasse annotiert wird. Im generierten Code wird dies durch ein statisches Attribut des Servicemanagers realisiert. Die Formalisierung erfolgt durch dynamische Funktionen, welche jedoch nicht anhand eines Agenten ein Attributwert zurückgeben, sondern anhand der Serviceadresse.

```
Shared-'Servicename'-'Attributname' :
                    (string  →  'Attributtyp');
```

Die Serviceadresse, die beim Aufrufen eines Services verwendet wird, ist für alle Sitzungsinstanzen einer Serviceinstanz gleich. Durch die dynamische Funktion *'Servicename'Address : (agent → string)* hat jede Sitzungsinstanz Zugriff auf seine Serviceadresse und kann somit auf das gemeinsam genutzte Attribut zugreifen. Falls innerhalb eines Protokollschritts auf ein solches Attribut lesend und schreibend zugegriffen wird, dann muss jeder Bereich im generierten Code, indem ein Zugriff auf dieses Attribut stattfindet, anhand dem statischen Attribut gelockt werden (siehe 5.1.4). Dadurch wird sichergestellt, dass ein Element, welches innerhalb eines Protokollschritts gelesen, geändert und wieder zurückgeschrieben wurde, währenddessen nicht bereits geändert wurde. Falls auf ein gemeinsam genutztes Attribut in einem Protokollschritt lesend zugegriffen wird und in einem anderen schreibend, dann werden verschränkte Abfolge der Zugriffe durch die nichtdeterministische Auswahl der Protokollschritte betrachtet. Solche gemeinsamen Attribute wurden z.B. in der ETicket-Anwendung verwendet, damit eine Liste der ausgestellten Tickets unabhängig von einer Sitzung gespeichert werden können und bei der Onlinebanking-Anwendung um die Konten zu verwalten.

7.2.5 Transport Layer Security

Transport Layer Security (TLS) ist eines der bekanntesten Protokolle zum sicheren Übertragen von Daten. Im Web Service Bereich wird TLS häufig verwendet um die Kommunikation mit Services zu schützen. Damit beim Entwickeln von sicheren Anwendungen, TLS nicht jedes mal neu modelliert werden muss, ist TLS fester Bestandteil von SecureMDD. Dessen Verwendung kann durch die Annotation des Stereotypen ≪TLS≫ auf einen Kommunikationspfad spezifiziert werden. Damit TLS jedoch beim Beweisen der Sicherheitseigenschaften einer neuen Anwendung nicht jedes mal neu verifiziert werden muss, was die Verifikation erheblich erschweren würde, wird TLS auf die Fähigkeiten des Angreifers abgebildet. Wenn ein Angreifer auf einem Kommunikationskanal die Fähigkeit hat Nachrichten mitzulesen (*read*), zu versenden (*send*) und zu unterdrücken (*suppress*), dann handelt es sich um einen Dolev-Yao Angreifer [31] auf diesem Kommunikationskanal. Falls auf diesem Kommunikationskanal die Kommunikationsteilnehmer TLS verwenden um ihren Nachrichtenverkehr zu schützen, dann verliert der Angreifer einige seiner Fähigkeiten. Da TLS als ein sicheres Protokoll angenommen wird, können einige seiner Sicherheitseigenschaften [30] verwendet werden. TLS beginnt mit einem Handshake, wodurch einer oder beide Kommunikationspartner sich authentifizieren. Die Nachrichten werden mit einem Sitzungsschlüssel verschlüsselt. Die Integrität der Nachrichten wird durch einen Message Authentication Code (MAC) sichergestellt und durch eine Sequenznummer werden verlorene oder schon empfangene Nachrichten erkannt. Ein erkannter Fehler führt zum Verbindungsabbruch.

Gegenseitige Authentifizierung

Wir nehmen an, dass ein Angreifer nicht in der Lage ist ein gültiges TLS-Zertifikat zu erhalten, welches von einer anderen Komponente akzeptiert wird. Dies bedeutet, dass der Angreifer keine TLS-gesicherte Kommunikation initiieren kann, falls gegenseitige Authentifizierung verwendet wird. In diesem Fall sind seine Fähigkeiten bezüglich einer Kommunikation zwischen zwei Komponenten eingeschränkt. Der Angreifer kann somit nur verschlüsselte Nachrichten lesen. Der verwendete Schlüssel ist ein Sitzungsschlüssel, der während der Authentifizierung ausgetauscht wurde und in keiner anderen Sitzung verwendet wird. Deswegen ist das Lesen der Nachrichten für den Angreifer sinnlos, da er die Nachrichten nicht entschlüsseln kann und das Wiedereinspielen von Nachrichten (Replay) ist wegen der Sequenznummern ebenfalls nicht möglich. Zudem werden nur logische Sicherheitseigenschaften betrachtet und keine Datenverkehrsanalysen, wo die Nachrichtenlänge oder die Zeit wichtig sein könnte. Des Weiteren verliert der Angreifer im formalen Modell seine Fähigkeit Nachrichten zu versenden. Wenn die Nachricht nicht mit dem richtigen Sitzungsschlüssel authentifiziert ist (den der Angreifer nicht besitzt), wird die MAC-Überprüfung fehlschlagen und die Nachricht wird nicht akzeptiert. Eine Nachricht, die abgefangen wurde ist mit dem richtigen Sitzungsschlüssel verschlüs-

selt, jedoch wird ein Wiedereinspielen anhand der Sequenznummer erkannt und somit die Nachricht nicht akzeptiert. Die Fähigkeit, Nachrichten zu unterdrücken geht ebenfalls verloren, da die nächste Nachricht eine für den Empfänger ungültige Sequenznummer besitzt. Jedoch besitzt der Angreifer die Fähigkeit eine Sitzung zu beenden, indem er eine alte Nachricht wiedereinspielt, da ein Fehler in einer TLS-Sitzung zum Abbruch der Sitzung führt. Somit kann eine TLS gesicherte Verbindung mit gegenseitiger Authentifizierung durch einen Angreifer formalisiert werden, der nur die Verbindung unterbrechen kann. Da eine TLS-Verbindung durch die schon vorgestellten Verbindungen realisiert ist, kann eine TLS-Verbindung sowieso nichtdeterministisch und somit nach jedem Protokollschritt abgebrochen werden.

Serverseitige Authentifizierung

Bei serverseitiger Authentifizierung kann der Angreifer wie auch bei gegenseitiger Authentifizierung die übertragenen Nachrichten zwischen einem Client und einem Service nicht lesen. Jedoch kann der Angreifer eine eigene TLS-gesicherte Kommunikation initiieren. Hierdurch hat der Angreifer Zugriff auf die übertragenen Klartextdaten, da er sie beim Senden erstellt und beim Empfangen mit dem vorher ausgetauschten Sitzungsschlüssel entschlüsselt. Das Unterdrücken von Nachrichten ist in diesem Fall sinnlos, da nur der Angreifer die Nachrichten versendet und somit selbst entscheiden kann, welche Nachrichten übertragen werden. Natürlich besitzt der Angreifer die Fähigkeit seine eigene Verbindung zum Service zu trennen. Eine mit TLS gesicherte Verbindung mit serverseitiger Authentifizierung könnte somit auf einen Angreifer abgebildet werden der fremde Verbindung nur unterbrechen kann, jedoch in der Lage ist eine eigene Verbindung zum Service aufzubauen und über diese Verbindung Nachrichten zu senden und zu empfangen sowie die Verbindung zu unterbrechen.

Services sind durch stateful Web Services realisiert und erzeugen für jeden Aufrufer eine Sitzungsinstanz. Da die Adresse einer Sitzungsinstanz jedoch auch über eine TLS-Verbindung übertragen wird, kann der Angreifer diese für andere Nutzer nicht erfahren. Somit kann der Angreifer nur mit einer für ihn generierten Sitzungsinstanz interagieren sowie fremde Verbindungen unterbrechen. Der Verbindungsabbruch einer fremden Verbindung ist auch hier durch den nichtdeterministischen Verbindungsabbau spezifiziert und der Verbindungsaufbau durch den Angreifer zu einem neuen Serviceagenten wird im folgenden Abschnitt beschrieben.

7.2.6 Allgemeine Erweiterungen

Wiederholende Ereignisse

In der Onlinebanking-Fallstudie wurde ein Protokoll modelliert, welches sicherstellt, dass unvollständig Transaktionen zu einem späteren Zeitpunkt noch einmal ausgeführt werden (siehe Kapitel 4.2.3). Dies wurde durch ein Recovery-Protokoll sichergestellt, welches alle fünf Minuten automatisch aufgerufen wird. Hierzu wur-

de SecureML um wiederholende Ereignisse erweitert (siehe Kapitel 4.2.2), welche durch Zeitevents modelliert werden können. Somit gibt es Protokollschritte, die nicht durch den Empfang einer Nachricht angestoßen werden, sondern durch ein Zeitevent. Das Recovery-Protokoll besteht aus Protokollschritten die genauso wie die anderen Protokollschritte in das formale Modell transformiert werden. Der Unterschied hierbei ist jedoch, dass der Protokollschritt, der das Zeitevent enthält keine Nachrichten aus der Inbox des Agenten verarbeitet, sondern nichtdeterministisch und unabhängig von eingehenden Nachrichten nach jedem Schritt (STEP) der ASM ausgeführt werden kann. Dies ist eine Überapproximation, wodurch das Recovery-Protokoll im formalen Modell öfters ausgeführt wird als im Realen. Wenn Sicherheitseigenschaften anhand dieser Überapproximation bewiesen wurden gelten sie jedoch für ein beliebiges Zeitintervall und somit auch für die den generierten Code, der alle fünf Minuten auf unvollständige Transaktionen überprüft.

Nachrichtenreihenfolge

Innerhalb eines Aktivitätsdiagramms wird ein valider Protokollablauf modelliert, welcher eine graphische Repräsentation der Nachrichtenreihenfolge enthält. Diese Reihenfolge wurde bisher ignoriert und alle Protokollschritte konnten unabhängig von dieser graphischen Reihenfolge aufgerufen werden. Durch die optionale Eigenschaft *fixMessageOrder* des Stereotypen ≪Message≫ (siehe Abschnitt 4.1) ist es möglich, dass diese Reihenfolge im Programmcode als auch im formalen Modell sichergestellt wird. Dies geschieht, indem das plattformunabhängige Modell durch das dynamische Setzen und Überprüfen von Zuständen automatisch erweitert wird (siehe 5.1.7). Zudem wird bei jedem Protokollabbruch der generierte Zustand zurückgesetzt sowie eine optionale Ausnahmebehandlung, die durch den Stereotyp ≪Exception≫ modelliert wurde, aufgerufen. Durch das gleiche Vorgehen wird für die Protokollschritte innerhalb einer Serviceoperationen eine sequentielle Abarbeitung im formalen Modell erreicht.

*-Listen und Key-Value-Listen

Listen fester Länge wurden schon in [62] formalisiert. Der einzige Unterschied bei der Formalisierung von *-Listen ist, dass die *hasFree*-Operation immer wahr ist. Key-Value-Listen besitzen jedoch neue Operationen und mussten neu formalisiert werden. Deren Implementierung ist in Abschnitt 5.1.7 beschrieben.

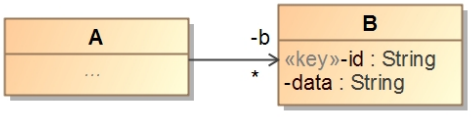

Abbildung 7.2: Klasse A besitzt eine Key-Value-Liste

Abbildung 7.2 zeigt beispielhaft die Modellierung einer Key-Value-Liste mit SecureML (siehe Abschnitt 4.2.1). Eine Key-Value-Liste wird durch eine gerichtete Assoziation mit einer Multiplizität größer 1 und dem Stereotypen ≪key≫ auf einem Klassenattribut, des Value-Elements modelliert. Der Typ der Value-Elemente entspricht der Klasse auf den die Assoziation zeigt (B) und die Liste hat den Name *listOf* + Klassenname. Bezogen auf das gezeigte Beispiel heißt die Key-Value-Liste *listOfB*.

```
1  functions
2    removeKey    : listofB × ID → listofB;
3    get          : listofB × ID → B;
4    set          : listofB × B → listofB;
5
6  predicates
7    containsKey  : listofB × ID;
8
9  axioms
10   removeKey-empty :
11     removeKey([], id) = [];
12   removeKey-found :
13     id = b.id → removeKey(b ' + listofb, id) = removeKey(listofb, id);
14   removeKey-rec:
15     not id = b.id →
16       removeKey(b ' + listofb, id) = b ' + removeKey(listofb, id);
17
18   containsKey-empty :
19     not containsKey([], id);
20   containsKey-found :
21     id = b.id → containsKey(b ' + listofb, id);
22   containsKey-rec :
23     not id = b.id → (containsKey(b ' + listofb, id) ↔
24               containsKey(listofb, id) );
25
26   get-found :
27     id = b.id → get(b ' + listofb, id) = b;
28   get-rec :
29     not id = b.id → get(b ' + listofb, id) = get(listofb, id);
30
31   set-empty :
32     set([], b) = b ';
33   set-found :
34     b.id = b0.id → set(b0 ' + listofb, b) = listofb + b ';
35   set-rec :
36     not b.id = b0.id → set(b0 ' + listofb, b) = b0 ' + set(listofb, b);
```

Listing 7.5: Key-Value-Liste

Listing 7.5 zeigt die Spezifikation der Operationen der modellierten Key-Value-Liste. Die Datenstruktur einer Key-Value-Liste ist als Liste spezifiziert. Jedoch gibt es spezielle Operationen. *removeKey* (löscht ein Element anhand eines Keys), *get* (gibt ein Element anhand einem Key zurück) und *set* (setzt ein Element) werden als Funktionen spezifiziert und *containsKey* als Prädikat. Die Axiomatisierung erfolgt in allen vier Fällen rekursiv. Hierzu wird die Liste strukturell durchlaufen, indem sie in das erste Element und den Rest aufgeteilt wird. Falls beim Entfernen eines Elements durch die Funktion *removeKey* die übergebene *id* mit der *id* in dem ersten Element nicht übereinstimmt, wird *removeKey(b ' + listofb , id)*

durch *b ' + removeKey(listofb, id)* ersetzt. Durch *b '* wird aus einem Element eine Liste, welche nur dieses Element enthält. Falls die übergebene *id* mit der *id* im ersten Element übereinstimmt, wird *removeKey(b ' + listofb , id)* durch *removeKey(listofb, id)* ersetzt und somit das Element entfernt. Falls die übergebene Liste leer ist, wird die leere Liste zurückgegeben. Somit wird aus allen Element deren ID ungleich der übergebenen ID sind, eine neue Liste erstellt und diese zurückgegeben. Vor dem Aufruf der *get*-Funktion innerhalb eines Protokollschritts wird mit dem *containsKey*-Prädikat überprüft, ob ein Element mit dem selben Key schon vorhanden ist. Falls dies nicht der Fall ist, wird der Protokollablauf beendet.

Zusätzlich zu den vier gezeigten Operationen können auch die Listenoperationen *size*, *at*, *remove*, *contains* und *hasFree* verwendet werden. Die *add*-Operation darf hierbei nicht verwendet werden, da hierdurch zwei Elemente mit der gleichen ID hinzugefügt werden könnten.

Garantierte Nachrichtenzustellung

Obwohl ein Angreifer keine Fähigkeiten besitzt bzw. diese ihm wegen TLS mit gegenseitiger Authentifizierung genommen wurden, können Nachrichten wegen einem Verbindungsabbruch trotzdem verloren gehen. Durch den Stereotypen ≪reliable≫ wird davon ausgegangen, dass dies wegen einer Standleitung oder dem Einsatz eines "Guaranteed Delivery"-Systems[4] nicht möglich ist. Somit ist auf solchen Leitungen selbst der nichtdeterministische Verbindungsabbruch nicht möglich.

Unsichere Terminals

Terminals mit der Eigenschaft *insecure* beschreiben unsichere Benutzergeräte, deren interner Speicher vom Angreifer ausgelesen und manipuliert werden kann. Solche Geräte sollten keine geheimen Informationen besitzen, da sie der Angreifer erfährt. Somit ist auch die Modellierung von kryptographisch sicheren Verbindungen zu einem solchen Gerät nicht sinnvoll, da der Angreifer alles mitlesen und manipulieren kann. Aus diesem Grund wurde die Modellierung eines solchen unsicheren Gerätes auf das Durchreichen von Nachrichten eingeschränkt. Ein solches unsicheres Terminal, welches nur Nachrichten weiterleitet, kann formalisiert werden, indem der Angreifer auf die eingehenden und ausgehenden Kommunikationspfade volle Angreiferfähigkeiten (*read, send, suppress*) erhält. Ein solches unsicheres Terminal kommt z.B. bei der ETicket-Fallstudie zum Einsatz, um Ende-zu-Ende verschlüsselte Nachrichten zwischen Chipkarte und Service durchzureichen.

7.3 Angreifer

Der Angreifer wurde teilweise schon beim Serviceaufruf und Verbindungsaufbau in Abschnitt 7.2.2 und 7.2.3 betrachtet. Da der Verbindungsaufbau zu Services durch

[4] http://camel.apache.org/guaranteed-delivery.html

die Aufrufer stattfindet und da durch den Einsatz von TLS der Angreifer neue Fähigkeiten erhält, muss der Angreifer in bestimmten Fällen auch in der Lage sein selbst Verbindungen aufzubauen. Hierbei wird zwischen einem Verbindungsaufbau zu einem verwendeten Serviceagenten und zu einem neuen Serviceagenten unterschieden. Zudem wird in diesem Abschnitt das Senden und Lesen von Nachrichten durch den Angreifer betrachtet sowie die Manipulation der Serviceadressen über unsichere Verbindungen.

7.3.1 Verbindungsaufbau zu einem verwendeten Serviceagenten

Ein realer Angreifer mit den Fähigkeiten *read, send, suppress* auf einer Leitung kann zu jeder Zeit Nachrichten senden, lesen und unterdrücken. Um Nachrichten senden zu können muss eine Verbindung aufgebaut worden sein. Falls keine Verbindung vorhanden ist, kann der reale Angreifer eine eigene Verbindung aufbauen. Da der Verbindungsaufbau zu einem Service durch einen modellierten Aufrufer beim Öffnen einer Sitzung erfolgt und nicht wie bei Chipkarten nichtdeterministisch, erhält der Angreifer im formalen Modell auf solchen Leitungen die Fähigkeit eigene Verbindungen aufzubauen. Somit kann der Angreifer entweder über eine bestehende oder über eine eigene Verbindung mit einem Serviceagenten interagieren. Solche Verbindungen können wie alle Verbindungen wieder nichtdeterministisch abgebaut werden. Ein realer Angreifer mit den Fähigkeiten *read, send, suppress* kann jedoch eine Verbindung zu einer Sitzungsinstanz aufbauen, welche schon zu einer Verbindung gehört, da der Angreifer deren Adresse durch die Fähigkeit *read* erfahren konnte. Im formalen Modell kann in diesem Fall von einem Verbindungsaufbau durch den Angreifer abstrahiert werden, da wenn eine Verbindung schon besteht, dann kann ein solcher Angreifer auf die bestehende Verbindung zugreifen und muss keine eigene aufbauen. Falls keine Verbindung zur einer existierenden Sitzungsinstanz mehr besteht, hat der Angreifer, mit den Fähigkeiten *read, send, suppress* auf einer Leitung zu einem Service, auch im formalen Modell die Fähigkeit eine eigene Verbindung aufzubauen. Hierzu wird die Spezifikation des Angreifers um die Prozedur *ATTACKER-CONNECT-RSS* erweitert (siehe Listing 7.6).

```
1  procedures
2    ATTACKER-CONNECT-RSS : connectionset
3              nonfunctional indeterministic;
4
5
6  declaration
7    ATTACKER-CONNECT-RSS
8      : ATTACKER-CONNECT-RSS (; connections)
9          {
10         choose conn with
11           eavesdrop(conn) ∧ suppress(conn) ∧ send(conn) ∧
12           isAttacker(conn.endpoint1.agent) ∧
13           conn.endpoint1.port = AttackerPort ∧
14           serviceConn-ok(conn) ∧
15           not connected(conn.endpoint2,connections)
16                  in {
17                        connections := connections ++ conn ;
18                     } ifnone skip
```

```
19          }
```

Listing 7.6: Spezifikation von *ATTACKER-CONNECT-RSS*

Der Angreifer wählt eine Verbindung auf der er die Fähigkeiten *read, send, suppress* besitzt (Zeile 11), er der Aufrufer mit seinem Angreiferport ist (Zeile 12-13) und die Verbindung das Prädikat *serviceConn-ok* erfüllt (Zeile 14) sowie der Endpunkt am anderen Ende der Leitung an keiner Verbindung teilnimmt (Zeile 15). Falls eine solche Verbindung existiert wird diese zu der Menge alle Verbindungen hinzugefügt. Das Prädikat *serviceConn-ok* wurde schon beim Aufbauen einer Verbindung durch einen modellierten Systemteilnehmer gezeigt (siehe Listing 7.3) und stellt sicher, dass sich am anderen Ende der Verbindung ein valider Endpunkt eines Services befindet und das die Portkombination valide ist. Damit auch ein Angreifer als Aufrufer unterstützt wird, musste die Spezifikation des in Listing 7.3 gezeigten *serviceConn-ok* Prädikates um die folgende (nicht modellierte) Portkombination vom Angreifer zum Service erweitert werden.

$$
\begin{aligned}
&(\text{conn.endpoint1.port} = \text{AttackerPort} \wedge \\
&\quad (\text{conn.endpoint2.port} = \text{Bank2AccountOwnerPCDefaultPort} \vee \\
&\quad \text{conn.endpoint2.port} = \text{Bank2BankDefaultPort}))
\end{aligned}
$$

Durch die Portüberprüfung in *serviceConn-ok* kann der Angreifer keine Verbindung zu sich selbst erstellen und in *serviceEndpoint-ok* wird sichergestellt, dass der Serviceagent schon verwendet wurde. Für die vollständige Spezifikation von *serviceConn-ok* siehe Listing 9.6 im Anhang.

