EIN MODELLGETRIEBENER ANSATZ ZUR ENTWICKLUNG INFORMATIONSFLUSSSICHERER SYSTEME

KUZMAN KATKALOV

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

ERSTGUTACHTER: Prof. Dr. Wolfgang Reif
ZWEITGUTACHTER: Prof. Dr. Bernhard Bauer
MÜNDLICHE PRÜFUNG: 03. Juli 2017

Vielen Dank an Wolfgang Reif,
Kurt Stenzel,
Marian Borek,
und Nina Moebius,
ohne die es diese Arbeit nicht gegeben hätte.

ABSTRACT

The ubiquitous mobile devices and web services acquire and collect a lot of personal data about their users. In many cases, the confidentiality of such data is not guaranteed and is frequently violated; thus, the users' sensitive data is often leaked to other users or third parties.

This work presents a model-driven approach called *IFlow* that allows the development of distributed applications consisting of mobile apps and web services with secure information flow. Using the modeling language MODELFLOW, the developer creates a model of a security critical application and specifies its information flow properties. Such properties are then guaranteed using automatic information flow control as well as interactive verification. The final application consists of Android apps and Java web services generated automatically from the application model that satisfy the specified information flow properties.

ZUSAMMENFASSUNG

Die allgegenwärtigen und immer verbundenen mobilen Geräte sammeln große Mengen an persönlichen Daten über ihre Nutzer. In vielen Fällen wird die Vertraulichkeit solcher Daten nicht garantiert; so kommt es bei mobilen Apps und Webservices oft zu Datenlecks, wodurch die Privatsphäre ihrer Nutzer verletzt wird.

Diese Arbeit stellt den modellgetriebenen Ansatz *IFlow* zur Entwicklung informationsflusssicherer Anwendungen bestehend aus mobilen Apps und Webservices vor. Hierzu wird mit der Modellierungssprache MODELFLOW das Modell einer sicherheitskritischer Anwendung erstellt, und ihre Informationsflusseigenschaften spezifiziert. Anschließend können diese Eigenschaften mit Hilfe vollautomatischer Informationsflussanalyse sowie interaktiver Verifikation garantiert werden. Die finale Anwendung besteht aus Android-Apps und Java-Webservices, die aus dem Modell generiert werden, und die modellierten Informationsflusseigenschaften erfüllen.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

Teil I

EINLEITUNG

„That puts it well," G. G. said.
„You're a policeman guarding human privacy."

„You know what Ray Hollis says about us?" Runciter said.
„He says we're trying to turn the clock back."

— Philip K. Dick, **Ubik**, 1968

1 MOTIVATION

In der Zeit der allgegenwärtigen und immer verbundenen mobilen Geräte scheint das Ideal der Privatsphäre nicht mehr als ein Wunschtraum zu sein.

Jedes solcher Geräte ist mit einer Reihe von Sensoren ausgestattet, mit denen es persönliche Daten über seinen Besitzer (oft ohne ihrer expliziten Zustimmung [44]) sammelt, und diese auf Webserver von Dritten hochlädt. Aber auch die Nutzer selbst sind allzu oft bereit, den mobilen Geräten ihre sensitive Informationen anzuvertrauen. Dazu zählen beispielsweise die Kontaktdaten ihrer Freunde, Kunden, oder Partner, private Photos, oder sogar höchst vertrauliche Zahlungsdaten.

Viele der Anwendungen und Services, die diese Daten speichern, austauschen, und verwerten, geben ihren Nutzern keine klare Auskunft über den Umfang und Verwendungszweck der gesammelten Informationen. Zudem kommt es bei mobilen Apps und Webservices oft zu Datenlecks [24], was zu Verletzungen der Privatsphäre ihrer Nutzer, aber auch Reputationsschäden bei den Entwicklern solcher Anwendungen führen kann. Bei komplexen verteilten Anwendungen bestehend aus einer Vielzahl von interagierenden mobilen Apps und Webservices sind solche Datenlecks oft das Ergebnis von Implementierungsfehlern. So gab es Juni 2016 einen Datenleck bei *Uber*, einem der größten Vermittlungsdienste zur Personenbeförderung, bei dem persönliche Daten wie Email-Adressen und vergangene Fahrten durch nicht autorisierte Nutzer eingesehen werden konnten [74].