7.3.2 Verbindungsaufbau zu einem neuen Serviceagenten

Ein Angreifer mit den Fähigkeiten *read, send, suppress* kann jedoch auch eine Verbindung zu einem noch nicht verwendeten Serviceagenten aufbauen. Dies ist z.B. dann wichtig, falls eine Sicherheitseigenschaft nur dann verletzt wird, wenn der Angreifer mit einem Serviceagenten einer Serviceinstanz kommuniziert bevor ein modellierter Aufrufer diese Serviceinstanz aufruft. Zudem kann ein Angreifer mit den Fähigkeiten *read, send, suppress* trotz TLS mit serverseitiger Authentifizierung ebenfalls eine eigene Verbindung zu einem noch nicht verwendeten Serviceagenten aufbauen. Aus diesen Gründen muss die Spezifikation des Angreifers um die Prozedur *ATTACKER-CONNECT-ToNewServiceAgent* erweitert werden.

```
1  procedures
2    ATTACKER_CONNECT_ToNewServiceAgent : connectionset
3              nonfunctional indeterministic;
4
5  declaration
6    ATTACKER_CONNECT_ToNewServiceAgent
7      : ATTACKER_CONNECT_ToNewServiceAgent (; connections)
8          {
9       choose address with connectByAttacker_possible(address)
10      in {
11        if serviceClassName(address) = Bank then {
```

```
12          let ag0 = Bank(next-BankAgentName) in {
13            BankAddress(ag0) := address;
14            Bank-verifyKey(ag0) := Bank-verifyKey(address);
15            ...
16            next-BankAgentName := next-BankAgentName + 1;
17            CONNECTTOSERVICE(ATTACKER, AttackerPort, ag0;
18                                                connections, inputs);
19          }
20        }
21      } ifnone skip
22    }
```

Listing 7.7: Spezifikation von *ATTACKER-CONNECT-ToNewServiceAgent* am Beispiel der Onlinebanking-Anwendung

Listing 7.7 zeigt die Prozedur *ATTACKER-CONNECT-ToNewServiceAgent* anhand der Onlinebanking-Anwendung. Diese kann nichtdeterministisch aufgerufen werden und erhält als Eingabe die Menge aller Verbindungen. Als erstes wird eine Adresse gewählt, die das Prädikat *connectByAttacker-possible* erfüllt (Zeile 9). Dabei wird geprüft, ob es sich bei der gewählten Adresse um eine valide Serviceadresse zu einem Service handelt, der aus einem unsicheren Netzwerk ohne TLS mit gegenseitiger Authentifizierung aufgerufen werden kann (siehe Listing 9.4 im Anhang). Anschließend wird eine Fallunterscheidung anhand des Servicenamens gemacht, damit ein Serviceagent vom richtigen Typ erstellt werden kann und die Zuweisungen wie z.B. die initialen Attribute für die richtige Serviceklasse stattfinden. Da in der Onlinebanking-Anwendung nur die Serviceklasse Bank vorkommt und diese TLS mit serverseitiger Authentifizierung verwendet und über ein unsicheres Netzwerk aufgerufen werden kann, gibt es keine weiteren Fälle. Ein Serviceagent wird auf dieselbe Weise erstellt wie beim Aufruf einer Serviceinstanz durch einen modellierten Aufrufer (siehe Listing 7.1 in Abschnitt 7.2.2). Es wird ein neuer und noch nie verwendeter Serviceagent der entsprechenden Serviceklasse anhand der Variable *next-BankAgentName* erstellt (Zeile 12). Anschließend wird dieser Serviceagent (*ag0*) einer Serviceinstanz mit der Adresse *address* zugewiesen (Zeile 13), die initialen Attribute werden zugewiesen (Zeile 14-15) und der Zähler *next-BankAgentName* wird erhöht (Zeile 16). Zum Schluss wird mittels der Prozedur *CONNECTTOSERVICE* (siehe Listing 7.2 in Abschnitt 7.2.3) eine Verbindung zwischen Angreifer und dem gerade definierten Serviceagenten erstellt (Zeile 17-18). Hierzu wird als Absender der Angreifer (*ATTACKER*) mit dem Port (*AttackerPort*) verwendet. Da der Aufrufer der Angreifer ist und dieser bei einer Nachrichtenübertragung nichtdeterministisch eine Verbindung (Endpunkt des Empfängers) auswählen darf, muss eine bestehende Verbindung nicht geschlossen werden. Deshalb wurde die Prozedur *CONNECTTOSERVICE* erweitert, indem bei einem Verbindungsaufbau durch den Angreifer seine bestehenden Verbindungen nicht geschlossen werden und seine Inbox auch nicht geleert wird. Für die vollständige Spezifikation von *CONNECTTOSERVICE* siehe Listing 9.5 im Anhang.

7.3.3 Senden und Lesen von Nachrichten

Welche Fähigkeiten ein Angreifer zwischen zwei Ports besitzt wird durch die folgenden drei Prädikate formalisiert:

```
send : connection;       (Nachrichten versenden)
eavesdrop : connection; (übertragenen Nachrichten lesen)
suppress : connection;  (übertragene Nachrichten
unterdrücken)
```

Für Verbindungen auf denen TLS angewendet wird, verliert der Angreifer die Fähigkeit Nachrichten zu lesen, zu unterdrücken und zu versenden. Somit dürfen TLS-Verbindungen bei diesen Prädikaten nicht berücksichtigt werden.

Das Übertragen einer Nachricht wird durch die Prozedur *ATTACKER-SEND*, die schon für Chipkartenanwendung formalisiert wurde, spezifiziert (siehe Listing 9.7 im Anhang) . Es wird eine Nachricht gewählt, welche aus dem Angreiferwissen erstellt werden kann, sowie ein Endpunkt, an den der Angreifer etwas senden kann. Hierzu muss eine Verbindung zu diesem Endpunkt existieren. Wenn eine solche Nachricht und ein solcher Endpunkt möglich sind, dann wird die Nachricht in die Inbox des Empfängers gelegt. Ob ein Angreifer etwas an einen Endpunkt senden kann wird durch das Prädikat *attacker-can-send* überprüft.

```
1  predicates
2    attacker-can-send : endpoint × connectionset;
3
4  axioms
5        attacker-can-send
6        : ⊢
7            attacker-can-send(endp, connections)
8          ↔ not isAttacker(endp.agent)
9            ∧ (ex conn.
10                 conn ∈ connections ∧ is-endpoint(endp, conn) ∧
11       (send(conn) ∨ isAttacker(other-endpoint(ends, conn).agent)))
12       ;
```

Listing 7.8: Spezifikation von attacker-can-send

Listing 7.8 spezifiziert das Prädikat *attacker-can-send*. Damit ein Angreifer nun Nachrichten versenden kann muss es eine Verbindung geben bei der der Empfänger nicht der Angreifer ist (Zeile 8-10) und der Angreifer auf dieser Verbindung Nachrichten versenden kann (*send*-Prädikat ist erfüllt) oder der Aufrufer der Angreifer ist (Zeile 11).

Das Lesen von übertragenen Nachrichten erfolgt durch die *SEND*-Prozedur, die bei modellierten Serviceaufrufen verwendet wird (siehe Abschnitt 7.2.2 sowie Listing 9.7 im Anhang). Falls der Angreifer auf der Verbindung über die eine Nachricht versendet wird lesen kann, dann wird die Nachricht zu seinem Angreiferwissen hinzugefügt. Ob ein Angreifer auf einer Verbindung Nachrichten lesen kann wird durch das Prädikat *attacker-can-read* überprüft.

```
1  predicates
2    attacker-can-read : connection;
```

```
3
4  axioms
5        attacker-can-read
6        : ⊢
7            attacker-can-read(conn)
8            ↔ eavesdrop(conn) ∨ is-endpoint(Attacker, conn)
9        ;
```

Listing 7.9: Spezifikation von attacker-can-read

Listing 7.9 spezifiziert das Prädikat *attacker-can-read*. Damit ein Angreifer Nachrichten lesen kann muss das *eavesdrop*-Prädikat erfüllt sein oder der Angreifer muss sich an einem Ende der Verbindung befinden.

Das Unterdrücken von Nachrichten sowie der nichtdeterministische Verbindungsabbau bleiben unverändert.

7.3.4 Manipulation der Adresse

Da ein Serviceaufruf anhand einer Adresse stattfindet und diese über eine unsichere Verbindung übertragen werden kann, kann sie auch vom Angreifer manipuliert werden. Dadurch kann ein Angreifer zusätzliche Fähigkeiten auf anderen Verbindungen erhalten, auf die der Angreifer keine direkten Fähigkeiten besitzt (da nicht modelliert bzw. TLS geschützt). Wenn der Angreifer seine eigene Adresse einschleust, dann wird ein Aufrufer trotz TLS den Angreifer aufrufen. Die übertragenen Zertifikate, die zur Authentifizierung bei TLS verwendet werden sind nämlich abhängig von der Adresse (Domänennamen) und der Angreifer kann für seine Adresse ein gültiges Zertifikat besitzen. Diese Fähigkeit des Angreifers wurde in Listing 7.1 im Abschnitt 7.2.2 bereits betrachtet. Die Manipulation der Serviceadresse durch den Angreifer kann zu schwerwiegenden Sicherheitslücken führen. In [32] wurde eine Sicherheitslücke im OAuth 2.0-Protokoll entdeckt, in der sich der Angreifer genau diese Fähigkeit zu nutze macht und somit auf die geheimen Ressourcen anderer Benutzer Zugriff erhält.

7.4 Sicherheitseigenschaften

In Abschnitt 4.3 wurde die Modellierung der Sicherheitseigenschaften durch OCL-Constraints vorgestellt. In diesem Abschnitt wird die Übersetzung dieser OCL-Constraints in das formale Modell betrachtet. Die geschieht automatisch anhand Transformationen, die Eclipse OCL[5] verwenden. Eclipse OCL ist eine Implementierung des OMG Standards Object Constraint Language (OCL)[6]. Die Implementierung besitzt einen OCL Interpreter, der OCL-Ausdrücke einließt, gegenüber einem UML-Modell validiert und anschließend in einem abstrakten Syntaxbaum (AST) speichert. Zum Transformieren eines OCL-Ausdrucks wird der abstrakte Syntaxbaum mit der Visitor API traversiert, verändert und anschließend zu einer gültigen

[5] https://projects.eclipse.org/projects/modeling.mdt.ocl
[6] http://www.omg.org/spec/OCL/

Spezifikation des formalen Modells zusammengefügt. Im Folgenden wird die Übersetzung von OCL-Constraints ins formale Modell anhand der ETicket-Fallstudie erläutert.

```
ETicketServer.allInstances()->
  collect(temp1| temp1.issuedTicketIDs)->
  forAll(id|
    isTicketReceivedByCard(id) or
    isTicketReadyForRetrieve(id)
  )
```

Listing 7.10: Erweiterte Darstellung der OCL-Constraint in Listing 4.3

Listing 7.10 zeigt die erweiterte Darstellung der OCL-Constraint in Listing 4.3, die die Sicherheitseigenschaft *Ausgestellte Tickets können aufgrund von Unterbrechungen nicht verloren gehen* definiert. OCL unterstützt eine Kurzschreibweise um auf Eigenschaften (z.B. Attribute, Assotiationen oder Operationen) für jedes Element einer Collection zuzugreifen. Dies wird definiert durch den Punkt-Operator auf eine Collection gefolgt von der Eigenschaft. Diese Kurzschreibweise wird durch das Eclipse OCL-Framework durch eine *collect*-Operation ersetzt.

```
1  predicates
2  paidTicketCanNotBeLost : (agent → listofID) × (agent → listofTicket) ×
3              (agent → listofID) × (agent → listofCardTicket);
4
5  axioms
6  paidTicketCanNotBeLost :
7  ⊢  paidTicketCanNotBeLost(
8       ETicketService-issuedTicketIDs,
9       ETicketCard-tickets,
10      ETicketCard-deletedTicketIDs,
11      ETicketService-tickets
12    )
13    ↔
14    (forAll1(ETicketCard-tickets,
15          ETicketCard-deletedTicketIDs,
16          ETicketService-tickets,
17          collect18(ETicketService-issuedTicketIDs,
18                ETicketServiceAllInstances)
19        )
20    );
```

Listing 7.11: Algebraische Spezifikation der OCL-Constraint in Listing 7.10

Listing 7.11 zeigt das Prädikat *paidTicketCanNotBeLost* und seine Axiomatisierung. Dieses Prädikat repräsentiert die erwähnte OCL-Constraint. Ein Prädikat ist eine Funktion, mit dem Rückgabetyp Boolean. *paidTicketCanNotBeLost* benötigt vier Parameter, *ETicketService-issuedTicketIDs*, *ETicketCard-tickets*, *ETicketCard-deletedTicketIDs* und *ETicketService-tickets*. Hierbei handelt es sich um dynamische Funktionen, die einen Agenten auf eine Liste von IDs bzw. Tickets abbilden. *ETicketService-issuedTicketIDs* speichert die vom *ETicketService* ausgestellten Ticket-IDs, *ETicketCard-deletedTicketIDs* speichert die von der Karte gelöschten Tickets, *ETicketCard-tickets* enthält die auf der Karte gespeicherten Tickets und *ETicketService-tickets* enthält die Tickets die vom Service solange

aufbewahrt werden bis sie abgeholt und auf einer Karte gespeichert werden. Diese dynamischen Funktionen sind initial leer und werden während der Ausführung der Protokolle gefüllt. *paidTicketCanNotBeLost* wird durch das Prädikat *forAll1* spezifiziert.

forAll1 erhält vier Parameter *ETicketCard-tickets*, *ETicketCard-deletedTicket IDs*, *ETicketService-tickets* und das Ergebnis der Funktion *collect1*. Die ersten drei Parameter werden für die selbstddefinierten OCL-Operationen *isTicketReceived-ByCard* und *isTicketReadyForRetrieve* benötigt. Die verwendeten Prädikate wie *forAll1* und *collect1* sind nummeriert, da diese Funktionen im Gegensatz zu den OCL-Operationen *forAll* und *collect* nicht generisch sind. Dies bedeutet, dass für jeden konkreten Aufruf eine Funktion generiert wird. Die OCL-Constraint fängt mit *ETicketService.allInstances()* an, wodurch als erstes alle *ETicketServices* aufgesammelt werden. Zwar darf in diesem Fall der *ETicketService* nur einmal instanziiert werden, jedoch handelt es sich hierbei um einen Spezialfall, der durch den allgemeinen Fall (das ein Service mehrere Instanzen besitzt) ebenfalls abgedeckt ist. Die algebraische Spezifikation verwendet die Konstante *InspectorDeviceAllInstances*. Da es sich hierbei um eine Konstante handelt, kann diese in den Prädikaten *paidTicketCanNotBeLost* und *forAll1* ohne als Parameter übergeben worden zu sein, verwendet werden. Danach werden alle ausgestellten Tickets vom *ETicketService* durch die *collect*-Operation aufgesammelt. Dies ist in der algebraischen Spezifikation durch die Funktion *collect1* spezifiziert.

```
1  functions
2  collect1 : (agent → listofID) × agents → listofID;
3
4  axioms
5  collect1_emp :
6  ⊢  collect1(ETicketService-issuedTicketIDs,[]) = [];
7
8  collect1_rec :
9  ⊢ collect1(ETicketService-issuedTicketIDs,ag ' + agents)
10   = ETicketService-issuedTicketIDs(ag) +
11   collect1(ETicketService-issuedTicketIDs,agents);
```

Listing 7.12: Formale Spezifikation für *collect1*

collect1 (siehe Listing 7.12) ist rekursiv definiert und besteht aus zwei Teilen. Der erste Teil beschreibt das Ende der Rekursion (*collect1_ emp*). Das bedeutet, wenn die Liste der Serviceagenten leer ist, dann ist der Rückgabewert eine leere Liste. Der zweite Teil (*collect1_ rec*) beschreibt die Rekursion. Diese gibt eine Liste von allen ausgestellten Ticket-IDs aller *ETicketServices* zurück. Konkreter bedeutet dies, das der Rückgabewert zusammengesetzt ist aus den Ticket-IDs für einen Service *ag* und dem Ergebnis des nächsten *collect1* Aufrufes ohne die schon verwendete Serviceinstanz *ag*. *ag* ' bedeutet, dass eine Liste mit *ag* als Element erstellt wird. Um diese Funktion aufzurufen werden die Parameter *ETicketService-issuedTicketIDs* und *ETicketServiceAllInstances* benötigt (siehe Listing 7.11).

```
1  predicates
2  forAll1 : (agent → listofTicket) × (agent → listofID) ×
3            (agent → listofCardTicket) × listofID;
```

```
4
5
6  axioms
7  forAll1_emp :
8  ⊢  forAll1(ETicketCard-tickets,
9        ETicketCard-deletedTicketIDs,
10       ETicketServer-tickets,[]
11   ) = true;
12
13 forAll1_rec :
14 ⊢  forAll1(ETicketCard-tickets,
15       ETicketCard-deletedTicketIDs,
16       ETicketServer-tickets,
17       a_ID ' + a_listofID
18   )=(
19     (
20       isTicketReceivedByCard(ETicketCard-tickets,
21                   ETicketCard-deletedTicketIDs,
22                   a_ID
23       )
24       ∨
25       isTicketReadyForRetrieve(ETicketServer-tickets,a_ID)
26     )
27     ∧
28     forAll1(ETicketCard-tickets,ETicketCard-deletedTicketIDs,
29           ETicketServer-tickets,a_listofID)
30     );
```

Listing 7.13: Formale Spezifikation für *forAll1*

Listing 7.13 zeigt das Prädikat *forAll1* und dessen Axiomatisierung. Das Prädikat ist ebenfalls rekursive definiert und überprüft für jede ID eines ausgestellten Tickets, ob das Ticket schon von einer Karte empfangen wurde (*isTicketReceivedByCard*) oder ob das Ticket immer noch zum Abholen vom Service bereitsteht (*isTicketReadyForRetrieve*). Das Ende der Rekursion (*forAll1_emp*) ist ähnlich wie bei *collect1*. Wenn die Liste der ausgestellten Ticket-IDs leer ist, dann gilt das Prädikat. Die Rekursion ist in (*forAll1_rec*) definiert. Es wird für das erste Listenelement (*a_ID*) (Liste der ausgestellten Tickets IDs) überprüft, ob *isTicketReceivedByCard* oder *isTicketReadyForRetrieve* gelten und anschließend wird die Überprüfung für die restlichen Elemente rekursive durchgeführt.

```
1  predicates
2  isTicketReadyForRetrieve : (agent → listofCardTicket) × ID;
3
4  axioms
5  isTicketReadyForRetrieve :
6  ⊢ isTicketReadyForRetrieve(ETicketServer-tickets,a_ID)
7    ↔ (includes1(
8        collect2(
9          collect3(
10           collect4(ETicketServer-tickets,
11                 ETicketServerAllInstances
12           )
13         )
14       ),a_ID
15     ));
```

Listing 7.14: Formale Spezifikation der selbst modellierten OCL-Operation *isTicketReadyForRetrieve*

Listing 7.14 beschreibt die algebraische Spezifikation der selbst modellierten OCL-Operation *isTicketReadyForRetrieve*, die im Kapitel 4.3.2 gezeigt wurde. Das Verhalten wurde in OCL durch *ETicketService.allInstances().tickets.ticket.id ->includes(id)* definiert. Dieser OCL-Ausdruck verwendet mehrfach die erwähnte Kurzschreibweise, weshalb hierfür mehrere *collect*-Funktionen spezifiziert wurden. Zuerst werden die IDs aller gespeicherten Tickets aufgesammelt (*collect2, collect3 und collect4*) und anschließend wird mit der *includes*-Operation überprüft, ob die übergebene ID einer der aufgesammelten IDs entspricht.

```
1 predicates
2 includes7 : listofID × ID;
3
4 axioms
5 includes1:
6 ⊢  includes1(a_listofID,a_ID) = (a_ID ∈ a_listofID);
```

Listing 7.15: Formale Spezifikation der OCL-Operation *includes*

Listing 7.15 formalisiert die OCL-*includes*-Operation. Diese nimmt eine Liste von Ticket-IDs und eine Ticket-ID und überprüft ob die Liste die ID enthält. Dies geschieht mit dem vordefinierten ∈-Operator. Der *includes*-Operator ∈ könnte ebenso direkt verwendet werden, anstatt in einem *includes*-Prädikat geschachtelt zu werden. Jedoch ist ∈ ein Infix-Operator und *includes* ist eine Operation. Daher ist es für die automatische Generierung systematischer für jede OCL-Operation eine algebraische Funktion zu definieren.

OCL-Invarianten und die initialen Eigenschaften werden automatisch in das formale Model übersetzt und können zum Verifizieren von Sicherheitseigenschaften verwendet werden.

7.5 Externe Web Services

7.5.1 Integration in das formale Modell

Um Sicherheitseigenschaften für eine Anwendung zu verifizieren, die externe Web Services verwendet, müssen auch die Annahmen für die externen Services betrachtet werden. Diese sind als OCL-Vorbedingung und Nachbedingung auf den externen Servicemethoden definiert. Diese Annahmen werden automatisch in prädikatenlogische Formeln übersetzt, und nachdem eine externe Servicemethode ausgeführt wurde, gilt die dazugehörige Annahme. Somit ist das Verhalten der externen Servicemethoden, die in den Protokollen verwendet werden, formal spezifiziert und die Sicherheitseigenschaften können bewiesen werden.

```
1  functions
2       collect47: (agent → listofPayment) ×
3                          agents → listofPayment;
4  predicates
5       exists48: listofPayment × Pay;
6
7  procedures
8       pay: Pay, agent: bool ×
9                 (agent → listofPayment);
10
11 axioms
12 collect47_emp: ⊢ collect47(PayService-payments,[]) = [];
13 collect47_rec: ⊢ collect47(PayService-payments,ag ' + agents) =
14                         PayService-payments(ag) +
15                         collect47(PayService-payments,agents);
16
17 exists48_emp: ⊢ not exists48([],a_Pay);
18 exists48_rec: ⊢ exists48(a_Payment ' + a_listofPayment, a_Pay) ↔
19      a_Payment.reference_Nonce =
20                         a_Pay.order_Order.transactionID_Nonce ∧
21      a_Payment.price_string = a_Pay.order_Order.price_string  ∨
22      exists48(a_listofPayment, a_Pay);
23
24 pay: exPayService(ag) ⊢
25      <pay(a_Pay,ag; result-bool-var, PayService-payments)>
26           (result-bool-var →
27                exists48(collect47(PayService-payments,ag),a_Pay));
```

Listing 7.16: Annahme an die externe Bezahlmethode von AuthorizeNET als algebraische Spezifikation

Listing 7.16 zeigt die algebraische Spezifikation der Annahme an die externe Bezahlmethode von AuthorizeNET, die bei der erweiterten ETicket-Anwendung in Abschnitt 4.5.3 verwendet wurde. Die Annahme ist als OCL-Nachbedingung (siehe Listing 4.4) anhand der *pay*-Operation der Proxyklasse *PayService* definiert (siehe Abbildung 4.34). Diese Annahme ist im Listing 7.16 in Zeile (8-9) als Prozedur deklariert. Im Gegensatz zu einer Sicherheitseigenschaft, die als Prädikat spezifiziert ist, kann eine Prozedur als Programm ausgeführt werden. Dies ist notwendig, da die *pay*-Operation als Teil des dynamischen Verhaltens einer modellierten Komponente aufgerufen werden muss. Die *pay*-Prozedur erweitert die Signatur der *pay*-Operation, um einen Agenten (in diesem Fall *PayService*) und eine dynamische Funktion, die einen Agenten auf den Wert eines seiner Attribute abbildet. Hierdurch können die Attribute des Agenten modifiziert werden. Die Axiomatisierung der Prozedur ist in Zeile (24-27) spezifiziert. Die *pay*-Prozedur kann ausgeführt werden, falls der Agent *ag* vom Typ *PayService* ist. Nachdem *pay* ausgeführt wurde, gilt der Ausdruck in Zeile (26-27). Dieser ist sehr ähnlich zu der OCL-Nachbedingung. Der Einfachheit wegen und da die Spezifikation ohnehin automatisch erstellt wird, wird für jedes Vorkommen einer OCL-Operation eine neue Funktion bzw. ein neues Prädikat generiert, welches ebenfalls den Rumpf der OCL-Operation spezifiziert. Zeile 2 deklariert und Zeile (12-15) definiert eine *collect*-Operation, die alle Zahlungen (*payments*) vom *PayService* aufsammelt. Zeile 5 und Zeile (17-22) spezifiziert die *exists*-Query vom Listing 4.4. Diese Query

überprüft, ob die Liste *PayService-payments* eine Zahlung enthält, mit der gleichen Transaktion ID wie in der Eingabenachricht *pay*.