Heute sind die Nutzer auf herkömmliche Sicherheitsmechanismen sowie die Datenschutz-Bestimmungen der Anwendung angewiesen, die jedoch keine starken Garantien bzgl. der Geheimhaltung ihrer sensitiven Information geben können. Ein solcher Mechanismus ist beispielsweise das Berechtigungssystem, wie es bei den aktuellen mobilen Plattformen wie Android, iOS, und Windows Phone zum Einsatz kommt. Der Entwickler einer mobilen App muss eine Liste von Berechtigungen angeben, die seine App vom Benutzer anfordert, um z.B. auf den GPS-Sensor oder die SD-Karte zugreifen zu können. Jedoch ist ein solches Berechtigungssystem zu grobkörnig, um die

Vertraulichkeit der Nutzerdaten zu garantieren. Installiert etwa der Nutzer eine App, die sowohl Zugriff auf den GPS-Sensor als auch den Zugang zum Internet erhält, so kann er sich nicht vergewissern, dass diese App seinen aktuellen Aufenthaltsort nicht an Drittparteien verrät. Auch durch den Zusammenschluss mehrerer Apps, von denen jede scheinbar harmlose Berechtigungen anfordert, kann es zu Datenlecks kommen [38, 83].

Um die Geheimhaltung sensitiver Informationen in verteilten Systemen bestehend aus mobilen Apps und Services zu garantieren, sind Techniken notwendig, die den *Informationsfluss* (*IF*) in solchen Systemen betrachten. Jedoch stellt die praktische Anwendung der theoretischen Grundlagen und Analysetechniken für Informationsflusskontrolle von realistischen und benutzerfreundlichen Systemen eine Herausforderung dar.

2 ZIELE UND ERREICHTE ERGEBNISSE

Das Hauptziel der vorliegenden Arbeit ist eine einheitliche Systementwicklungsmethodik, die die Stärken der statischen Codeanalyse und interaktiver Verifikation integriert, um sinnvolle und interessante Informationsflusseigenschaften für reale Anwendungen zu garantieren. Dieser Ansatz soll den Entwickler dabei unterstützen, starke Informationsflussgarantien für seine Anwendung zu liefern, indem er diese bereits in der Designphase berücksichtigt (*Privacy by Design*). Der Fokus liegt dabei auf verteilten Anwendungen bestehend aus mobilen Apps und Webservices.

Im Rahmen der Arbeit entstand der modellgetriebene Ansatz *IFlow* zur Entwicklung informationsflusssicherer Anwendungen (siehe Kapitel 3 für eine Übersicht über die Systementwicklungsmethodik von IFlow). Die folgenden Ergebnisse wurden dabei erreicht:

- **Entwicklung einer Modellierungssprache zum Design informationsflusssicherer Anwendungen und ihrer Eigenschaften**

  Im Lauf der Arbeit entstand die Modellierungssprache MODELFLOW, die auf der *Unified Modeling Language* (UML) [71] aufbaut. Damit ist es möglich, die Struktur und das Verhalten einer verteilten Anwendung bestehend aus mobilen Apps und Services zu modellieren. Die Sprache erlaubt die intuitive und flexible Spezifikation der Informationsflusseigenschaften dieser Anwendung, sowie die Integration von händisch und extern implementierten Modulen (wie etwa einer graphischen Benutzeroberfläche oder einer vorinstallierten mobilen App), um die flexible Entwicklung von möglichst realistischen Anwendungen zu ermöglichen. Automatische Modelltransformationen generieren aus dem Anwendungsmodell ein formales Modell sowie ausführbaren Java-Code, die bzgl. ihrer Informationsflusseigenschaften verifiziert bzw. analysiert werden können.

  Kapitel 9 beschreibt im Detail die Sprache MODELFLOW zur Modellierung der Struktur, des Verhaltens, und der Informationsflusseigenschaften einer IFlow-Anwendung. Das Kapitel baut auf den in [48, 50–53, 93] publizierten Arbeiten auf.

- **Erarbeitung formaler Grundlagen und der formalen Semantik der Modellierungssprache**

  Die formale Semantik von MODELFLOW ist durch ein formales Modell gegeben, das die modellierte Anwendung als eine abstrakte Zustandsmaschine (engl. *abstract state machine, ASM*)

abbildet, und das theoretische Informationsflussframework von Rushby [80] instantiiert. Damit ist es möglich, die modellierten Informationsflusseigenschaften mit dem interaktiven Beweiser KIV [3] formal zu verifizieren, die sich auf die finale Anwendung vererben.

Kapitel 15 beschreibt die Informationsflusstheorie und das formale Modell einer IFlow-Anwendung. Die formalen Grundlagen des IFlow-Ansatzes wurden in [53, 91–93] publiziert.

- **Ansatz zur vollautomatischen Informationsflusskontrolle von verteilten Anwendungen bestehend aus mobilen Apps und Webservices**

  Um die vollautomatische, sprachbasierte Informationsflussanalyse des generierten und händisch implementierten Code zu ermöglichen, wurde ein Ansatz zur automatischen Abstraktion der verteilten Anwendung bestehend aus mobilen Apps und Webservices als eine monolitische Java-Anwendung entwickelt. Dies ermöglicht den Einsatz der vollautomatischen, sprachbasierten Informationsflusskontrolle der modellierten Anwendung mit dem *Java Object-sensitive ANAlysis* (JOANA)-Framework [35], sowie den Check der optionalen händischen Implementierung. Der Ansatz garantiert, dass sich die so gezeigten Informationsflusseigenschaften auf die finale, verteilte Anwendung bestehend aus mobilen Apps für die Android-Plattform und Java-Webservices vererbt.

  Kapitel 16 beschreibt den Aufbau und automatische Informationsflussanalyse des Java-Codes einer IFlow-Anwendung, während Abschnitt 17.2 die eingesetzten Abstraktionstechniken erläutert. Die Grundlagen automatischer Informationsflussanalyse in IFlow wurden in [49, 51, 53] publiziert.

- **Benutzerzentrische Sicht auf Informationsflusssicherheit**

  Mobile Apps unterscheiden sich von anderen sicherheitskritischen Systemen wie Webservices oder Smartcard-Anwendungen u.a. durch ihre direkte Schnittstelle zum Endnutzer. Deshalb wurde besonderer Wert auf die intuitive Verständlichkeit und Transparenz der modellierten Informationsflusseigenschaften, sowie die sichere Integration einer benutzerfreundlichen graphischen Benutzeroberfläche gelegt. Zudem wurde ein Ansatz implementiert, der dem Nutzer die Spezifikation und Analyse eigener Informationsflusseigenschaften einer IFlow-Anwendung ermöglicht.

  Unterabschnitt 16.2.4 geht auf die benutzerzentrische Sicht im IFlow-Ansatz ein, und baut dabei auf der in [51] publizierten Arbeit auf.

- **Evaluation des Ansatzes anhand mehrerer Fallstudien**

  Der IFlow-Ansatz wurde anhand einer Reihe von Fallstudien evaluiert. Dazu zählen die Anwendungen *TravelPlanner* zur Flugbuchung bei einer Reiseagentur, *BankingApp* zum Verwalten eines Online-Kontos, *DistanceTracker* zum Aufzeichnen einer Laufdistanz, *ContactSMSManager* zum Verwalten von privaten Kontakten und Versand von SMS, sowie *PrivateTaxi* zum Vermittlung von Fahrgemeinschaften.