Im Gegensatz dazu, ist die Konvertierungsmethode *c1* aus Abbildung 4.28 in Kapitel 4.5.1 nicht in OCL spezifiziert, sondern mit MEL in einem UML-Subaktivitätsdiagramm. Diese Konvertierungsmethoden sind daher nicht wie die Annahmen deklarativ spezifiziert, weshalb sie in ein Programm übersetzt werden, welches Schritt für Schritt in dynamischer Logik ausgeführt wird. Das dynamische Verhalten der Komponenten wird auf dieselbe Weise modelliert und übersetzt.

7.5.2 Austauschbarkeit ohne erneute Verifikation

Damit ein externer Service nach der Verifikation mit geringem Verifikationsaufwand ausgetauscht werden kann, ist die Annahme, die zum Beweisen der Sicherheitseigenschaften verwendet wird unabhängig von der konkreten Zusicherung des realen Web Services. Um zu zeigen, dass die Sicherheitseigenschaft auch für den realen Web Service gilt, ist ein Refinement zwischen Annahme und Zusicherung notwendig. Dies wird durch ein Datarefinement sichergestellt, welches nur die Nachrichten und die internen Zustände des Proxies und des externen Web Services betrachtet. Daher ist das Refinement unabhängig von der eigentlichen Anwendungslogik und den Sicherheitsprotokollen.

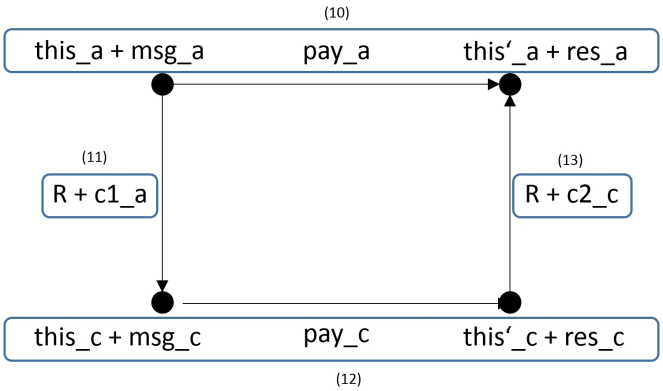

Abbildung 7.3: Data Refinement

```
1  pay_a: PayService × Pay → PayService × Boolean;
2  pay_c: AuthorizeNET × CreateCustomerProfileTransactionSoapIn
3       → AuthorizeNET × CreateCustomerProfileTransactionSoapOut;
4  R : PayService × AuthorizeNET;
5
6  R(ps, an) ↔ forall s. s ∈ ps.payments.reference.nonce ↔
7  s ∈ an.accounts.transactions[Inpayment].orderInfo.description;
8
9  ProofObligation:
10 pay_a(this_a, msg_a) = [this'_a, res_a] ∧
11 R(this_a,this_c) ∧ msg_c = c1_a(msg_a) ∧
12 pay_c(this_c, msg_c) = [this'_c, res_c]
13 -> c2_c(res_c) = res_a ∧ R(this'_a, this'_c);
```

Listing 7.17: Data Refinement

Abbildung 7.3 und Listing 7.17 zeigen ein solches Datarefinement für das elektronische Ticketsystem, welches den Bezahldienst AuthorizeNET verwendet. Zum besseren Verständnis wird in Listing 7.17 eine Pseudosprache verwendet. *pay_a* beschreibt die Operation *pay* vom *PayService* mit der Annahme aus Listing 4.4, die als algebraische Spezifikation in Listing 7.16 formalisiert wurde. Da die Operation den Zustand von *PayService* modifiziert, erhält die Prozedur nicht nur die Nachricht *msg_a* vom Typ *Pay*, sondern auch die *PayService* Instanz *this_a* als Eingabeparameter und gibt die modifizierte *PayService*-Instanz *this'_a* und das Ergebnis der *pay*-Operation *res_a* vom Typ Boolean zurück (siehe Zeile (1,10) in Listing 7.17 und (10) in Abbildung 7.3). *pay_c* ist auch eine Prozedur (2,12) und ergibt sich aus der Zusicherung des externen Web Services. Das Datarefinement zeigt, dass das konkrete *pay_c* ein Refinement des abstrakten *pay_a* ist. Hierzu muss die Nachricht *msg_a* nach *msg_c* und das Ergebnis *res_c* nach *res_a* übersetzt werden. Dies geschieht durch die modellierten Konvertierungsmethoden *c1* und *c2* (siehe Abbildung 4.28), die in die formalen Prozeduren *c1_a* und *c2_c* übersetzt wurden. Die Konvertierung zwischen den internen Zuständen von *PayService* und *AuthorizeNET* muss durch die Relation *R* (4-7) spezifiziert werden. In der ETicket-Anwendung ist diese Beziehung sehr einfach, da der *PayService*-Proxy nur eine Liste von Zahlungen (*payments*) besitzt. Die Beziehung, die als Prädikat spezifiziert wurde, beschreibt, dass jedes Element in dieser Liste in den eingehenden Zahlungen von *AuthorizeNET* vorhanden ist und auch umgekehrt. Um das Refinement zu zeigen, muss die Beweisverpflichtung in Listing 7.17 (10-13) bewiesen werden. Vereinfacht ausgedrückt, wenn *pay_c* mit dem konvertierten Input von *pay_a* aufgerufen wird (11-13), entspricht der konvertierte Output dem Output von *pay_a* (10,13). Wegen der unterschiedlichen Datenstruktur, der unterschiedlichen Namen und unterschiedlichen Informationsgehalt zwischen dem Proxy und dem externen Web Service kann es schwer sein zu erkennen, dass die Zusicherung ein Refinement der Annahme ist. Deshalb muss die Beweisverpflichtung bewiesen werden.

Ein Problem hierbei ist jedoch, dass beim Datarefinement der Angreifer nicht betrachtet wurde. Damit sich die Sicherheitseigenschaften entlang des Datarefinements vererben, muss jedoch auch der Angreifer betrachtet werden. Der Angreifer

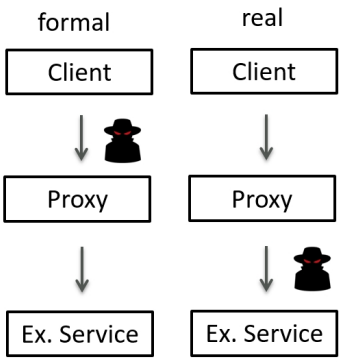

Abbildung 7.4: Angreifer beim Refinement

wird bei der Verifikation der Sicherheitseigenschaften jedoch nur in Abhängigkeit der Annahme und somit nur in Abhängigkeit der zwischen Client und Proxy übertragenen Nachrichten betrachtet (siehe Abbildung 7.4). Somit gibt es im Formalen einen Angreifer zwischen Client und Proxy, aber nicht zwischen Proxy und dem externen Service. Im realen Programmcode ruft der Client den Proxy intern auf, weshalb es hier keinen Angreifer gibt, sondern dieser erst auf der Leitung zwischen Proxy und dem realen externen Service sitzt. Das Problem hierbei ist, dass der Proxy die Nachrichten konvertiert und die tatsächlich übertragenen Nachrichten zwischen Proxy und dem realen externen Service nicht den Nachrichten zwischen Client und Proxy entsprechen müssen. Tatsächlich können die konkreten Nachrichten in drei Typen unterteilt werden.

- abstrakte Nachrichten
- falsch kodierte Nachrichten
- Nachrichten mit Attributen, die bei der Konvertierung nicht betrachtet wurden

Falls die konkreten Nachrichten den abstrakten Nachrichten entsprechen, dann vererben sich die Sicherheitseigenschaften entlang des Datarefinements. Falls es sich um falsch kodierte Nachrichten handelt, dann wird angenommen, dass diese vom Kodierungsmechanismus erkannt und verworfen werden. Somit müssen diese Nachrichten bei der Verifikation nicht betrachtet werden. Und als letztes kann es sich um Nachrichten mit Attributen handeln, die bei der Konvertierung nicht betrachtet wurden. Dies ist möglich, da der externe Service mehr unterstützen kann als von der modellierten Anwendung verwendet wird (z.B Rabattcodes). Der Angreifer ist jedoch abhängig von seinen Fähigkeiten in der Lage diese nicht verwendeten Felder in den konkreten Nachrichten zu verändern und somit ein formal nicht betrachtetes Verhalten zu verursachen. Damit dies nicht möglich ist wird die

Sicherstellung von Nachrichtenintegrität gefordert. Dies kann z.B. durch Signaturen oder die Verwendung von TLS sichergestellt werden. Somit ist der Angreifer nicht in der Lage die Nachrichten zu manipulieren. Jedoch muss zusätzlich sichergestellt werden, dass die konkrete Nachricht nicht mehr Informationen enthält als die abstrakte Nachricht. Hierzu dürfen die Konvertierungsmethoden nur Zugriff auf die Inputparamter besitzen und es dürfen keine neuen sicherheitsrelevanten Informationen generiert werden (d.h. keine Erstellen von Sicherheitsdatentypen). Das Ergebnis ist, dass sich somit die konkreten Nachrichten auf die abstrakten abbilden lassen und sich die Sicherheitseigenschaften entlang des Datarefinements vererben.

7.6 Verifikation

Aus einem SecureML-Modell wird einerseits automatisch Code generiert und andererseits ein formales Modell. Die Transformationen wurden so erstellt, dass der generierte Code ein Refinement des formalen Modells ist. Dadurch ist es möglich, dass Sicherheitseigenschaften für die generierte Anwendung verifiziert werden können. Gewöhnlich werden nur generische Eigenschaften wie Geheimhaltung oder Authentifikation für Sicherheitsprotokolle bewiesen (siehe [73] für einen Überblick). Im Gegensatz dazu liegt der Schwerpunkt bei SecureMDD auf anwendungsspezifischen Sicherheitseigenschaften (siehe Kapitel 4.3). SecureML unterstützt drei Arten von OCL-Constraints. *ApplicationConstraints* (Sicherheitseigenschaften), *InitialzeConstraints* (initiale Eigenschaften) und *HelperConstraints* (Hilfsaussagen). Die *ApplicationConstraints* und die *HelperConstraints* werden mit OCL-Invarianten definiert und müssen nach jedem Protokollschritt gelten. Dies bedeutet, dass sie auch initial gelten. Jedoch gelten einige *Constraints* nicht für einen beliebigen initialen Zustand. Zum Beispiel kann die Sicherheitseigenschaft *Tickets werden nicht mehrfach gestempelt* nicht gelten, falls die initiale Eigenschaft *alle Karten haben unterschiedliche IDs* nicht gilt. Deshalb ist es wichtig den initialen Zustand ebenfalls zu spezifizieren. Gewöhnlich kennt der Anwendungsentwickler nicht alle initialen Annahmen, die für die Verifikation notwendig sind. Daher müssen einige initiale Annahmen beim Verifizieren von Hand spezifiziert werden.

$$InitCons \wedge AddInitAssum \vdash AppCons \wedge HelpCons \tag{7.1}$$

Theorem 7.1 sollte initial gelten und beschreibt, dass die Sicherheitseigenschaften (AppCons) und die Hilfsaussagen (HelpCons) aus den initialen Eigenschaften (InitCons) und einigen zusätzlichen initialen Annahmen (AddInitAssum) folgen müssen. Die Sicherheitseigenschaften sowie die Hilfsaussagen müssen jedoch vor und nach jedem Protokollschritt gelten (siehe Theorem 7.2).

$$AppCons \wedge HelpCons \xrightarrow{STEP} AppCons \wedge HelpCons \tag{7.2}$$

Mit den zwei erwähnten Theoremen und dem formalen Modell können die

Sicherheitseigenschaften verifiziert werden. Für die ETicket-Anwendung wurden die drei vorgestellten Sicherheitseigenschaften (siehe Abschnitt 4.3) verifiziert.

Die Debitcard-Anwendung wurde ebenfalls verifiziert. Debitcard ist ein EC-Karten-System. Ein Kunde kann mit seiner Debitcard an einem Bankautomaten (*ATM*) Geld abheben, aber auch an einem Service-Terminal Überweisungen tätigen. Es wird zwischen Banken mit Filialen (*AffiliatedBank*) und reinen Onlinebanken (*DirectBank*) unterschieden. Ein Kunde einer Onlinebank kann ebenfalls Geld an einem Bankautomaten einer anderen Bank abheben. Hierzu ruft der Web Service der Bankfiliale (*AffiliatedBank*) den Web Service der Onlinebank (*DirectBank*) auf. Das Deployment diagramm, das Klassendiagramm sowie die Aktivitätsdiagramme, welche die Hauptfunktionalität beschreiben sind im Anhang in Abschnitt 9.1 abgebildet. Die bewiesene anwendungsspezifische Sicherheitseigenschaft ist, dass der Geldbetrag im System konstant bleibt. Konkret bedeutet dies:

> Die Summe von allen Kontoständen plus dem Betrag, der an allen Geldautomaten abgehoben wurde ist konstant.

Diese Eigenschaft kann als OCL-Constraint formalisiert werden:

```
Bank.allInstances().accounts.balance->sum() +
ATM.allInstances().moneyPaidOut->sum() = C
```

wobei Bank eine Oberklasse von *AffiliatedBank* und *DirectBank* ist und die Konten im Attribut *accounts* speichert werden. *accounts* speichert wiederum die Kontostände der Konten im Attribut *balance*. Zudem speichert ein Bankautomat (ATM) alle ausbezahlten Beträge in dem Attribut *moneyPaidOut*. Des Weiteren ist C eine unspezifizierte Konstante ist. Diese OCL-Constraint wird in eine Eigenschaft der abstrakten Zustandsmaschine (ASM) übersetzt, z.B. "die Summe ist Konstant in allen Zuständen aller Protokollabläufe der ASM".

```
BanksMoney(accounts) =
          Bank.allInstances().accounts.balance->sum()

ATMMoney(moneyPaidOut) =
          ATM.allInstances().moneyPaidOut->sum()

sum = BanksMoney(accounts) + ATMMoney(moneyPaidOut)
```

BanksMoney(accounts) und *ATMMoney(moneyPaidOut)* sind die algebraischen Terme, die bei der Übersetzung der OCL-Constraint entstanden sind. *BanksMoney* iteriert über die Bankkonten und summiert deren Guthaben (*balance*) auf und *ATMMoney* iteriert über alle Bankautomaten (*ATM*) und summiert das ausbezahlte Geld *moneyPaidOut* auf.

Die Eigenschaft kann somit wie folgt formalisiert werden:

$$\text{init}(\ldots) \wedge sum = \text{C} \rightarrow [\text{STEP}^*]\ sum = \text{C}$$

$[\cdot]$ ist der Box-Operator der dynamischen Logik. $[\alpha]\varphi$ bedeutet, dass wenn α terminiert danach die Bedingung φ gilt. Es wird im initialen Zustand angefangen (z.B. bisher wurden keine Protokollschritte ausgeführt, alle Komponenten befinden sich in ihren initialen Zuständen, der Angreifer besitzt ein initiales Angreiferwissen usw.). Anschließend werden die Protokollschritte nichtdeterministisch ausgewählt und ausgeführt. Dadurch werden alle endlichen Abläufe von Schritten betrachtet. Wie im Abschnitt 7.1.3 schon erwähnt besteht die ASM aus einer while-Schleife, welche einen Schritt ausführt oder abbricht. Somit werden Invarianten für jeden Schritt einer ASM bewiesen. Eine solche Invariante muss natürlich auch im initialen Zustand gelten:

$$(\text{INV}(\ldots) \wedge \mathit{sum} = \text{C}) \;\rightarrow\; [\text{STEP}]\ (\text{INV}(\ldots) \wedge \mathit{sum} = \text{C})$$

Die gezeigte Eigenschaft ist jedoch nicht vollständig, da bisher nicht abgeschlossene Protokollabläufe ebenfalls betrachtet werden müssen. Zum Beispiel existiert die Situation, dass während einer Überweisung der Betrag vom Absender schon abgebucht wurde, jedoch noch nicht beim Empfänger hinzugebucht wurde. Somit sitzt der Betrag noch in der Nachricht zwischen den zwei Banken fest (es befindet sich im Transit) und muss mit berücksichtigt werden. Somit muss die aktuelle Summe wie folgt berechnet werden:

sum = BanksMoney(accounts) + ATMMoney(atms) + MoneyInTransit(inboxes)

Die Definition von *MoneyInTransit* verwendet Inboxen, die für die Nachrichtenübertragen verwendet werden. Die korrekte OCL-Constraint müsste somit Inboxen verwenden. Zum Beispiel, wenn alle Inboxen leer sind, dann ist *MoneyInTransit* = 0 wodurch die Eigenschaft wie folgt ausgedrückt werden kann:

```
AllInboxes().isEmpty implies[7]
Bank.allInstances().accounts.balance->sum() +
ATM.allInstances().moneyPaidOut->sum() = C
```

Es ist möglich *MoneyInTransit* in OCL zu definieren oder es direkt im KIV-System zu spezifizieren. Wie sich herausgestellt hat, ist die korrekte Definition ziemlich kompliziert und wurde erst nach mehreren Verbesserungen gefunden. Der Grund für die Komplexität liegt im Verhalten der Protokolle selbst, aber auch in den Angreiferfähigkeiten.

Um Geld von einem Bankautomat (ATM) abzuheben wird das Benutzerkonto belastet und der Betrag, der ausbezahlt werden soll wird entweder direkt an den Bankautomaten geschickt oder über die *AffiliatedBank*. Da das Konto schon belastet wurde, jedoch das Geld noch nicht ausbezahlt, muss der Betrag mitgezählt werden. Bei einer Onlineüberweisung (siehe Abbildung 9.15 im Anhang) wird das Absenderkonto belastet und der Betrag in der *Subtransaction*-Nachricht an eine

[7] *implies* ist ein Schlüsselwort in OCL, wohingegen -> für Operationen auf Collections ist.

andere Bank geschickt. Dieser Betrag muss zu der Bank des Empfängers hinzugezählt werden. Die Bank des Empfängers antwortet mit einer *SubtransactionSuccessful*-Nachricht. Jedoch kann das Hinzubuchen des Betrags zum Empfängerkonto wegen einer falschen Kontonummer fehlschlagen. Deshalb besitzt die Nachricht eine binäre Variable, welche beim Fehlschlagen auf *false* gesetzt wird und bei einer erfolgreichen Überweisung auf *true*. Somit muss der Betrag, der in der *SubtransactionSuccessful*-Nachricht enthalten ist genau dann und nur dann dazugezählt werden, wenn die binäre Variable *false* ist.

Jedoch muss auch noch der Angreifer betrachtet werden. Dieser hat in der betrachteten Anwendung zwar sehr eingeschränkte Fähigkeiten, jedoch besitzt er die volle Kontrolle über den Kommunikationspfad zwischen dem Service-Terminal (*PC*) und *OnlinebankingService* (siehe Abbildung 9.13 im Anhang). Dies bedeutet, dass er beliebige Nachrichten wie *TerminalPayOut*, *Subtransaction* oder *SubtransactionSuccessful* einschleusen kann, selbst wenn sie komplett nutzlos sind, da sie vom *PC* und *OnlinebankingService* ignoriert werden. Somit dürfen die Beträge nur in den Nachrichten zwischen den Banken und den Bankautomaten gezählt werden, da der Angreifer dort keine Nachrichten einschleusen kann.

Verifikationsaufwand

Für die Beweise der Debitcard-Fallstudie wurden 244 Theoreme verwendet, die 953 Benutzerinteraktionen und 17125 Beweisschritte benötigten. Die erste Version dieser Fallstudie benötigte 484 Theoreme, 4229 Benutzerinteraktionen und 27038 Beweisschritte. Für die ETicket-Fallstudie wurden 923 Theoreme verwendet, die 2852 Benutzerinteraktionen und 43038 Beweisschritte benötigten. Die erste Version benötigte 1631 Theoreme, 5788 Benutzerinteraktionen und 69439 Beweisschritte. Dies zeigt wie sich die Verifikationstechnik von SecureMDD-Anwendungen und kryptographischen Protokollen über die Zeit verbessert hat. Andere Fallstudien zeigten ähnliche Verbesserungen. Alle Spezifikationen, Lemmas und Beweise können zusammen mit anderen Fallstudien auf der Webseite[8] gefunden werden.

7.7 Verwandte Arbeiten

In dieser Arbeit wurde das Verhalten von Web Services sowie die Kommunikation mit anderen Web Services in ASMs spezifiziert. Andere Ansätze zum Spezifizieren von Web Services mit ASMs sind die von Börger und Thalheim [24] und die von Börger und Sörensen [23]. Börger betrachtet Aspekte wie Nebenläufigkeit, heterogene Zustände und Modularität. Es werden jedoch keine sicherheitskritischen Systeme betrachtet in denen ein Angreifer versucht die Sicherheitseigenschaften zu verletzen. Ebenfalls werden keine stateful Web Services betrachtet.

Pironti et al. [74] generieren verifizierten Aufrufercode, welcher einen existierenden TLS-Services verwendet. Jedoch mussten sie hierfür tausende Zeilen von Code

[8] http://www.isse.uni-augsburg.de/projects/reif/securemdd/

für die Konvertierung der Nachrichtentypen händisch und ohne Sicherheitsgarantien schreiben. In [40] wird die Verifikation von Systemen mit externen Services betrachtet. Jedoch wird hierbei nicht die Sicherheit der Anwendung betrachtet. Mir ist kein modellgetriebener Ansatz bekannt, welcher die sichere Integration und Austauschbarkeit von existierenden Web Services in sicherheitskritischen Anwendung unter Beachtung eines Angreifers sowie Sicherheitseigenschaften für die gesamte Anwendung einschließlich der externen Web Services betrachtet.

Zum Beweisen der Sicherheit von kryptographischen Protokollen existieren bereits einige Verifikationstechniken und Werkzeuge. Ein Überblick über diese ist in [73] zu finden. Die meisten von ihnen basieren auf automatischen Tools mit dem Fokus auf generische Sicherheitsprotokolle (z.B. Authentifizierungsprotokolle) und der Verifikation von Standardsicherheitseigenschaften. Im Gegensatz dazu sind die in dieser Arbeit gezeigten Protokolle sowie Sicherheitseigenschaften sehr spezifisch für die betrachtete Anwendung. Es ist unklar ob die automatischen Beweiser mit solchen Sicherheitseigenschaften zurechtkommen würden.

Dennoch existieren einige Ansätze, die mit anwendungsspezifischen Sicherheitseigenschaften umgehen können. Ein Ansatz der mit diesem verwandt ist, ist der induktive Ansatz von Paulson [71], der zur Verifikation den Theorembeweiser Isabelle verwendet. Bella erweiterte diesen Ansatz für Chipkarten [12] jedoch nur mit dem Fokus auf generische Sicherheitsprotokolle. In [13] geben Bella, Massacci und Paulson einen Überblick über ihre Arbeit an SET (Secure Electronic Transaction). Dabei handelt es sich um eine Menge von E-Commerce Protokollen, die von Visa und Mastercard entwickelt wurden. Die Fallstudie wurde mit ihrem induktiven Ansatz formal modelliert und verifiziert. Die betrachteten und bewiesenen (anwendungsspezifischen) Sicherheitseigenschaften sind, dass nur die Bank die Zahlungsinformationen der Kunden erfährt und nicht der Verkäufer sowie, dass die Bank die Bestellinformationen nicht erfährt.

Grünbauer et al. [37] verwendeten AUTOFOCUS um eine Bankanwendung, in der Kunden eine Onlineüberweisung tätigen können, zu modellieren. Im wesentlichen wird jedoch nur das Schlüsselaustauschprotokoll und der erste Teil des Überweisungsprotokolls der in dieser Arbeit betrachteten Debitcard-Fallstudie behandelt. Der tatsächliche Überweisungsvorgang wird jedoch nicht betrachtet. Es wurde die Geheimhaltung und die Authentizität der Überweisungen mittels Model Checking bewiesen. Jedoch mussten die Angreiferfähigkeiten vereinfacht werden, da das ursprüngliche Modell zu komplex war um dieses automatisch zu verifizieren.

KAPITEL 8

Schluss

In dieser Arbeit wurde ein modellgetriebener Ansatz für sicherheitskritische Web Service-Anwendungen, die mit Chipkarten kommunizieren können, vorgestellt. Der Ansatz baut auf den Ergebnissen von Moebius auf [62], die sich mit der modellgetriebenen Entwicklung von sicherheitskritische Smart Card-Anwendungen befassen.

Eine große Herausforderung bei der Entwicklung eines modellgetriebenen Ansatzes für Web Services-Anwendungen ist jedoch, dass der Anwendungsbereich von Web Services sehr vielseitig ist und reale Anwendungen mit existierenden Web Services kommunizieren. Dieses Problem wurde gelöst, indem der Ansatz modular um bereits existierende Web Services erweitert werden kann. Falls die Funktionalität der existierenden Web Services für die zu zeigenden Systemeigenschaften relevant ist, werden Annahmen spezifiziert unter denen das modellierte System verifiziert wird. Die Sicherstellung der Annahme für eine existierende Web Service-Operation, die nur lokalen Code ausführt kann z.B. mit KeY unabhängig von einem Angreifer verifiziert werden, da der Angreifer schon bei der Verifikation des gesamten Systems betrachtet wird. Zudem kann ein existierender Web Service ausgetauscht werden, ohne dass die gesamte Anwendung neu verifiziert werden muss, obwohl sich die Nachrichtenstruktur unterscheidet. Des Weiteren wurde SecureML, eine Modellierungssprache für sicherheitskritische Web Service-Anwendungen, vorgestellt. Anhand dieser Sprache wird lauffähiger Code für Web Services generiert. Dies bedeutet, dass nicht nur die Web Service-Schnittstelle generiert wird, sondern auch das interne Verhalten der Web Services. Zudem können die generierten Web Services mit Chipkarten über modellierte kryptographische Protokolle kommunizieren. Außerdem werden aus einem SecureML-Anwendungsmodell formale Spezifikationen generiert um anwendungsspezifische Sicherheitseigenschaften für die Anwendung mittels Model-Checking zu überprüfen und mittels interaktiver Verifikation zu beweisen. Die anwendungsspezifischen Sicherheitseigenschaften können in einem SecureML-Anwendungsmodell mit einer Erweiterung von OCL definiert werden. Für das Model-Checking wurde AVANTSSAR verwendet. Dabei handelt es sich um ein Tool zum automatischen Validieren der Sicherheit von

serviceorientierten Architekturen. Hierdurch können schon beim Designen der Anwendung Sicherheitslücken, frühzeitig gefunden werden. Es hat sich jedoch herausgestellt, dass die Anwendungsmodelle durch ihren hohen Detailierungsgrad, der jedoch notwendig ist um lauffähigen Code zu generieren und anwendungsspezifische Sicherheitseigenschaften zu garantieren, zu groß sind um in angemessener Zeit mit Model-Checking überprüft zu werden. Deshalb wurde SecureML um die Modellierung von Abstraktionen erweitert. Die formale Spezifikation zum interaktiven Verifizieren, formalisiert Web Services mit ASMs und algebraischen Spezifikationen. Hierbei wurden Aspekte wie dynamische Adressierung, Sitzungsmanagement und sichere Standardübertragungsprotokolle unter Berücksichtigung eines Angreifers betrachtet.

Der in dieser Arbeit vorgestellte Ansatz wurde anhand mehrerer Fallstudien evaluiert sowie mit großem Erfolg mehrere Jahre von Studenten in einer praktischen Vorlesungsveranstaltung genutzt. Die Studenten konnten lauffähige Anwendungen entwickeln, sich jedoch dank SecureMDD hauptsächlich auf die sicherheitskritischen Protokolle konzentrieren. Das Vorgehen bestand darin, zuerst die Systemteilnehmer, die Kommunikationswege und die Angreiferfähigkeiten mit SecureML zu definieren. Anschließend mussten sie ein kryptographisches Protokoll entwickeln, welches vorgegebene Eigenschaften unter Berücksichtigung eines Angreifers sicherstellt. Zum Schluss wurden die generierten und lauffähigen Anwendungen auf Servern und Chipkarten installiert.

KAPITEL 9

Anhang

9.1 Modellierung

Onlinebanking

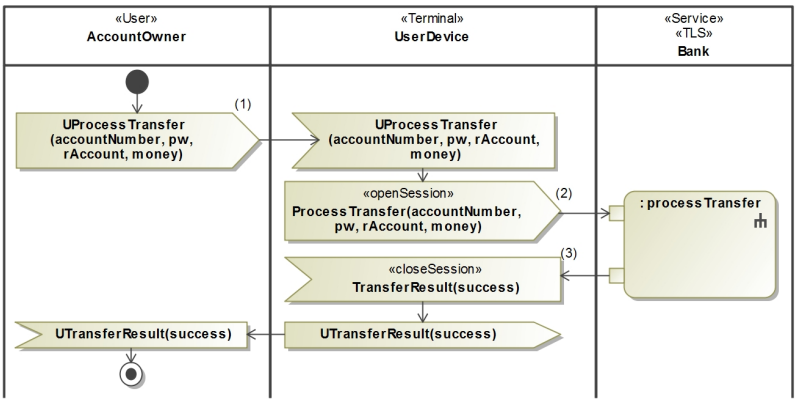

Abbildung 9.1: Aktivitätsdiagramm zum Tätigen einer Überweisung

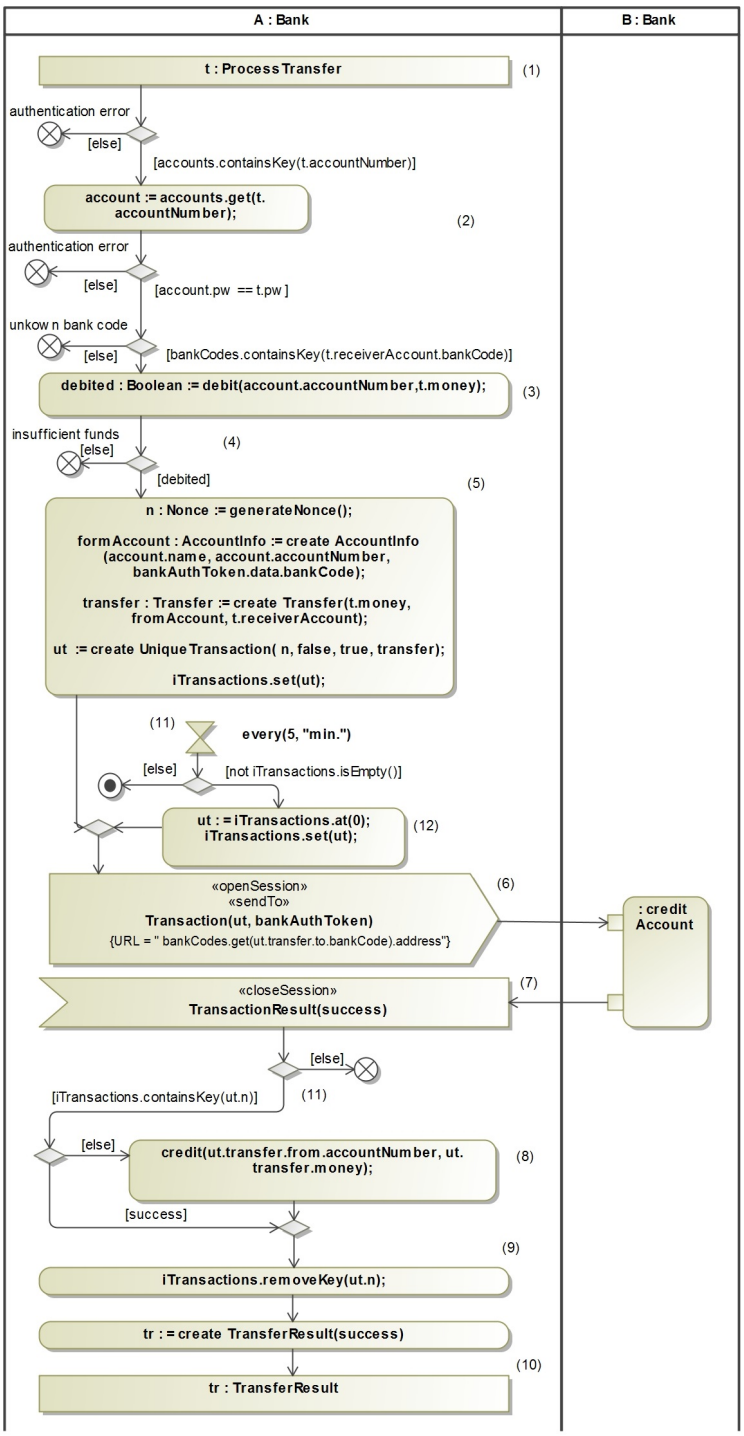

Abbildung 9.2: Serviceoperation *processTransfer* zur internen Ausführung einer Überweisung

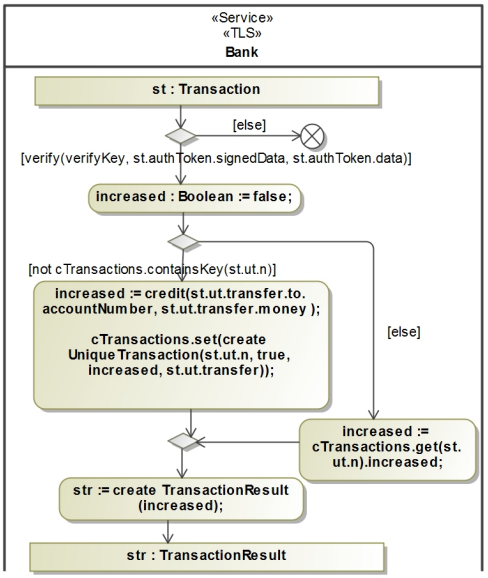

Abbildung 9.3: Serviceoperation *creditAccount*, die einen Betrag einem Empfängerkonto hinzubucht

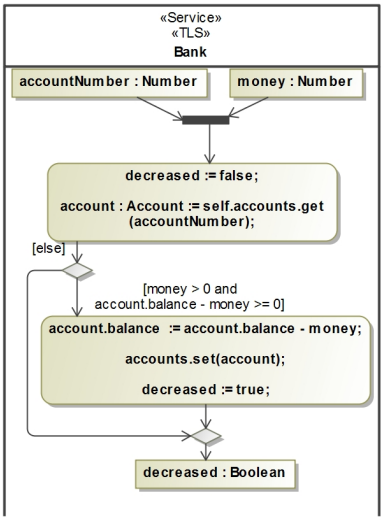

Abbildung 9.4: Interne Operation *debit*, die ein Konto mit einem bestimmten Betrag belastet

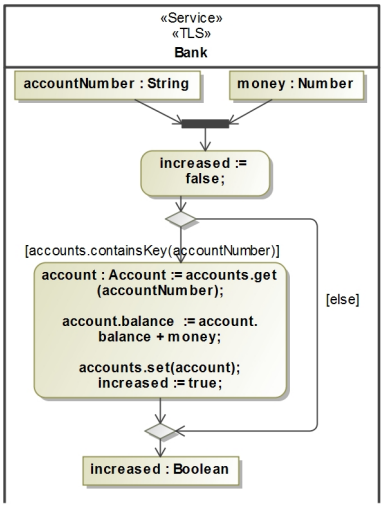

Abbildung 9.5: Interne Operation *credit*, die ein Betrag zu ein Konto hinzugucht