  Die Fallstudien und ihre Informationsflusseigenschaften werden in Teil v beschrieben. Ausgewählte Fallstudien wurden u.a. in [50, 93] publiziert.

Teil II

IFLOW: EIN MODELLGETRIEBENER ANSATZ ZUR ENTWICKLUNG INFORMATIONSFLUSSSICHERER SYSTEME

3 SYSTEMENTWICKLUNGSMETHODIK

IFlow ist ein modellgetriebener Ansatz, der den Entwickler bei der Konzeption, Modellierung, Verifikation, Analyse, und Implementierung informationsflusssicherer Anwendungen unterstützt. Der Fokus liegt dabei auf verteilten Anwendungen bestehend aus mobilen Apps und Webservices.

Der Ansatz entstand im Rahmen des gleichnamigen, DFG-geförderten Projekts „IFlow“ als Teil des koordinierten Programms „Reliably Secure Software Systems (RS^3)“

Der IFlow-Ansatz ermöglicht es einem Entwickler, den Nutzern seiner Anwendung starke Garantien darüber zu liefern, wie diese mit vertraulichen Daten umgeht, und welche Systemnutzer oder Service-Betreiber diese Daten in Erfahrung bringen können. Die Privatsphäre der Endnutzer wird dabei bereits beim Entwurf der Anwendung berücksichtigt (*Privacy by Design*). Dazu nutzt der Ansatz modell- und sprachbasierte Techniken der *Informationsflusskontrolle*, die den Fluss geheimer Information durch ein System betrachtet. Solche Techniken geben stärkere Garantien bezüglich Vertraulichkeit von Daten als der bloße Einsatz von Verschlüsselung oder Zugriffskontrolle, da sie die Geheimhaltung von Informationen selbst nach ihrer Entschlüsselung bzw. nach einem erlaubten Zugriff garantieren können.

Der wissenschaftliche Hauptbeitrag von IFlow ist eine einheitliche Systementwicklungsmethodik, die die Stärken der statischen Codeanalyse und interaktiver Verifikation integriert, um sinnvolle und interessante Informationsflusseigenschaften für reale Anwendungen zu garantieren.

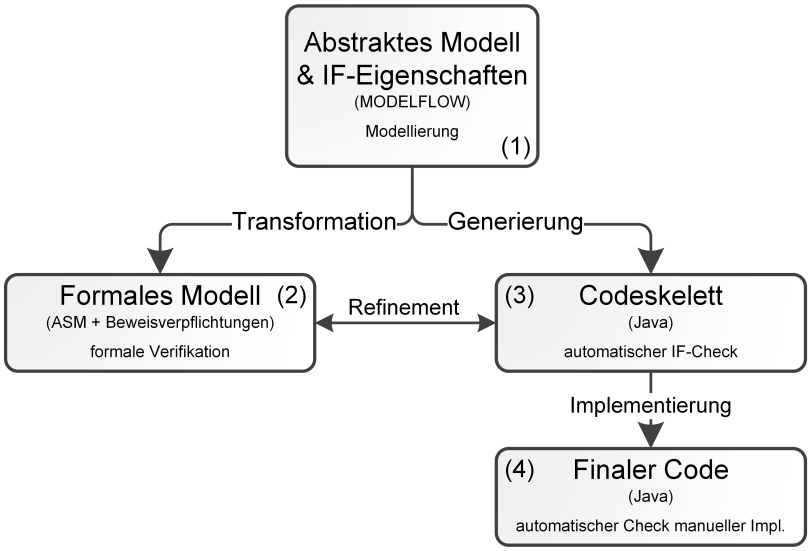


Abbildung 1: Überblick über den IFlow-Ansatz

Abbildung 1 gibt einen Überblick über den IFlow-Ansatz, der in vier Phasen unterteilt wird: (1) Modellierung, (2) interaktive Verifikation des formalen Modells, (3) statische Analyse des Codeskeletts, und (4) Ausbau des Codeskeletts zur finalen Anwendung. Im Folgenden werden diese Phasen näher erläutert.