ETicket

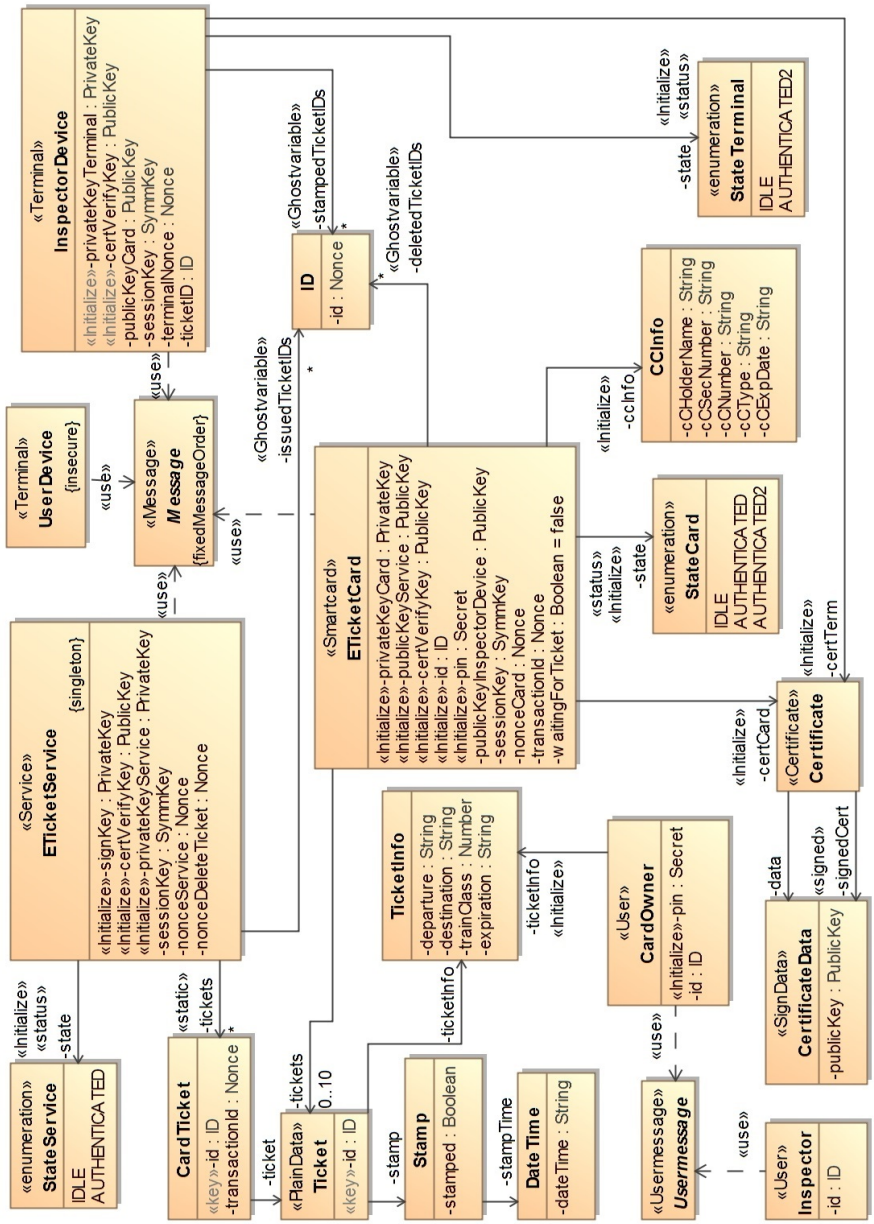

Abbildung 9.6: Klassendiagramm von ETicket

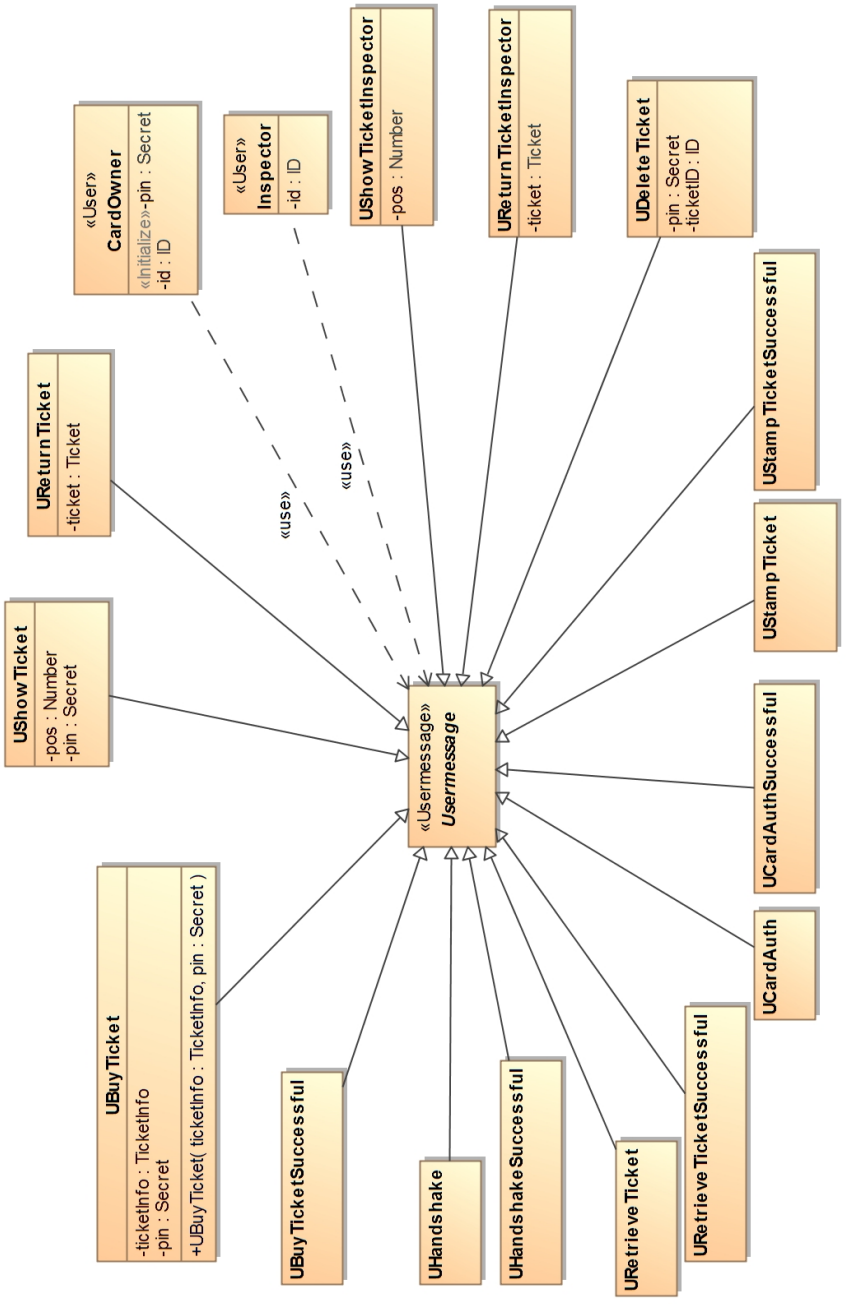

Abbildung 9.7: Benutzernachrichten von ETicket

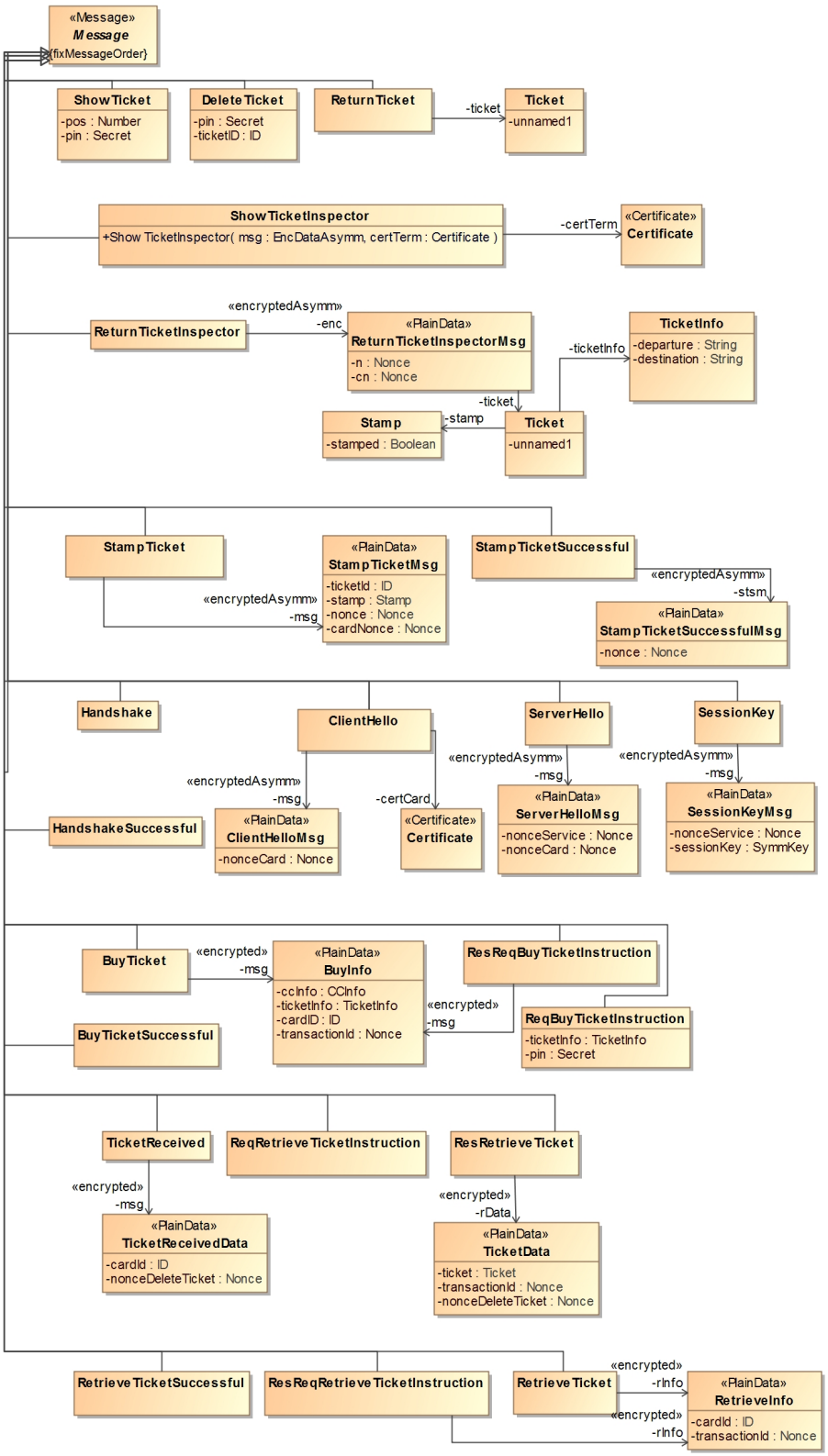

Abbildung 9.8: Nachrichten von ETicket

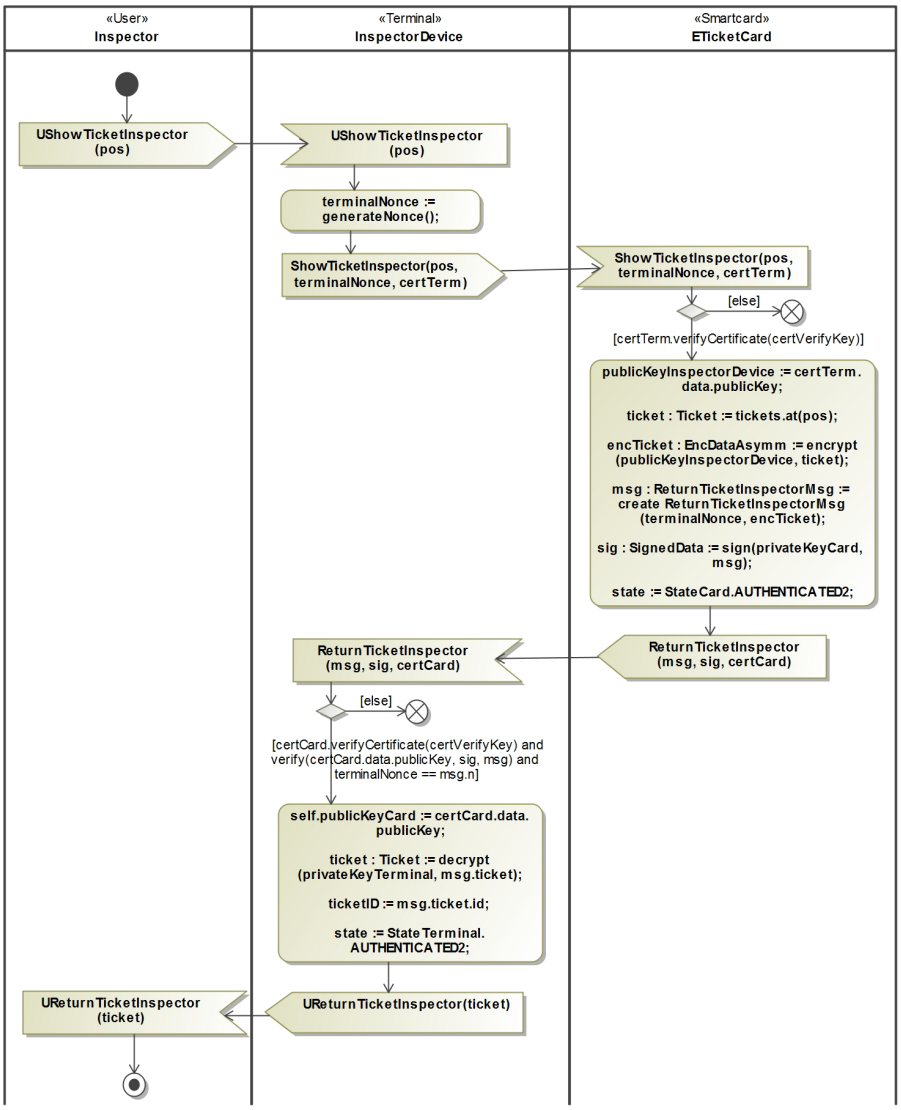

Abbildung 9.9: Protokoll zum Anzeigen von Tickets beim Schaffner

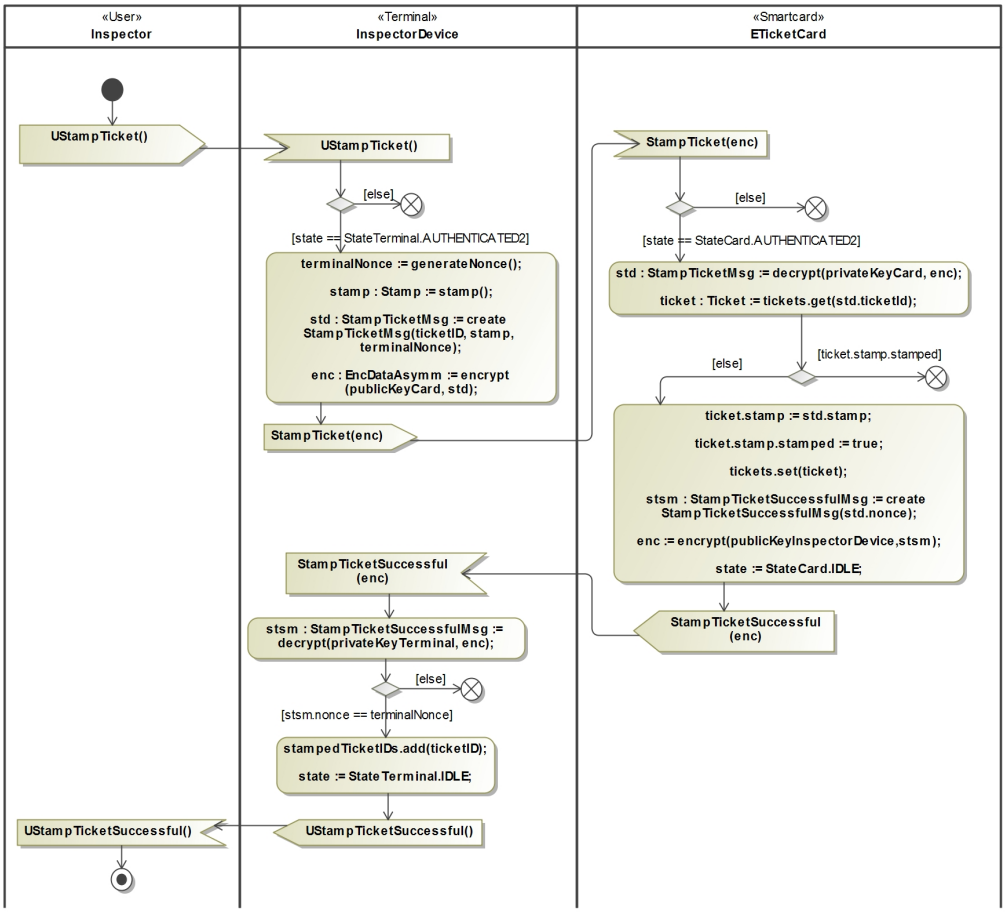

Abbildung 9.10: Protokoll zum Stempeln von Tickets

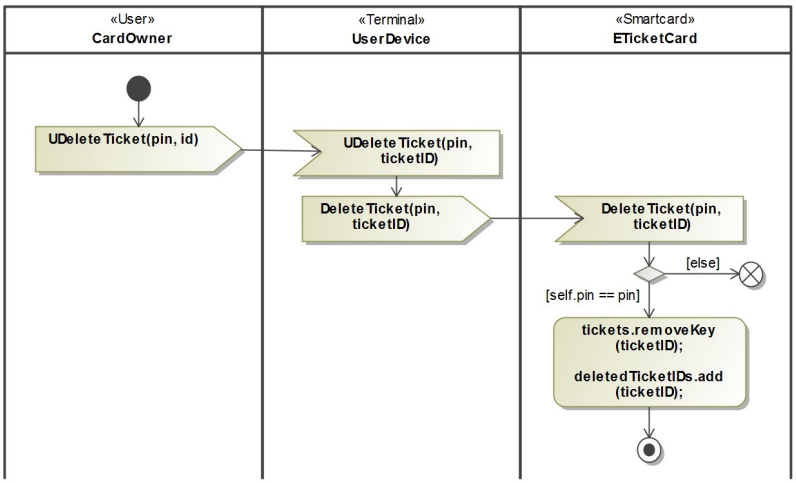

Abbildung 9.11: Protokoll zum Löschen von Tickets

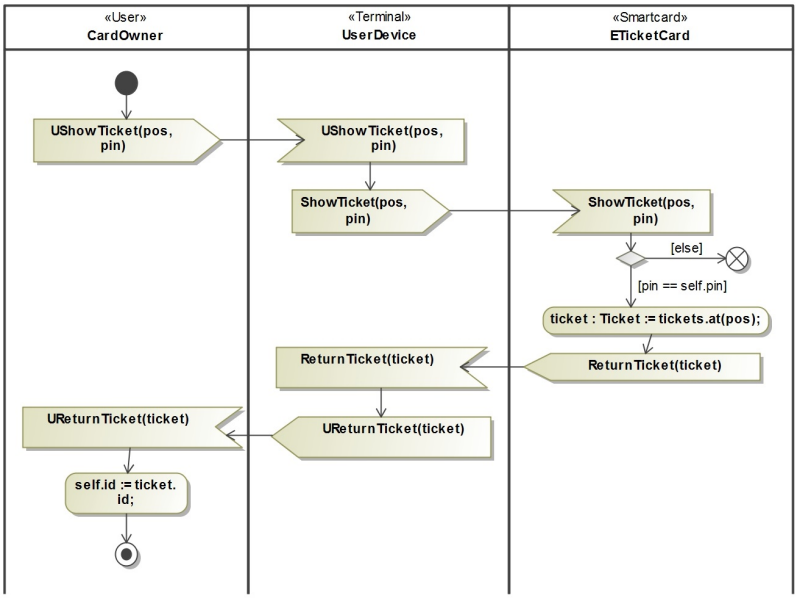

Abbildung 9.12: Protokoll zum Anzeigen von Tickets

Debitcard

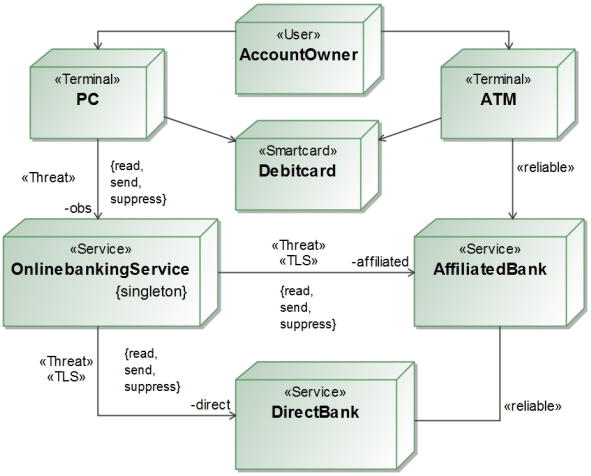

Abbildung 9.13: Deployment diagramm von Debitcard

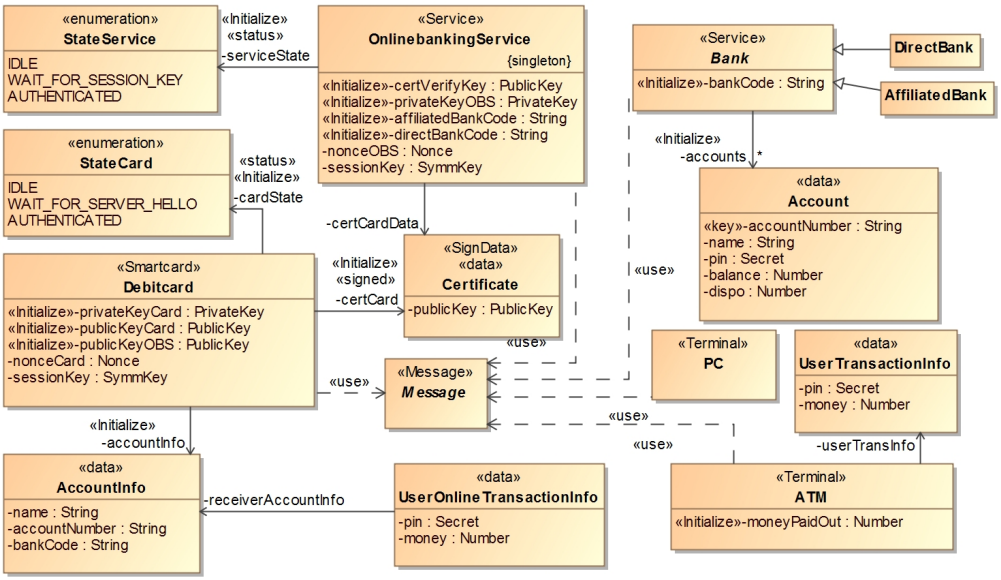

Abbildung 9.14: Klassendiagramm von Debitcard

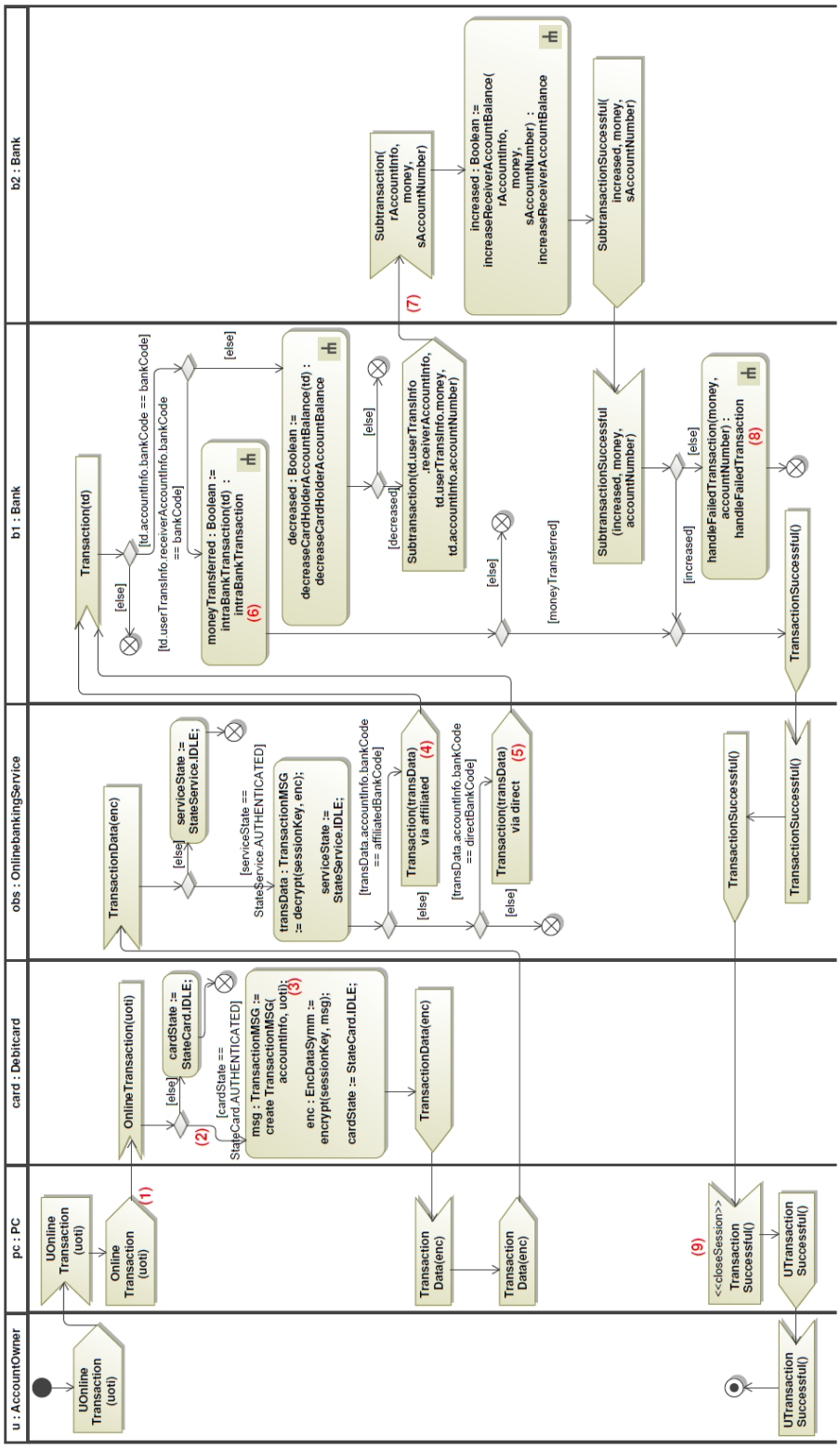

Abbildung 9.15: Protokoll zum Überweisen von Geld

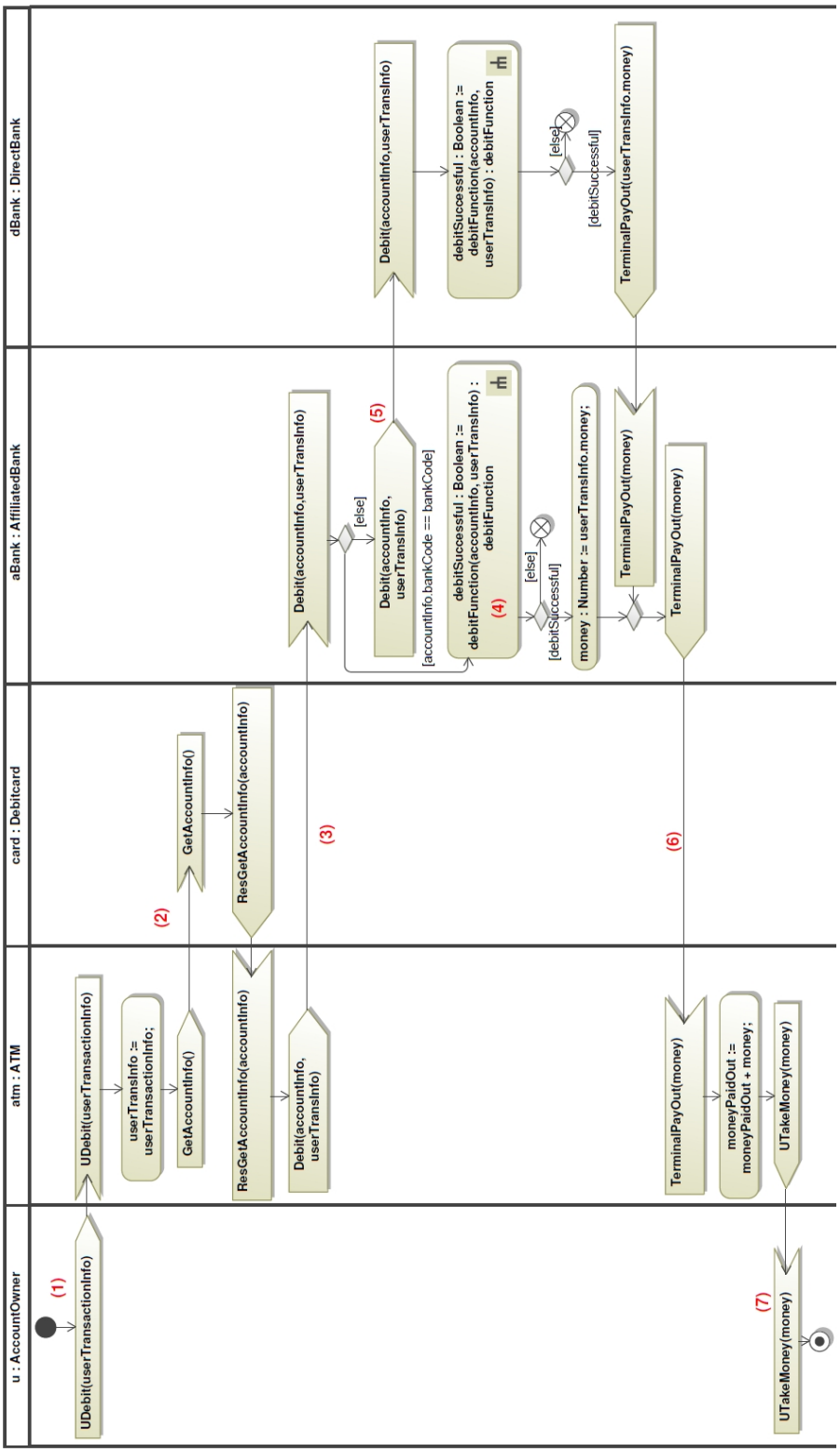

Abbildung 9.16: Protokoll zum Abheben von Geld an einem Bankautomaten

OAuth

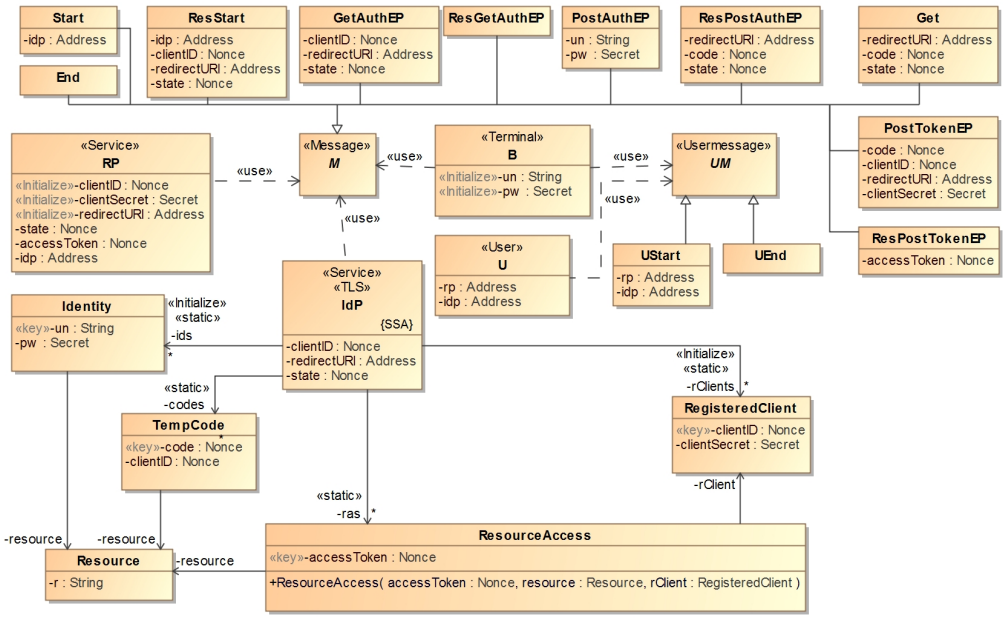

Abbildung 9.17: Klassendiagramm der OAuth-Anwendung

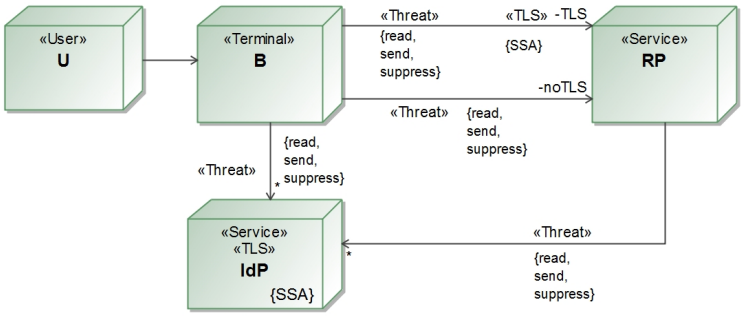

Abbildung 9.18: Deploymentdiagramm der OAuth-Anwendung

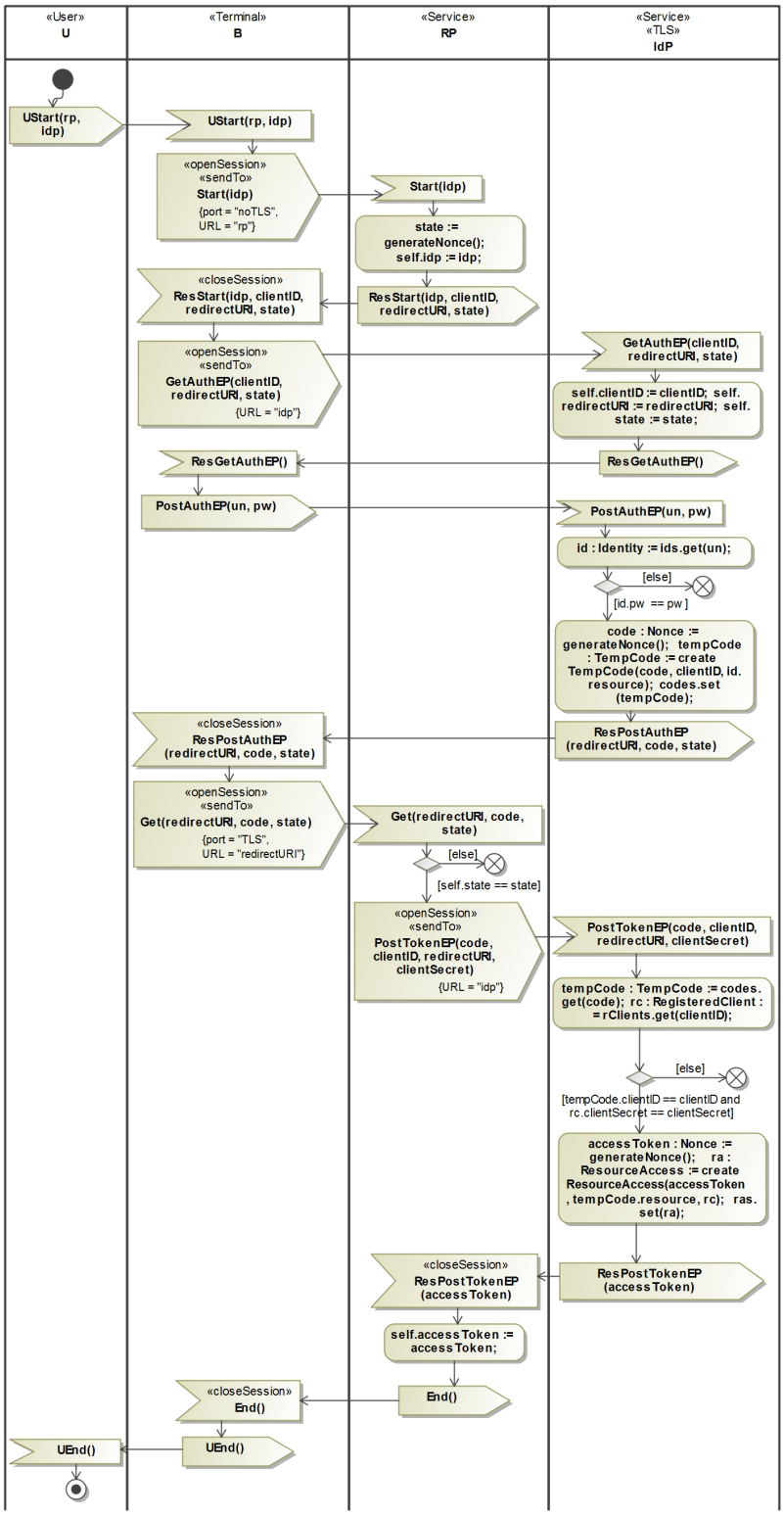

Abbildung 9.19: Authorisierungsprozess

9.2 Ausführbarer Code

```
@WebMethod
public synchronized BuyTicketSuccessful
 processBuyTicket(BuyTicket inmsg)
  throws ServiceException {

 synchronized(manager){
  try{
   EncDataSymm enc = inmsg.getMsg();

   if (state == StateService.getAUTHENTICATED()) {

    BuyInfo msg = (BuyInfo) (EncDataSymm.decrypt(sessionKey,
      enc));

    if (pay(msg.getCcInfo(), msg.getTicketInfo())) {

     Nonce n = Nonce.generateNonce();
     ID ticketID = new ID(n);
     Ticket ticket = new Ticket(ticketID,
       msg.getTicketInfo(), iStamp);
     CardTicket cardTicket = new CardTicket(
            msg.getCardID(),
            msg.getTransactionId(),
            ticket);
     tickets.set(cardTicket);
     issuedTicketIDs.add(ticketID);

     return new BuyTicketSuccessful();

    } else {
     stop("FlowFinal");
     return null;
    }

   } else {
    stop("FlowFinal");
    return null;
   }
  } catch (ServiceException e) {
   ExceptionService();
   throw e;
  }
 }
}
```

Listing 9.1: Servicecode zum Kaufen eines Tickets

```
@WebMethod
public synchronized TransferResult processTransfer(ProcessTransfer t)
  throws ServiceException {
 synchronized(manager){
  TransferResult tr = new TransferResult();
  if (accounts.containsKey(t.getAccountNumber())) {
   setAccount(accounts.get(t.getAccountNumber()));
   if (account.getPw().equals(t.getPw())) {
    if (bankCodes.containsKey(t.getReceiverAccount().
      getBankCode())){
     boolean debited = debit(account.getAccountNumber(),
       t.getMoney());
     if (debited) {
      Nonce n = Nonce.generateNonce();
      AccountInfo fromAccount = new AccountInfo(
        account.getName(), account.getAccountNumber(),
        bankAuthToken.getData().getBankCode());
      Transfer transfer = new Transfer(t.getMoney(),
        fromAccount, t.getReceiverAccount());
      setUt(new UniqueTransaction(n, false, true, transfer));
      iTransactions.set(ut);

      return processTransactionResult(
       (TransactionResult) sendMsg(
        new Transaction(ut, bankAuthToken),
        Ports.Bank2Bank_default,
        bankCodes.get(
          ut.getTransfer().getTo().getBankCode())
         .getAddress(), true, true)
      );
     } else {
```

```
32          stop("insufficient funds");
33          return null;
34         }
35        } else {
36         stop("unkown bank code");
37         return null;
38        }
39       } else {
40        stop("authentication error");
41        return null;
42       }
43      } else {
44       stop("authentication error");
45       return null;
46      }
47     }
48    }
```

Listing 9.2: Servicecode zur internen Ausführung einer Überweisung

9.3 ASLan++ Spezifikationen

```
specification eTicket
channel_model ICM
entity Environment
{
 types
  number < nat;
  string < text;
  boolean < nat;
  nonce < nat;
  secret < text;
  class < message;
  enum < nat;
  cCInfo < class;
  ticketInfo < class;
  dateTime < class;
 symbols
  init : fact;
  succ(nat) : nat;
  isKeyPair(private_key, public_key) : fact;

  %Boolean constants
  t, f : boolean;

  %agents
  eticketservice, eticketcard, cardowner : agent;

  %Message constants
  uRetrieveTicket, reqRetrieveTicketInstruction,
  retrieveTicket, ticketReceived, uRetrieveTicketSuccessful,
  resRetrieveTicket, resReqRetrieveTicketInstruction,
  retrieveTicketSuccessful, uBuyTicket,
  resReqBuyTicketInstruction, buyTicketSuccessful,
  reqBuyTicketInstruction, uBuyTicketSuccessful,
  buyTicket, uHandshake, handshake, clientHello,
  serverHello, sessionKey, uHandshakeSuccessful,
  handshakeSuccessful : message;

  %Class constants
  ticket, certificate, cardTicket, buyInfo, sessionKeyMsg,
  serverHelloMsg, clientHelloMsg, retrieveInfo, ticketData,
  stamp, inspectorDevice, inspector, ticketReceivedData,
  stampTicketMsg, iD, stampTicketSuccessfulMsg, attacker,
  initializeConstraints, applicationConstraints, helperConstraints,
  certificateData, returnTicketInspectorMsg, showTicketInspectorMsg : class;

  %Enum Literals
  statecard_idle, statecard_authenticated, statecard_authenticated2,
  stateservice_idle, stateservice_authenticated, stateterminal_idle,
  stateterminal_authenticated2 : enum;

  %Initialized Attributes
  ETicketService_signKey : private_key;
  ETicketService_certVerifyKey : public_key;
  ETicketService_privateKeyService : private_key;
  ETicketService_state : enum;
  ETicketService_iStamp_stampTime : dateTime;
  ETicketService_iStamp_stamped : boolean;
  eTicketService_tickets(agent):
   nonce*nonce*nonce*ticketInfo*boolean*dateTime set;
  eTicketService_issuedTicketIDs(agent): nat*nonce set;
  ETicketCard_privateKeyCard : private_key;
  ETicketCard_certCard_data_publicKey : public_key;
  ETicketCard_certCard_signedCert : message;
  ETicketCard_publicKeyService : public_key;
  ETicketCard_certVerifyKey : public_key;
  ETicketCard_ccInfo : cCInfo;
  ETicketCard_id_id : nonce;
  ETicketCard_pin : secret;
  ETicketCard_state : enum;
  eTicketCard_tickets(agent):
   nonce*ticketInfo*boolean*dateTime set;
  eTicketCard_deletedTicketIDs(agent): nat*nonce set;
  CardOwner_pin : secret;
  CardOwner_ticketInfo : ticketInfo;


 entity Session(CardOwner : agent, ETicketCard : agent,
  ETicketService : agent, CardOwner_pin : secret,
  CardOwner_ticketInfo : ticketInfo, ETicketCard_privateKeyCard : private_key,
```

```
80    ETicketCard_certCard_data_publicKey : public_key,
81    ETicketCard_certCard_signedCert : message,
82    ETicketCard_publicKeyService : public_key,
83    ETicketCard_certVerifyKey : public_key, ETicketCard_ccInfo : cCInfo,
84    ETicketCard_id_id : nonce, ETicketCard_pin : secret, ETicketCard_state : enum,
85    ETicketService_signKey : private_key, ETicketService_certVerifyKey : public_key,
86    ETicketService_privateKeyService : private_key, ETicketService_state : enum,
87    ETicketService_iStamp_stampTime : dateTime,
88    ETicketService_iStamp_stamped : boolean)
89   {
90
91    entity ETicketService (Actor : agent, ETicketCard : agent,
92     CardOwner : agent, SignKey : private_key, CertVerifyKey : public_key,
93     PrivateKeyService : private_key, State : enum, IStamp_stampTime : dateTime,
94     IStamp_stamped : boolean)
95    {
96     symbols
97     %Class variables
98     SessionKey : symmetric_key;
99     NonceService : nonce;
100    NonceDeleteTicket : nonce;
101    %Local variables
102    L_ticketReceived_trd_cardId_id : nonce;
103    L_ticketReceived_trd_nonceDeleteTicket : nonce;
104    L_retrieveTicket_rData_ticket_id_id : nonce;
105    L_retrieveTicket_rData_ticket_ticketInfo : ticketInfo;
106    L_retrieveTicket_rData_ticket_stamp_stampTime : dateTime;
107    L_retrieveTicket_rData_ticket_stamp_stamped : boolean;
108    L_retrieveTicket_rData_transactionId : nonce;
109    L_retrieveTicket_rData_nonceDeleteTicket : nonce;
110    L_retrieveTicket_resEnc : message;
111    L_retrieveTicket_msg_cardId_id : nonce;
112    L_retrieveTicket_msg_transactionId : nonce;
113    L_retrieveTicket_cardTicket_id_id : nonce;
114    L_retrieveTicket_cardTicket_ticket_id_id : nonce;
115    L_retrieveTicket_cardTicket_ticket_ticketInfo : ticketInfo;
116    L_retrieveTicket_cardTicket_ticket_stamp_stampTime : dateTime;
117    L_retrieveTicket_cardTicket_ticket_stamp_stamped : boolean;
118    L_retrieveTicket_cardTicket_transactionId : nonce;
119    L_buyTicket_msg_ccInfo : cCInfo;
120    L_buyTicket_msg_ticketInfo : ticketInfo;
121    L_buyTicket_msg_cardID_id : nonce;
122    L_buyTicket_msg_transactionId : nonce;
123    L_buyTicket_n : nonce;
124    L_buyTicket_ticketID_id : nonce;
125    L_buyTicket_ticket_id_id : nonce;
126    L_buyTicket_ticket_ticketInfo : ticketInfo;
127    L_buyTicket_ticket_stamp_stampTime : dateTime;
128    L_buyTicket_ticket_stamp_stamped : boolean;
129    L_buyTicket_cardTicket_id_id : nonce;
130    L_buyTicket_cardTicket_ticket_id_id : nonce;
131    L_buyTicket_cardTicket_ticket_ticketInfo : ticketInfo;
132    L_buyTicket_cardTicket_ticket_stamp_stampTime : dateTime;
133    L_buyTicket_cardTicket_ticket_stamp_stamped : boolean;
134    L_buyTicket_cardTicket_transactionId : nonce;
135    L_clientHello_chm_nonceCard : nonce;
136    L_clientHello_shm_nonceService : nonce;
137    L_clientHello_shm_nonceCard : nonce;
138    L_clientHello_enc1 : message;
139    L_sessionKey_skm_nonceService : nonce;
140    L_sessionKey_skm_sessionKey : symmetric_key;
141    %Message variables
142    M_ticketReceived_enc : message;
143    M_retrieveTicket_enc : message;
144    M_buyTicket_enc : message;
145    M_clientHello_enc : message;
146    M_clientHello_certCard_data_publicKey : public_key;
147    M_clientHello_certCard_signedCert : message;
148    M_sessionKey_enc : message;
149    M_decryptKey : public_key;
150    %Counter variables
151    IssuedTicketIDsCounter : nat;
152    body
153    {
154     IssuedTicketIDsCounter := 0;
155     while (true)
156     {
157      select
158      {
159       on(ETicketCard -> Actor : clientHello.(crypt(?M_decryptKey,
160         clientHelloMsg.(?L_clientHello_chm_nonceCard)).certificate.(
161           sign(inv(CertVerifyKey),certificateData.(
162            ?M_clientHello_certCard_data_publicKey))
```

```
163         )) & isKeyPair(PrivateKeyService, M_decryptKey)) :
164        {
165         NonceService := fresh();
166         L_clientHello_shm_nonceService := NonceService;
167         L_clientHello_shm_nonceCard := L_clientHello_chm_nonceCard;
168         L_clientHello_enc1 := crypt(M_clientHello_certCard_data_publicKey,
169           serverHelloMsg.(L_clientHello_shm_nonceService.
170                 L_clientHello_shm_nonceCard));
171         Actor -> ETicketCard : serverHello.(L_clientHello_enc1);
172         select
173         {
174          on(ETicketCard -> Actor : sessionKey.(crypt(?M_decryptKey, sessionKeyMsg.(
175             ?L_sessionKey_skm_nonceService.?L_sessionKey_skm_sessionKey)))
176             & isKeyPair(PrivateKeyService, M_decryptKey)) :
177          {
178           goal_Secret :(SessionKey) := L_sessionKey_skm_sessionKey;
179           State := stateservice_authenticated;
180           Actor -> CardOwner : uHandshakeSuccessful;
181           select
182           {
183            on(ETicketCard -> Actor : retrieveTicket.(scrypt(SessionKey,
184             retrieveInfo.(iD.(?L_retrieveTicket_msg_cardId_id).
185              ?L_retrieveTicket_msg_transactionId)))
186            & contains(eTicketService_tickets(Actor),
187             (L_retrieveTicket_msg_cardId_id,
188             ?L_retrieveTicket_cardTicket_transactionId,
189             ?L_retrieveTicket_cardTicket_ticket_id_id,
190             ?L_retrieveTicket_cardTicket_ticket_ticketInfo,
191             ?L_retrieveTicket_cardTicket_ticket_stamp_stamped,
192             ?L_retrieveTicket_cardTicket_ticket_stamp_stampTime))) :
193            {
194             L_retrieveTicket_cardTicket_id_id := L_retrieveTicket_msg_cardId_id;
195             NonceDeleteTicket := fresh();
196             L_retrieveTicket_rData_ticket_id_id :=
197              L_retrieveTicket_cardTicket_ticket_id_id;
198             L_retrieveTicket_rData_ticket_ticketInfo :=
199              L_retrieveTicket_cardTicket_ticket_ticketInfo;
200             L_retrieveTicket_rData_ticket_stamp_stamped :=
201              L_retrieveTicket_cardTicket_ticket_stamp_stamped;
202             L_retrieveTicket_rData_ticket_stamp_stampTime :=
203              L_retrieveTicket_cardTicket_ticket_stamp_stampTime;
204             L_retrieveTicket_rData_transactionId :=
205              L_retrieveTicket_msg_transactionId;
206             L_retrieveTicket_rData_nonceDeleteTicket := NonceDeleteTicket;
207
208             L_retrieveTicket_resEnc := scrypt(SessionKey, ticketData.(
209              ticket.(iD.(L_retrieveTicket_rData_ticket_id_id).
210              L_retrieveTicket_rData_ticket_ticketInfo.stamp.(
211              L_retrieveTicket_rData_ticket_stamp_stamped.
212              L_retrieveTicket_rData_ticket_stamp_stampTime)).
213              L_retrieveTicket_rData_transactionId.
214              L_retrieveTicket_rData_nonceDeleteTicket)
215             );
216
217             Actor -> ETicketCard : resRetrieveTicket.(L_retrieveTicket_resEnc);
218             select
219             {
220              on(ETicketCard -> Actor : ticketReceived.(scrypt(SessionKey,
221               ticketReceivedData.(iD.(?L_ticketReceived_trd_cardId_id).
222               NonceDeleteTicket)))) :
223              {
224               L_ticketReceived_trd_nonceDeleteTicket := NonceDeleteTicket;
225               if (contains( eTicketService_tickets(Actor),
226                  (L_ticketReceived_trd_cardId_id, ?, ?, ?, ?, ?)))
227               {
228                remove(eTicketService_tickets(Actor),
229                   (L_ticketReceived_trd_cardId_id, ?, ?, ?, ?, ?));
230               }
231               State := stateservice_idle;
232               Actor -> CardOwner : uRetrieveTicketSuccessful;
233              }
234              on(true) :
235              {
236               State := stateservice_idle;
237              }
238             }
239            }
240            on(ETicketCard -> Actor : buyTicket.(scrypt(SessionKey, buyInfo.(
241             ?L_buyTicket_msg_ccInfo.?L_buyTicket_msg_ticketInfo.iD.(
242             ?L_buyTicket_msg_cardID_id).?L_buyTicket_msg_transactionId))) & true) :
243            {
244             L_buyTicket_n := fresh();
245             L_buyTicket_ticketID_id := L_buyTicket_n;
```

```
246             L_buyTicket_ticket_id_id := L_buyTicket_ticketID_id;
247             L_buyTicket_ticket_ticketInfo := L_buyTicket_msg_ticketInfo;
248             L_buyTicket_ticket_stamp_stamped := IStamp_stamped;
249             L_buyTicket_ticket_stamp_stampTime := IStamp_stampTime;
250             L_buyTicket_cardTicket_id_id := L_buyTicket_msg_cardID_id;
251             L_buyTicket_cardTicket_transactionId := L_buyTicket_msg_transactionId;
252             L_buyTicket_cardTicket_ticket_id_id := L_buyTicket_ticket_id_id;
253             L_buyTicket_cardTicket_ticket_ticketInfo :=
254                 L_buyTicket_ticket_ticketInfo;
255             L_buyTicket_cardTicket_ticket_stamp_stamped :=
256                 L_buyTicket_ticket_stamp_stamped;
257             L_buyTicket_cardTicket_ticket_stamp_stampTime :=
258                 L_buyTicket_ticket_stamp_stampTime;
259             if (contains( eTicketService_tickets(Actor),
260                 (L_buyTicket_cardTicket_id_id, ?, ?, ?, ?, ?)))
261             {
262              remove(eTicketService_tickets(Actor),
263                    (L_buyTicket_cardTicket_id_id, ?, ?, ?, ?, ?));
264             }
265             add(eTicketService_tickets(Actor),
266               (L_buyTicket_cardTicket_id_id, L_buyTicket_cardTicket_transactionId,
267                L_buyTicket_cardTicket_ticket_id_id,
268                L_buyTicket_cardTicket_ticket_ticketInfo,
269                L_buyTicket_cardTicket_ticket_stamp_stamped,
270                L_buyTicket_cardTicket_ticket_stamp_stampTime
271               )
272             );
273             add(eTicketService_issuedTicketIDs(Actor), (IssuedTicketIDsCounter,
274              L_buyTicket_ticketID_id));
275             IssuedTicketIDsCounter := succ(IssuedTicketIDsCounter);
276             Actor -> CardOwner : uBuyTicketSuccessful;
277            }
278            on(true) :
279            {
280             State := stateservice_idle;
281            }
282           }
283          }
284          on(true) :
285          {
286           State := stateservice_idle;
287          }
288  } } } } } }
289
290
291  entity ETicketCard (ETicketService : agent, Actor : agent, CardOwner : agent,
292   PrivateKeyCard : private_key, CertCard_data_publicKey : public_key,
293   CertCard_signedCert : message, PublicKeyService : public_key,
294   CertVerifyKey : public_key, CcInfo : cCInfo, Id_id : nonce,
295   Pin : secret, State : enum
296  ){
297   symbols
298   %Class variables
299   PublicKeyInspectorDevice : public_key;
300   SessionKey : symmetric_key;
301   NonceCard : nonce;
302   TransactionId : nonce;
303   WaitingForTicket : boolean;
304   %Local variables
305   L_resRetrieveTicket_trd_cardId_id : nonce;
306   L_resRetrieveTicket_trd_nonceDeleteTicket : nonce;
307   L_resRetrieveTicket_enc : message;
308   L_reqRetrieveTicketInstruction_rInfo_cardId_id : nonce;
309   L_reqRetrieveTicketInstruction_rInfo_transactionId : nonce;
310   L_reqRetrieveTicketInstruction_enc : message;
311   L_resRetrieveTicket_msg_ticket_id_id : nonce;
312   L_resRetrieveTicket_msg_ticket_ticketInfo : ticketInfo;
313   L_resRetrieveTicket_msg_ticket_stamp_stampTime : dateTime;
314   L_resRetrieveTicket_msg_ticket_stamp_stamped : boolean;
315   L_resRetrieveTicket_msg_transactionId : nonce;
316   L_resRetrieveTicket_msg_nonceDeleteTicket : nonce;
317   L_reqBuyTicketInstruction_msg_ccInfo : cCInfo;
318   L_reqBuyTicketInstruction_msg_ticketInfo : ticketInfo;
319   L_reqBuyTicketInstruction_msg_cardID_id : nonce;
320   L_reqBuyTicketInstruction_msg_transactionId : nonce;
321   L_reqBuyTicketInstruction_enc : message;
322   L_serverHello_skm_nonceService : nonce;
323   L_serverHello_skm_sessionKey : symmetric_key;
324   L_serverHello_enc : message;
325   L_serverHello_shm_nonceService : nonce;
326   L_serverHello_shm_nonceCard : nonce;
327   L_handshake_chm_nonceCard : nonce;
328   L_handshake_enc : message;
```

```
329   %Message variables
330   M_resRetrieveTicket_resEnc : message;
331   M_reqBuyTicketInstruction_ticketInfo : ticketInfo;
332   M_reqBuyTicketInstruction_pin : secret;
333   M_serverHello_enc1 : message;
334   M_decryptKey : public_key;
335   %Counter variables
336   DeletedTicketIDsCounter : nat;
337   body
338   {
339    DeletedTicketIDsCounter := 0 ;
340    WaitingForTicket := f;
341    while (true)
342    {
343     select
344     {
345      on(CardOwner -> Actor : handshake) :
346      {
347       NonceCard := fresh();
348       L_handshake_chm_nonceCard := NonceCard;
349       L_handshake_enc := crypt(PublicKeyService,
350         clientHelloMsg.(L_handshake_chm_nonceCard));
351       Actor -> ETicketService : clientHello.(L_handshake_enc.
352             certificate.(CertCard_signedCert));
353       select
354       {
355        on(ETicketService -> Actor : serverHello.(
356         crypt(?M_decryptKey, serverHelloMsg.(
357          ?L_serverHello_shm_nonceService.NonceCard))) &
358          isKeyPair(PrivateKeyCard, M_decryptKey)
359        ) :{
360         L_serverHello_shm_nonceCard := NonceCard;
361         goal_Secret :(SessionKey) := fresh();
362         L_serverHello_skm_nonceService := L_serverHello_shm_nonceService;
363         L_serverHello_skm_sessionKey := SessionKey;
364         L_serverHello_enc := crypt(PublicKeyService, sessionKeyMsg.(
365           L_serverHello_skm_nonceService.L_serverHello_skm_sessionKey));
366         State := statecard_authenticated;
367         Actor -> ETicketService : sessionKey.(L_serverHello_enc);
368         select
369         {
370          on(CardOwner -> Actor : reqRetrieveTicketInstruction &
371            WaitingForTicket = t) :
372          {
373           L_reqRetrieveTicketInstruction_rInfo_cardId_id := Id_id;
374           L_reqRetrieveTicketInstruction_rInfo_transactionId :=
375            TransactionId;
376           L_reqRetrieveTicketInstruction_enc := scrypt(SessionKey,
377            retrieveInfo.(iD.(
378             L_reqRetrieveTicketInstruction_rInfo_cardId_id).
379             L_reqRetrieveTicketInstruction_rInfo_transactionId));
380           Actor -> ETicketService : retrieveTicket.(
381               L_reqRetrieveTicketInstruction_enc);
382           select
383           {
384            on(ETicketService -> Actor : resRetrieveTicket.(
385             scrypt(SessionKey, ticketData.(
386             ticket.(iD.(?L_resRetrieveTicket_msg_ticket_id_id).
387             ?L_resRetrieveTicket_msg_ticket_ticketInfo.stamp.(
388             ?L_resRetrieveTicket_msg_ticket_stamp_stamped.
389             ?L_resRetrieveTicket_msg_ticket_stamp_stampTime)).
390             ?L_resRetrieveTicket_msg_transactionId.
391             ?L_resRetrieveTicket_msg_nonceDeleteTicket)))
392            ) :{
393             if (contains( eTicketCard_tickets(Actor),
394               (L_resRetrieveTicket_msg_ticket_id_id, ?, ?, ?)))
395             {
396              remove(eTicketCard_tickets(Actor),
397                (L_resRetrieveTicket_msg_ticket_id_id, ?, ?, ?));
398             }
399             add(eTicketCard_tickets(Actor),
400               (L_resRetrieveTicket_msg_ticket_id_id,
401                L_resRetrieveTicket_msg_ticket_ticketInfo,
402                L_resRetrieveTicket_msg_ticket_stamp_stamped,
403                L_resRetrieveTicket_msg_ticket_stamp_stampTime));
404             L_resRetrieveTicket_trd_cardId_id := Id_id;
405             L_resRetrieveTicket_trd_nonceDeleteTicket :=
406                 L_resRetrieveTicket_msg_nonceDeleteTicket;
407             L_resRetrieveTicket_enc := scrypt(SessionKey, ticketReceivedData.(
408              iD.(L_resRetrieveTicket_trd_cardId_id).
409              L_resRetrieveTicket_trd_nonceDeleteTicket));
410             State := statecard_idle;
411             WaitingForTicket := f;
```

```
412             Actor -> ETicketService : ticketReceived.(L_resRetrieveTicket_enc);
413            }
414            on(true) :
415            {
416             State := statecard_idle;
417            }
418           }
419          }
420          on(CardOwner -> Actor : reqBuyTicketInstruction.(
421            ?M_reqBuyTicketInstruction_ticketInfo.Pin) &
422            (WaitingForTicket = f )) :
423          {
424           WaitingForTicket := t;
425           TransactionId := fresh();
426           L_reqBuyTicketInstruction_msg_ccInfo := CcInfo;
427           L_reqBuyTicketInstruction_msg_ticketInfo :=
428               M_reqBuyTicketInstruction_ticketInfo;
429           L_reqBuyTicketInstruction_msg_cardID_id := Id_id;
430           L_reqBuyTicketInstruction_msg_transactionId := TransactionId;
431           L_reqBuyTicketInstruction_enc := scrypt(SessionKey, buyInfo.(
432            L_reqBuyTicketInstruction_msg_ccInfo.
433            L_reqBuyTicketInstruction_msg_ticketInfo.iD.(
434            L_reqBuyTicketInstruction_msg_cardID_id).
435            L_reqBuyTicketInstruction_msg_transactionId)
436           );
437           Actor -> ETicketService : buyTicket.(L_reqBuyTicketInstruction_enc);
438          }
439          on(true) :
440          {
441           State := statecard_idle;
442          }
443         }
444        }
445        on(true) :
446        {
447         State := statecard_idle;
448        }
449  } } } } } }
450
451
452  entity CardOwner (ETicketService : agent, ETicketCard : agent,
453    Actor : agent, Pin : secret, TicketInfo : ticketInfo)
454  {
455   symbols
456   %User variables
457   %Class variables
458   Id_id : nonce;
459   M_decryptKey : public_key;
460   body
461   {
462    Actor -> ETicketCard : reqRetrieveTicketInstruction;
463    Actor -> ETicketCard : reqBuyTicketInstruction.(TicketInfo.Pin);
464    Actor -> ETicketCard : handshake;
465   }
466  }
467  body
468  {
469   new ETicketService(ETicketService, ETicketCard, CardOwner,
470   ETicketService_signKey, ETicketService_certVerifyKey,
471   ETicketService_privateKeyService, ETicketService_state,
472   ETicketService_iStamp_stampTime, ETicketService_iStamp_stamped );
473
474   new ETicketCard(ETicketService, ETicketCard, CardOwner,
475   ETicketCard_privateKeyCard, ETicketCard_certCard_data_publicKey,
476   ETicketCard_certCard_signedCert, ETicketCard_publicKeyService,
477   ETicketCard_certVerifyKey, ETicketCard_ccInfo,
478   ETicketCard_id_id, ETicketCard_pin, ETicketCard_state );
479
480   new CardOwner(ETicketService, ETicketCard, CardOwner, CardOwner_pin,
481      CardOwner_ticketInfo );
482  }
483 }
484 body
485 {
486  ETicketCard_publicKeyService := fresh();
487  ETicketCard_certCard_data_publicKey := fresh();
488  ETicketService_certVerifyKey :=fresh();
489  ETicketCard_certVerifyKey := ETicketService_certVerifyKey;
490  ETicketService_privateKeyService := inv(ETicketCard_publicKeyService);
491  ETicketCard_privateKeyCard := inv(ETicketCard_certCard_data_publicKey);
492  ETicketService_signKey := inv(ETicketService_certVerifyKey);
493  isKeyPair(ETicketService_privateKeyService,ETicketCard_publicKeyService);
494  isKeyPair(ETicketCard_privateKeyCard,ETicketCard_certCard_data_publicKey);
```

```
495   isKeyPair(ETicketService_signKey,ETicketService_certVerifyKey);
496   ETicketCard_certCard_signedCert :=
497     sign(ETicketService_signKey,certificateData.
498       (ETicketCard_certCard_data_publicKey));
499   ETicketCard_state := statecard_idle;
500   ETicketService_state := stateservice_idle;
501
502   CardOwner_pin := fresh();
503   CardOwner_ticketInfo := fresh();
504
505   ETicketCard_id_id := fresh();
506   ETicketCard_ccInfo := fresh();
507   ETicketCard_pin := CardOwner_pin;
508
509   ETicketService_iStamp_stampTime := fresh();
510   ETicketService_iStamp_stamped := f;
511
512   iknows(ETicketCard_certCard_data_publicKey);
513   iknows(ETicketCard_certCard_signedCert);
514   iknows(ETicketCard_publicKeyService);
515   iknows(ETicketCard_certVerifyKey);
516   iknows(ETicketService_certVerifyKey);
517   iknows(ETicketCard_id_id);
518
519   new Session(cardowner, eticketcard, eticketservice, CardOwner_pin,
520     CardOwner_ticketInfo, ETicketCard_privateKeyCard,
521     ETicketCard_certCard_data_publicKey,
522     ETicketCard_certCard_signedCert, ETicketCard_publicKeyService,
523     ETicketCard_certVerifyKey, ETicketCard_ccInfo, ETicketCard_id_id,
524     ETicketCard_pin, ETicketCard_state, ETicketService_signKey,
525     ETicketService_certVerifyKey, ETicketService_privateKeyService,
526     ETicketService_state, ETicketService_iStamp_stampTime,
527     ETicketService_iStamp_stamped);
528  }
529  goals
530  %a ticket will not be lost
531  inv2: [](forall X0 ID . (exists X1 X2 X3 X4 X5 X6 . (
532    contains(eTicketService_issuedTicketIDs(eticketservice), (X0,ID)) =>
533     (contains(eTicketService_tickets(eticketservice), (X1, X6 ,ID,X2,X3,X4)) |
534      contains(eTicketCard_deletedTicketIDs(eticketcard), (X5,ID)) |
535      contains(eTicketCard_tickets(eticketcard), (ID,X2,X3,X4))
536     )
537    )));
538 }
```