MODELLIERUNG Der Entwickler erstellt ein abstraktes Modell der verteilten Anwendung mit MODELFLOW–der Modellierungssprache von IFlow, die auf UML aufbaut (1). MODELFLOW ergänzt UML um ein IFlow-spezifisches UML-Profil, und definiert Regeln zur Modellierung von verteilten Anwendungen und ihren Informationsflusseigenschaften.

Für Modellierung mit MODELFLOW *kann ein UML-Tool wie MagicDraw verwendet werden,* nomagic.com

Das Anwendungsmodell muss alle notwendigen Informationen über die Anwendungskomponenten (d.h., mobile Apps und Webservices) enthalten. Dies umfasst die statische sowie die dynamische Sicht auf die Anwendung, die die Struktur und das Verhalten solcher Komponenten festlegen. Zum Systemverhalten zählt die Interaktion der Komponenten, die Verarbeitung der dabei ausgetauschten Nachrichten, und ggf. Filtermethoden, die geheime Nutzerinformationen anonymisieren. Weitere Teile des Systems wie etwa graphische Benutzeroberflächen dürfen im Modell unterspezifiziert sein, und zu einem späteren Zeitpunkt von Hand implementiert werden. Der Entwickler kann zudem das Modell mit Anforderungen an das Verhalten der zukünftigen, manuellen Implementierung annotieren, die bei der Verifikation und statischer Analyse als Annahmen verwendet werden.

Zudem definiert das Modell die anwendungsspezifischen Informationsflusseigenschaften des Systems in einer für den Nutzer intuitiv verständlichen Form. Zu den unterstützten Eigenschaften zählt die Vertraulichkeit geheimer Information (*Nichtinterferenz*), sowie Garantien bezüglich ihrer expliziten Freigabe (*Deklassifikation*). Dadurch kann z.B. der Fluss geheimer Benutzerinformation wie etwa Kreditkarten- oder Kontaktdaten zu bestimmten Anwendungskomponenten verboten, oder nur in gefilterter bzw. anonymisierter Form erlaubt werden.

INTERAKTIVE VERIFIKATION Manche der modellierten Informationsflusseigenschaften wie etwa die Sicherheit der modellierten Filterfunktionen müssen mit Hilfe von interaktiver Verifikation garantiert werden. Das abstrakte Anwendungsmodell wird hierzu automatisch zu einem formalen Modell transformiert, das auf einer abstrakten Zustandsmaschine (ASM) und algebraischen Spezifikationen basiert (Abbildung 1, (4)). Die ASM bildet dabei das modellierte Verhalten der Anwendung ab; die Beweisverpflichtungen werden aus den spezifizierten Informationsflusseigenschaften abgeleitet.

Das formale Modell kann mit dem interaktiven Beweiser KIV [4] eingelesen werden, um die Anwendung bezüglich der Beweisverpflichtungen zu verifizieren.

STATISCHE CODEANALYSE Aus dem abstrakten Anwendungsmodell wird automatisch ein monolithisches Java-Codeskelett generiert, das die Struktur und das Verhalten der modellierten Anwendung implementiert. Dieses Codeskelett abstrahiert dabei von plattformspezifischen Funktionalität wie Netzwerkkommunikation oder Zugriff auf den GPS-Sensor des mobilen Geräts, die für die finale, lauffähige Anwendung benötigt wird, und simuliert dabei ihre Informationsflüsse. Auch die Teile des Systems, deren manuelle Implementierung noch nicht vorliegt, werden auf diese Weise abstrahiert.