Listing 9.3: Generierte ASlan++ Spezifikation der in Abbildung 4.22 eingeschränkten ETicket-Anwendung

9.4 Formale Spezifikationen

```
predicates
 connectByAttacker-possible : string;

axioms
 connectByAttacker-possible
 : connectByAttacker-possible(address)
 ↔ serviceClassName(address) = Bank
```

Listing 9.4: Spezifikation von *connectionByAttacker-possible* am Beispiel der Onlinebanking-Anwendung

```
procedures
 CONNECTTOSERVICE agent × ports × agent × port : connectionset,
   (agent → ports → messagelist) nonfunctional indeterministic;

declaration
 CONNECTTOSERVICE
 : CONNECTTOSERVICE (ag, outport, ag0; connections, inputs)
       {
  if ag ≠ ATTACKER then {
   choose conn
    with connected(ag ⊙ outport ,connections)
    in {
     connections := connections -- conn ;
     inputs(ag)(outport) := [] ;
    } ifnone skip
  }
  choose conn
   with serviceConn-ok(conn) ∧
     conn.endpoint1.agent = ag ∧
     conn.endpoint1.port = outport ∧
     conn.endpoint2.agent = ag0
   in {
     connections := connections ++ conn ;
   } ifnone skip
  };
```

Listing 9.5: Spezifikation von CONNECTTOSERVICE

```
predicates
 serviceConn-ok : connection;

axioms
 serviceConn-ok :
 serviceConn-ok(conn) ↔
  serviceEndpoint-ok(conn.endpoint2) ∧
  (
   (conn.endpoint1.port = AccountOwnerPC2Bank_default ∧
    (conn.endpoint2.port = Bank2AccountOwnerPCDefaultPort ∨
    conn.endpoint2.port = AttackerPort)) ∨
   (conn.endpoint1.port = Bank2Bank_default ∧
    (conn.endpoint2.port = Bank2BankDefaultPort ∨
    conn.endpoint2.port = AttackerPort)) ∨
   (conn.endpoint1.port = AttackerPort ∧
    (conn.endpoint2.port = Bank2AccountOwnerPCDefaultPort ∨
    conn.endpoint2.port = Bank2BankDefaultPort))
  );
```

Listing 9.6: Spezifikation von serviceConn-ok

```

enrich ConnectDisconnect with
  procedures
      SEND  message × ports × agent × connectionset : attackerdataset ×
                                                      (agent
                                                       → ports
                                                         → messagelist
                                                      ) nonfunctional indeterministic;
   variables
      inputs : agent → ports → messagelist;
      input : ports → messagelist;
      attacker-known : attackerdataset;
      outport, port0 : ports;
      outmsg : message;
```

```
15        ag0 : agent;
16
17  declaration
18     SEND
19     : SEND (outmsg, outport, ag, connections; attacker-known, inputs)
20         {
21      if (connected(ag ⊙ outport, connections)) then {
22          choose  conn with connected(ag ⊙ outport, conn, connections)
23                in let ag0 = other-endpoint(ag ⊙ outport, conn)
24                                    .agent,
25                       port0 = other-endpoint(ag ⊙ outport, conn)
26                                    .port
27                   in {
28                         if attacker-can-read(conn)
29                         then attacker-known := attacker-known ∫+ outmsg ;
30                         if not isAttacker(ag0)
31                         then inputs := add(ag0, port0, outmsg, inputs);
32                      } ifnone skip
33                }
34         };
35
36  end enrich
```

Listing 9.7: Spezifikation der Send-Prozedur

```
1   asm specification ASM
2   using ASMAUX
3
4   variables
5   account-Account-var : Account;
6   accountNumber-string-var : string;
7   bankAuthToken-AuthToken-var : AuthToken;
8   debited-bool-var : bool;
9   decreased-bool-var : bool;
10  fromAccount-AccountInfo-var : AccountInfo;
11  inc-bool-var : bool;
12  increased-bool-var : bool;
13  money-int-var : int;
14  n-Nonce-var : Nonce;
15  pw-Secret-var : Secret;
16  rAccount-AccountInfo-var : AccountInfo;
17  success-bool-var : bool;
18  transfer-Transfer-var : Transfer;
19  ut-UniqueTransaction-var : UniqueTransaction;
20  url-String-var : String;
21  stop : bool;
22  inmsg : message;
23
24  state variables
25   AccountOwner-accountNumber, AccountOwner-pw, UserDevice-_state_userdevice,
26   Bank-verifyKey, Bank-address, Bank-_state_bank, Bank-bankAuthToken,
27   Bank-bankCodes, Bank-ut, Bank-account, Shared-Bank-accounts,
28   Shared-Bank-iTransactions, Shared-Bank-cTransactions, Initial-Bank-verifyKey,
29   Initial-Bank-bankAuthToken, Initial-Bank-bankCodes, Initial-Bank-accounts,
30   next-BankAgentName, BankAddress, UserDevice-UserDevice2Bank_default-Address,
31   attackerAddress, all-nonces, next-nonce, inputs, connections, attacker-known,
32   stop;
33
34  initial state
35      init(AccountOwner-accountNumber, AccountOwner-pw, UserDevice-_state_userdevice,
36    Bank-verifyKey, Bank-address, Bank-_state_bank, Bank-bankAuthToken,
37    Bank-bankCodes, Bank-ut, Bank-account, Shared-Bank-accounts,
38    Shared-Bank-iTransactions, Shared-Bank-cTransactions, Initial-Bank-verifyKey,
39    Initial-Bank-bankAuthToken, Initial-Bank-bankCodes, Initial-Bank-accounts,
40    next-BankAgentName, BankAddress, UserDevice-UserDevice2Bank_default-Address,
41    attackerAddress, all-nonces, next-nonce, inputs, connections, attacker-known,
42    stop)
43
44  final state stop
45
46
47  asm rule STEP
48
49  declaration
50
51  STEP : STEP
52  {
53   let asm-step = [?], exception_occurred = false
54   in
55   {
56    if asm-step = connect then
57    {
```

```
58      CONNECT(; connections, inputs)
59     }
60     else if asm-step = disconnect then
61     {
62      DISCONNECT(; connections, inputs)
63     }
64     else if asm-step = attacker-agent-step then
65     {
66      ATTACKER(connections, attacker-known; inputs)
67     }
68     else if asm-step = user-agent-step then
69     {
70      choose ag with exUser(ag) in {
71       USER(ag, connections, all-nonces, next-nonce;
72          attacker-known, inputs)
73      } ifnone skip
74     }
75     else if asm-step = UserDevice-agent-step then
76     {
77      choose ag with exUserDevice(ag) in {
78       USERDEVICESTEP
79      } ifnone skip
80     }
81     else if asm-step = Bank-agent-step then
82     {
83      choose ag with exBank(ag) in {
84       BANKSTEP
85      } ifnone skip
86     }
87     else if asm-step = Bank-ProcessTransferTimeEvent-step then
88     {
89      ProcessTransferTimeEvent#
90     }
91    };
92     stop := [?];
93  };
94
95
96  USERDEVICESTEP : USERDEVICESTEP
97  {
98  choose port with is-valid-port(ag, port) and inputs(ag)(port) \neq [] in
99  {
100  let inmsg = inputs(ag)(port).first in {
101   inputs := rem(ag, port, inputs);
102   if (isTransferResult(inmsg)) then { TransferResult# } else
103   if (isUProcessTransfer(inmsg)) then { UProcessTransfer# } else
104   { STOPSTEP(; exception_occurred) }
105  }
106 } ifnone skip
107 };
108
109 BANKSTEP : BANKSTEP
110 {
111 choose port with is-valid-port(ag, port) and inputs(ag)(port) \neq [] in
112 {
113  let inmsg = inputs(ag)(port).first in {
114   inputs := rem(ag, port, inputs);
115   if (isProcessTransfer(inmsg)) then { ProcessTransfer# } else
116   if (isTransactionResult(inmsg)) then { TransactionResult# } else
117   if (isTransaction(inmsg)) then { Transaction# } else
118   { STOPSTEP(; exception_occurred) }
119  }
120 } ifnone skip
121 };
122
123
124 UProcessTransfer# : UProcessTransfer# {
125 let accountNumber-string-var = inmsg.accountNumber_string,
126  pw-Secret-var = inmsg.pw_Secret,
127  rAccount-AccountInfo-var = inmsg.receiverAccount_AccountInfo,
128  money-int-var = inmsg.money_int in {
129   if ((UserDevice-_state_userdevice(ag) = GENERATED_STATE_IDLE)) then {
130    UserDevice-_state_userdevice(ag) := GENERATED_STATE_TRANSFERRESULT;
131
132    let url-String-var = UserDevice-UserDevice2Bank_default-Address(ag);
133    in {
134     if (serviceClassName(url-String-var) = Bank) then {
135      let ag0 = Bank(next-BankAgentName) in {
136       BankAddress(ag0) := url-String-var;
137
138       Bank-verifyKey(ag0) := Initial-Bank-verifyKey(url-String-var);
139       Bank-bankAuthToken(ag0) := Initial-Bank-bankAuthToken(url-String-var);
140       Bank-bankCodes(ag0) := Initial-Bank-bankCodes(url-String-var);
```

```
141       Bank-accounts(ag0) := Initial-Bank-accounts(url-String-var);
142
143       next-BankAgentName := next-BankAgentName + 1;
144       CONNECTTOSERVICE(ag, Bank2Bank_default, ag0; connections, inputs);
145       SEND(mkProcessTransfer(accountNumber-string-var, pw-Secret-var,
146         rAccount-AccountInfo-var, money-int-var), UserDevice2Bank_default,
147         ag, connections; attacker-known, inputs);
148       }
149      }
150     }
151    }
152    else {
153     UserDevice-_state_userdevice(ag) := GENERATED_STATE_IDLE;
154     STOPSTEP(; exception_occurred);
155    };
156    }
157 };
158
159 TransferResult# : TransferResult# {
160 let success-bool-var = inmsg.success_bool in {
161  if ( UserDevice-_state_userdevice(ag) = GENERATED_STATE_TRANSFERRESULT) then {
162   UserDevice-_state_userdevice(ag) := GENERATED_STATE_IDLE;
163   SEND(mkUTransferResult(success-bool-var), UserDevice2AccountOwnerDefaultPort,
164     ag, connections; attacker-known, inputs);
165  }
166  else {
167   UserDevice-_state_userdevice(ag) := GENERATED_STATE_IDLE;
168   STOPSTEP(; exception_occurred);
169  };
170 }
171 };
172
173 ProcessTransfer# : ProcessTransfer# {
174 if ((Bank-_state_bank(ag) = GENERATED_STATE_IDLE)) then {
175  if (containsKey(Shared-Bank-accounts(BankAddress(ag)),
176   inmsg.accountNumber_string)) then
177  {
178   if (not containsKey(Shared-Bank-accounts(BankAddress(ag)),
179     inmsg.accountNumber_string)) then
180   {
181    Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
182    STOPSTEP(; exception_occurred);
183   }
184   else {
185    Bank-account(ag) :=
186     get(Shared-Bank-accounts(BankAddress(ag)), inmsg.accountNumber_string);
187    if ((Bank-account(ag).pw_Secret = inmsg.pw_Secret)) then {
188     if (containsKey(Bank-bankCodes(ag),
189      inmsg.receiverAccount_AccountInfo.bankCode_string)) then {
190      let debited-bool-var = false
191      in {
192       debit(Bank-account(ag).accountNumber_string,
193        inmsg.money_int, ag; debited-bool-var, Bank-verifyKey,
194        Bank-address, Bank-_state_bank, Bank-bankAuthToken,
195        Bank-bankCodes, Shared-Bank-accounts, Shared-Bank-iTransactions,
196        Shared-Bank-cTransactions, Bank-ut, Bank-account,
197        BankAddress, exception_occurred);
198       if (exception_occurred) then { skip }
199       else {
200        if (debited-bool-var) then {
201         let n-Nonce-var = mkNonce("") in {
202          generateNonce#(all-nonces; next-nonce, n-Nonce-var);
203          let fromAccount-AccountInfo-var =
204           mkAccountInfo(Bank-account(ag).name_string,
205            Bank-account(ag).accountNumber_string,
206            Bank-bankAuthToken(ag).data_AuthTokenData.bankCode_string
207           ) in {
208           let transfer-Transfer-var =
209            mkTransfer(inmsg.money_int, fromAccount-AccountInfo-var,
210             inmsg.receiverAccount_AccountInfo)
211           in {
212            Bank-ut(ag) := mkUniqueTransaction(n-Nonce-var, false,
213                      true, transfer-Transfer-var);
214            Shared-Bank-iTransactions(BankAddress(ag)) :=
215             set(Shared-Bank-iTransactions(BankAddress(ag)), Bank-ut(ag));
216            Bank-_state_bank(ag) := GENERATED_STATE_TRANSACTIONRESULT;
217
218            let url-String-var =
219             get(Bank-bankCodes(ag),
220              Bank-ut(ag).transfer_Transfer.to_AccountInfo.bankCode_string)
221              .address_string;
222            in {
223             if (attackerAddress = url-String-var) then {
```

```
224              CONNECTTOSERVICE(ag, Bank2Bank_default, ATTACKER; connections, inputs);
225              SEND(mkTransaction(Bank-ut(ag), Bank-bankAuthToken(ag)),
226                Bank2Bank_default, ag, connections; attacker-known, inputs);
227             }
228             else{
229              if (serviceClassName(url-String-var) = Bank) then {
230              let ag0 = Bank(next-BankAgentName) in {
231               BankAddress(ag0) := url-String-var;
232
233               Bank-verifyKey(ag0) := Initial-Bank-verifyKey(url-String-var);
234               Bank-bankAuthToken(ag0) := Initial-Bank-bankAuthToken(url-String-var);
235               Bank-bankCodes(ag0) := Initial-Bank-bankCodes(url-String-var);
236               Bank-accounts(ag0) := Initial-Bank-accounts(url-String-var);
237
238               next-BankAgentName := next-BankAgentName + 1;
239               CONNECTTOSERVICE(ag, Bank2Bank_default, ag0; connections, inputs);
240               SEND(mkTransaction(Bank-ut(ag), Bank-bankAuthToken(ag)),
241                 Bank2Bank_default, ag, connections; attacker-known, inputs);
242              }
243             }
244            }
245           }
246          }
247         }
248        }
249        else {
250         Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
251         STOPSTEP(; exception_occurred);
252        };
253       }
254      }
255     }
256     else {
257      Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
258      STOPSTEP(; exception_occurred);
259     };
260    }
261    else {
262     Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
263     STOPSTEP(; exception_occurred);
264    };
265   }
266  }
267  else {
268   Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
269   STOPSTEP(; exception_occurred);
270  };
271 }
272 else {
273  Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
274  STOPSTEP(; exception_occurred);
275 };
276 };
277
278 TransactionResult# : TransactionResult# {
279 let success-bool-var = inmsg.increased_bool in {
280  if (Bank-_state_bank(ag) = GENERATED_STATE_TRANSACTIONRESULT) then {
281   if (containsKey(Shared-Bank-iTransactions(BankAddress(ag)),
282    Bank-ut(ag).n_Nonce)) then
283   {
284    if (success-bool-var) then {
285     Shared-Bank-iTransactions(BankAddress(ag)) :=
286      removeKey(Shared-Bank-iTransactions(BankAddress(ag)), Bank-ut(ag).n_Nonce);
287     Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
288     SEND(mkTransferResult(success-bool-var), Bank2UserDeviceDefaultPort, ag,
289      connections; attacker-known, inputs);
290    }
291    else {
292     let inc-bool-var = false
293     in {
294      credit(Bank-ut(ag).transfer_Transfer.from_AccountInfo.accountNumber_string,
295        Bank-ut(ag).transfer_Transfer.money_int, ag; inc-bool-var,
296        Bank-verifyKey, Bank-address, Bank-_state_bank, Bank-bankAuthToken,
297        Bank-bankCodes, Shared-Bank-accounts, Shared-Bank-iTransactions,
298        Shared-Bank-cTransactions, Bank-ut, Bank-account, BankAddress,
299        exception_occurred);
300      if (exception_occurred) then { skip }
301      else {
302       Shared-Bank-iTransactions(BankAddress(ag)) :=
303        removeKey(Shared-Bank-iTransactions(BankAddress(ag)),
304         Bank-ut(ag).n_Nonce);
305       Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
306
```

```
307         SEND(mkTransferResult(success-bool-var), Bank2UserDeviceDefaultPort, ag,
308           connections; attacker-known, inputs);
309         }
310        }
311       }
312      }
313      else {
314       Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
315       STOPSTEP(; exception_occurred);
316
317      };
318     }
319     else {
320      Bank-_state_bank(ag) := GENERATED_STATE_IDLE;
321      STOPSTEP(; exception_occurred);
322     };
323    }
324    };
325
326    Transaction# : Transaction# {
327    if (verify(Bank-verifyKey(ag), inmsg.authToken_AuthToken.signedData_SignedData,
328      wrapAuthTokenData2SignData(inmsg.authToken_AuthToken.data_AuthTokenData))) then {
329     let increased-bool-var = false
330     in {
331      if (not containsKey(Shared-Bank-cTransactions(BankAddress(ag)),
332        inmsg.ut_UniqueTransaction.n_Nonce)) then {
333       increased-bool-var := credit(inmsg.ut_UniqueTransaction.transfer_Transfer.
334        to_AccountInfo.accountNumber_string,
335        inmsg.ut_UniqueTransaction.transfer_Transfer.money_int, ag; Bank-verifyKey,
336        Bank-address, Bank-_state_bank, Bank-bankAuthToken, Bank-bankCodes,
337        Shared-Bank-accounts, Shared-Bank-iTransactions, Shared-Bank-cTransactions,
338        Bank-ut, Bank-account, BankAddress, exception_occurred);
339       if (exception_occurred) then { skip }
340       else {
341        Shared-Bank-cTransactions(BankAddress(ag)) :=
342         set(Shared-Bank-cTransactions(BankAddress(ag)),
343          mkUniqueTransaction(inmsg.ut_UniqueTransaction.n_Nonce, true,
344            increased-bool-var, inmsg.ut_UniqueTransaction.transfer_Transfer));
345        SEND(mkTransactionResult(increased-bool-var), Bank2BankDefaultPort, ag,
346         connections; attacker-known, inputs);
347       }
348      }
349      else {
350       increased-bool-var :=
351        get(Shared-Bank-cTransactions(BankAddress(ag)),
352         inmsg.ut_UniqueTransaction.n_Nonce).increased_bool;
353       SEND(mkTransactionResult(increased-bool-var), Bank2BankDefaultPort, ag,
354        connections; attacker-known, inputs);
355      };
356     }
357    }
358    else {
359     STOPSTEP(; exception_occurred);
360    };
361    };
362
363    ProcessTransferTimeEvent# : ProcessTransferTimeEvent#
364    {
365     Bank-ut(ag) := at(Shared-Bank-iTransactions(BankAddress(ag)), 0);
366     Shared-Bank-iTransactions(BankAddress(ag)) :=
367      set(Shared-Bank-iTransactions(BankAddress(ag)), Bank-ut(ag));
368
369     let url-String-var =
370      get(Bank-bankCodes(ag),
371       Bank-ut(ag).transfer_Transfer.to_AccountInfo.bankCode_string).address_string;
372     in {
373      if (attackerAddress = url-String-var) then {
374       CONNECTTOSERVICE(ag, Bank2Bank_default, ATTACKER; connections, inputs);
375       SEND(mkTransaction(Bank-ut(ag), Bank-bankAuthToken(ag)), Bank2Bank_default, ag,
376         connections; attacker-known, inputs);
377      }
378      else{
379       if (serviceClassName(url-String-var) = Bank) then {
380        let ag0 = Bank(next-BankAgentName) in {
381         BankAddress(ag0) := url-String-var;
382
383         Bank-verifyKey(ag0) := Initial-Bank-verifyKey(url-String-var);
384         Bank-bankAuthToken(ag0) := Initial-Bank-bankAuthToken(url-String-var);
385         Bank-bankCodes(ag0) := Initial-Bank-bankCodes(url-String-var);
386         Bank-accounts(ag0) := Initial-Bank-accounts(url-String-var);
387
388         next-BankAgentName := next-BankAgentName + 1;
389         CONNECTTOSERVICE(ag, Bank2Bank_default, ag0; connections, inputs);
```

```
390       SEND(mkTransaction(Bank-ut(ag), Bank-bankAuthToken(ag)), Bank2Bank_default, ag,
391         connections; attacker-known, inputs);
392       }
393      }
394     }
395    }
396   };
397
398   end asm specification
```

Listing 9.8: Spezifikation der ASM für die Onlinebanking-Anwendung

```
1   enrich USER, allInstances, ManualFuns, accessfuns, Status,
2    listofBankCodefuns, listofAccountfuns,
3    listofUniqueTransactionfuns, listofUniqueTransactionfuns
4   with
5   predicates
6    init : (agent -> string), (agent -> Secret),
7     (agent -> GeneratedStateUserDevice), (agent -> PublicKey),
8     (agent -> string), (agent -> GeneratedStateBank),
9     (agent -> AuthToken), (agent -> listofBankCode),
10    (agent -> UniqueTransaction), (agent -> Account),
11    (string -> listofAccount), (string -> listofUniqueTransaction),
12    (string -> listofUniqueTransaction), (string -> PublicKey),
13    (string -> AuthToken), (string -> listofBankCode),
14    (string -> listofAccount), nat, (agent -> string),
15    (agent -> string), string, (nat -> Nonce), nat,
16    (agent -> ports -> messagelist), connectionset,
17    attackerdataset, bool;
18
19  procedures
20   STOPSTEP : bool ;
21
22   debit string, int,  agent : bool, (agent -> PublicKey), (agent -> string),
23   (agent -> GeneratedStateBank), (agent -> AuthToken), (agent -> listofBankCode),
24   (string -> listofAccount), (string -> listofUniqueTransaction),
25   (string -> listofUniqueTransaction), (agent -> UniqueTransaction),
26   (agent -> Account), bool nonfunctional;
27
28   credit string, int,  agent : bool, (agent -> PublicKey), (agent -> string),
29   (agent -> GeneratedStateBank), (agent -> AuthToken), (agent -> listofBankCode),
30   (string -> listofAccount), (string -> listofUniqueTransaction),
31   (string -> listofUniqueTransaction), (agent -> UniqueTransaction),
32   (agent -> Account), bool nonfunctional;
33
34  variables
35   exception_occurred : bool;
36   AccountOwner-accountNumber : (agent -> string);
37   AccountOwner-pw : (agent -> Secret);
38   Bank-_state_bank : (agent -> GeneratedStateBank);
39   Bank-account : (agent -> Account);
40   Bank-address : (agent -> string);
41   Bank-bankAuthToken : (agent -> AuthToken);
42   Bank-bankCodes : (agent -> listofBankCode);
43   Bank-ut : (agent -> UniqueTransaction);
44   Bank-verifyKey : (agent -> PublicKey);
45   UserDevice-_state_userdevice : (agent -> GeneratedStateUserDevice);
46   account-Account-var : Account;
47   accountNumber-string-var : string;
48   decreased-bool-var : bool;
49   increased-bool-var : bool;
50   money-int-var : int;
51
52   Initial-Bank-verifyKey : (string -> PublicKey);
53   Initial-Bank-bankAuthToken : (string -> AuthToken);
54   Initial-Bank-bankCodes : (string -> listofBankCode);
55   Initial-Bank-accounts : (string -> listofAccount);
56
57   Shared-Bank-cTransactions : (string -> listofUniqueTransaction);
58   Shared-Bank-iTransactions : (string -> listofUniqueTransaction);
59   Shared-Bank-accounts : (string -> listofAccount);
60
61   next-BankAgentName : nat;
62   BankAddress : (agent -> string)
63
64   UserDevice-UserDevice2Bank_default-Address : (agent -> string);
65   attackerAddress : string;
66
67
68  declaration
69   STOPSTEP : STOPSTEP(; exception_occurred) { exception_occurred := true; };
70
```

```
71  debit(accountNumber-string-var, money-int-var, ag; decreased-bool-var,
72   Bank-verifyKey, Bank-address, Bank-_state_bank, Bank-bankAuthToken,
73   Bank-bankCodes, Shared-Bank-accounts, Shared-Bank-iTransactions,
74   Shared-Bank-cTransactions, Bank-ut, Bank-account, BankAddress,
75   exception_occurred)
76  {
77    decreased-bool-var := false;
78    if (not containsKey(Shared-Bank-accounts(BankAddress(ag)),
79     accountNumber-string-var))
80    then {
81     STOPSTEP(; exception_occurred); }
82    else {
83     let account-Account-var =
84      get(Shared-Bank-accounts(BankAddress(ag)), accountNumber-string-var)
85     in {
86      if (((money-int-var > 0) and
87       ((account-Account-var.balance_int - money-int-var) \ge 0)))
88      then {
89       account-Account-var.balance_int :=
90        (account-Account-var.balance_int - money-int-var);
91       Shared-Bank-accounts(BankAddress(ag)) :=
92        set(Shared-Bank-accounts(BankAddress(ag)), account-Account-var);
93       decreased-bool-var := true;
94      }
95     }
96    }
97
98  };
99
100 credit(accountNumber-string-var, money-int-var,  ag; increased-bool-var,
101  Bank-verifyKey, Bank-address, Bank-_state_bank, Bank-bankAuthToken,
102  Bank-bankCodes, Shared-Bank-accounts, Shared-Bank-iTransactions,
103  Shared-Bank-cTransactions, Bank-ut, Bank-account, BankAddress,
104  exception_occurred)
105 {
106  increased-bool-var := false;
107  if (containsKey(Shared-Bank-accounts(BankAddress(ag)),
108   accountNumber-string-var))
109  then {
110   if (not containsKey(Shared-Bank-accounts(BankAddress(ag)),
111    accountNumber-string-var))
112   then {
113    STOPSTEP(; exception_occurred); }
114   else {
115    let account-Account-var =
116     get(Shared-Bank-accounts(BankAddress(ag)), accountNumber-string-var)
117    in {
118     account-Account-var.balance_int :=
119      (account-Account-var.balance_int + money-int-var);
120     Shared-Bank-accounts(BankAddress(ag)) :=
121      set(Shared-Bank-accounts(BankAddress(ag)), account-Account-var);
122     increased-bool-var := true;
123    }
124   }
125   }
126 };
127
128
129 end enrich
```

Listing 9.9: Spezifikation der ASMAUX für die Onlinebanking-Anwendung

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] M. Alam. Model driven security engineering for the realization of dynamic security requirements in collaborative systems. In *Models in Software Engineering*, pages 278–287. Springer, 2007.

[2] M. Alam, R. Breu, and M. Breu. Model driven security for web services (MDS4WS). In *Multitopic Conference, 2004. Proceedings of INMIC 2004. 8th International*, pages 498–505. IEEE, 2004.

[3] R. J. Anderson and R. M. Needham. Programming satan's computer. In *Computer Science Today*, volume 1000, pages 426–440. Springer, 1995.

[4] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282, 2012.

[5] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10. ACM, 2008.

[6] W. Arsac, L. Compagna, G. Pellegrino, and S. Ponta. Security validation of business processes via model-checking. *Engineering Secure Software and Systems*, pages 29–42, 2011.

[7] K. Baina, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *Advanced Information Systems Engineering*, pages 527–543. Springer, 2004.

[8] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[9] F. Banti, R. Pugliese, and F. Tiezzi. An accessible verification environment for UML models of services. *Journal of Symbolic Computation*, 46(2):119–149, 2011.

[10] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, pages 39–91, 2006.

[11] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach.* Springer-Verlag, Berlin, Heidelberg, 2007.

[12] G. Bella. Mechanising a Protocol for Smart Cards. In *Proc. of e-Smart 2001, international conference on research in smart cards*, volume 2140 of *LNCS*. Springer-Verlag, 2001.

[13] G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET purchase protocols. *Journal of Automated Reasoning*, 36(1-2):5–37, 2006.

[14] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *Internet Computing, IEEE*, 7(1):40–48, 2003.

[15] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. Mckenzie. *SMV — Symbolic Model Checking*, pages 131–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[16] M. Borek, K. Katkalov, N. Moebius, W. Reif, G. Schellhorn, and K. Stenzel. Integrating a model-driven approach and formal verification for the development of secure service applications. In B. Thalheim, K.-D. Schewe, A. Prinz, and B. Buchberger, editors, *Correct Software in Web Applications and Web Services*, Texts & Monographs in Symbolic Computation, pages 45–81. Springer International Publishing, 2015.

[17] M. Borek, N. Moebius, K. Stenzel, and W. Reif. Model-driven development of secure service applications. In *35th Annual IEEE Software Engineering Workshop (SEW 2012)*, pages 62–71. IEEE, 2012.

[18] M. Borek, N. Moebius, K. Stenzel, and W. Reif. Model checking of security-critical applications in a model-driven approach. In R. Hierons, M. Merayo, and M. Bravetti, editors, *Software Engineering and Formal Methods*, volume 8137 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2013.

[19] M. Borek, N. Moebius, K. Stenzel, and W. Reif. Security requirements formalized with ocl in a model-driven approach. In *Model-Driven Requirements Engineering Workshop (MoDRE)*. IEEE, 2013.

[20] M. Borek, K. Stenzel, K. Katkalov, and W. Reif. Abstracting security-critical applications for model checking in a model-driven approach. In *6th IEEE International Conference On Software Engineering And Service Science (ICSESS)*. IEEE, 2015.

[21] M. Borek, K. Stenzel, K. Katkalov, and W. Reif. Integration and exchangeability of external security-critical web services in a model-driven approach. In M. Jeusfeld and K. Karlapalem, editors, *Advances in Conceptual Modeling*, volume 9382 of *Lecture Notes in Computer Science*, pages 63–73. Springer International Publishing, 2015.

[22] M. Borek, K. Stenzel, K. Katkalov, and W. Reif. Secure integration of third party components in a model-driven approach. In *TLDKS Special Issue on Cloud Computing*. Springer, 2016.

[23] E. Börger and O. Sörensen. *Handbook of Conceptual Modeling. Theory, Practice, and Research Challenges*, chapter BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics, pages 287–332. Springer, 2011.

[24] E. Börger and B. Thalheim. Modeling workflows, interaction patterns, web services and business processes: The ASM-based approach. In *Proceedings of ABZ 2008*. Springer LNCS 5238, 2008.

[25] R. Breu, G. Popp, and M. Alam. Model based development of access policies. *International Journal on Software Tools for Technology Transfer*, 9(5):457–470, 2007.

[26] P. Bright. Common payment processing protocols found to be full of flaws, 2015. `http://arstechnica.com/security/2015/12/common-payment-processing-protocols-found-to-be-full-of-flaws`.

[27] A. D. Brucker, I. Hang, G. Lückemeyer, and R. Ruparel. Securebpmn: Modeling and enforcing access control requirements in business processes. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 123–126. ACM, 2012.

[28] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pages 682–682. Springer, 1999.

[29] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel. Sound development of secure service-based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 115–124. ACM, 2004.

[30] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF Network Working Group, August 2008. `http://www.ietf.org/rfc/rfc5246.txt`.

[31] 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 Computer Society, 1981.

[32] D. Fett, R. Küsters, and G. Schmitz. A comprehensive formal security analysis of oauth 2.0. *CoRR*, abs/1601.01229, 2016.

[33] H. Foster, L. Gönczy, N. Koch, P. Mayer, C. Montangero, and D. Varró. UML extensions for service-oriented systems. In *Rigorous software engineering for service-oriented systems*, pages 35–60. Springer-Verlag, 2011.

[34] S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró. Non-functional properties in the model-driven development of service-oriented systems. *Software & Systems Modeling*, 10(3):287–311, 2011.

[35] M. Green. A few thoughts on cryptographic engineering. `https://blog.cryptographyengineering.com/2015/08/16/the-network-is-hostile/`, August 2015.

[36] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-driven web services development. In *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*, pages 42–45. IEEE, 2004.

[37] J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and verification of layered security protocols: A bank application. In *Proceedings of SAFECOMP 2003*. Springer LNCS 2788, 2003.

[38] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh. Security requirements engineering: A framework for representation and analysis. *Software Engineering, IEEE Transactions on*, 34(1):133–153, 2008.

[39] 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.

[40] B. Hariri, D. Calvanese, G. D. Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 163–174. ACM, 2013.

[41] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1):6:1–6:42, Mar. 2013.

[42] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444, 2006.

[43] G. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[44] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus – a tool for distributed systems specification. In *Proceedings of Formal Techniques in real-time and fault-tolerant Systems*, pages 467–470. Springer, 1996.

[45] International Organization for Standardization (ISO). *Financial transaction card originated messages*, 2003. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=31628`.

[46] Internet Engineering Task Force (IETF). *The OAuth 2.0 Authorization Framework*, 2012. `https://tools.ietf.org/html/rfc6749`.

[47] Internet Engineering Task Force (IETF). *OAuth 2.0 Threat Model and Security Considerations*, 2013. `https://tools.ietf.org/html/rfc6819`.

[48] *Java Card 2.2.2 Application Programming Interfaces*, 2006. `http://www.oracle.com/technetwork/java/javacard/specs-138637.html`.

[49] J. Jensen and M. Jaatun. Security in model driven development: A survey. In *Sixth International Conference on Availability, Reliability and Security, ARES 2011*, Lecture Notes in Computer Science, pages 704–709. Springer, 2011.

[50] J. Jürjens. Model-based security engineering with uml. In *Foundations of Security Analysis and Design III*, pages 42–77. Springer, 2005.

[51] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[52] K. Kasal, J. Heurix, and T. Neubauer. Model-Driven Development Meets Security: An Evaluation of Current Approaches. In *44th Hawaii International Conference on System Sciences (HICSS)*, pages 1–9. IEEE Computer Society, 2011.

[53] K. Katkalov, N. Moebius, K. Stenzel, M. Borek, and W. Reif. Modeling test cases for security protocols with SecureMDD. *Computer Networks*, 58(0):99 – 111, 2014.

[54] B. Li, Y. Zhou, and J. Pang. Model-driven automatic generation of verified BPEL code for web service composition. In *Asia-Pacific Software Engineering Conference (APSEC'09)*, pages 355–362. IEEE, 2009.

[55] G. 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.

[56] P. Mayer. *MDD4SOA: Model-Driven Development for Service-Oriented Architectures*. PhD thesis, LMU, Munich, Germany, 2010.

[57] P. Mayer, N. Koch, A. Schroeder, and A. Knapp. The UML4SOA profile. Technical report, LMU, Munich, Germany, 2010.

[58] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-driven service orchestration. In *Proceedings of 12th IEEE International EDOC Conference (EDOC 2008)*. IEEE press, 2008.

[59] M. Menzel. *Model-driven Security in Service-oriented Architectures.* PhD thesis, Potsdam University, 2011. `http://opus.kobv.de/ubp/volltexte/2012/5905/`.

[60] M. Menzel and C. Meinel. SecureSOA: Modelling security requirements for service-oriented architectures. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 146–153. IEEE, 2010.

[61] N. Mitra and Y. Lafon. *SOAP Version 1.2.* W3C, 2007.

[62] N. Moebius. *Modellgetriebene Entwicklung sicherer Smart Card-Anwendungen.* doctoralthesis, Universität Augsburg, 2013.

[63] N. Moebius, K. Stenzel, M. Borek, and W. Reif. Incremental development of large, secure smart card applications. In *Proceedings of the Workshop on Model-Driven Security.* ACM, 2012.

[64] M. Montali, M. Pesic, W. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographiess. *ACM Transactions on the Web (TWEB)*, 4(1):3, 2010.

[65] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist. *WS-SecurityPolicy 1.2.* OASIS, 2006.

[66] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *Web Services Security: SOAP Message Security 1.0.* OASIS, 2004.

[67] Y. Nakamura, M. Tatsubori, T. Imamura, and K. Ono. Model-driven security based on a web services security architecture. In *IEEE International Conference on Services Computing*, pages 7–15. IEEE Press, 2005.

[68] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[69] S. Nolte. *QVT-Operational Mappings: Modellierung mit der Query Views Transformation.* Springer, 2009.

[70] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, 2011.

[71] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6, 1998.

[72] M. Pavlidis, S. Islam, and H. Mouratidis. A case tool to support automated modelling and analysis of security requirements, based on secure tropos. In *IS Olympics: Information Systems in a Diverse World*, pages 95–109. Springer, 2012.

[73] J. C. L. Pimental and R. Monroy. Formal support to security protocol development: A survey. *Computacion y Sistemas*, 12(1), 2008.

[74] A. Pironti, D. Pozza, and R. Sisto. Formally-based semi-automatic implementation of an open security protocol. *Journal of Systems and Software, Elsevier*, 85(4), 2012.

[75] D. Pozza, R. Sisto, and L. Durante. Spi2java: automatic cryptographic protocol java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, volume 1, pages 400–405 Vol.1, 2004.

[76] M. Ray and S. Dispensa. Renegotiating TLS. Technical report, PhoneFactor Inc., Nov. 2009.

[77] R. M. Roberto Chinnici, Marc Hadley. *The Java API for XML-Based Web Services (JAX-WS) 2.0*. JCP, 2006.

[78] M. Saleem, J. Jaafar, and M. Hassan. A domain-specific language for modelling security objectives in a business process models of soa applications. *AISS*, 4(1):353–362, 2012.

[79] M. Turuani. The cl-atse protocol analyser. *Term Rewriting and Applications*, pages 277–286, 2006.

[80] D. Von Oheimb and S. Mödersheim. Aslan++ a formal security specification language for distributed systems. In *Formal Methods for Components and Objects*, pages 1–22. Springer, 2012.

[81] G. Wimmel and A. Wisspeintner. *Extended Description Techniques for Security Engineering*, pages 469–485. Springer US, Boston, MA, 2001.

[82] W. Yeung. A formal and visual modeling approach to choreography based web services composition and conformance verification. *Expert Systems with Applications*, 38(10):12772–12785, 2011.