Diese Abstraktionen erlauben es dem Entwickler, ein statisches Informationsflussanalysetool wie JOANA [32] einzusetzen, da sie die Komplexität der Anwendung reduzieren. So können viele der modellierten Informationsflusseigenschaften vollautomatisch gecheckt werden, ohne dass der Entwickler Erfahrung mit interaktiver Verifikation vorweisen muss. Die Checkverpflichtungen werden dabei automatisch aus den modellierten Eigenschaften abgeleitet.

Da zwischen dem formalen Modell und dem Codeskelett eine Refinement-Beziehung vorliegt, gelten die formal verifizierten Eigenschaften auch für das Codeskelett.

AUSBAU DES CODESKELETTS ZUR FINALEN ANWENDUNG Damit das generierte Codeskelett als eine verteilte Anwendung auf echter Hardware lauffähig wird, werden die Abstraktionen der plattformspezifischer Funktionalität durch Code ersetzt, der diese implementiert. Der IFlow-Ansatz garantiert, dass sich die Informationsflusseigenschaften des Codeskeletts und des formalen Modells auf die finale Anwendung vererben. Dazu wird die vom Entwickler bereitgestellte, manuelle Implementierung der noch fehlenden Funktionalität zusätzlich auf die im Modell festgelegten Anforderungen hin analysiert, bevor sie in die Anwendung integriert wird. Dadurch wird sichergestellt, dass die händische Implementierung die Informationsflusseigenschaften der Anwendung nicht verletzt. Anschließend können die Anwendungskomponenten als Android-Apps und Java-Webservices auf entsprechende Hardware installiert werden, und erfüllen die garantierten Eigenschaften der Gesamtanwendung.

4

BESONDERE HERAUSFORDERUNGEN

4.1 WAS BEDEUTET INFORMATIONSFLUSSSICHERHEIT?

Das Ziel des IFlow-Ansatzes ist es, die modellgetriebene Entwicklung von realen und informationsflusssicheren Anwendungen zu ermöglichen. Die wohl bekannteste Formalisierung der Informationsflusssicherheit ist die *Nichtinterferenz*-Eigenschaft nach Goguen und Meseguer [31]. Diese sagt aus, dass die öffentliche Systemausgabe, die ein Angreifer beobachten kann, nicht von geheimen Informationen oder Systemaktionen abhängt. Die Informationsflusssicherheit einer realen Anwendung geht jedoch über diese formale Eigenschaft hinaus. Geheime Daten müssen beispielsweise oft öffentlicher gemacht (d.h., *deklassifiziert*) werden können. So hängt etwa das Ergebnis der Funktion, die bei der Anmeldung das Passwort des Nutzers überprüft, von dem in der Datenbank gespeicherten, geheimen Passwort ab; gleichzeitig muss ein öffentlicher Beobachter dieses Ergebnis erfahren dürfen, um sich anmelden zu können, weshalb es für ihn deklassifiziert werden muss. Ein Ansatz zur Entwicklung realer, informationsflusssicherer Systeme muss daher die Spezifikation und Verifikation von Eigenschaften bezüglich der Sicherheit solcher Deklassifikation ermöglichen.

4.2 PRAKTISCHE ANWENDBARKEIT FORMALER METHODEN UND STATISCHER CODEANALYSE FÜR INFORMATIONSFLUSSKONTROLLE

Formale Frameworks für Informationsflusssicherheit wie [31, 62, 63, 80] sind recht komplex, weshalb die Formalisierung realer Systeme und ihrer Eigenschaften nicht trivial ist [72]. Ein Ansatz, der sich an einen Entwickler von mobilen Apps und Webservices richtet, muss daher eine Programmier- oder Modellierungssprache zur Verfügung stellen, dessen Elemente in einem solchen Framework instantiiert werden können, und sie dennoch intuitiv und zugänglich bleibt.

Jedoch kann nicht davon ausgegangen werden, dass die meisten Entwickler von mobilen Apps und Webservices mit formaler Methoden und Verifikation von Informationsflusseigenschaften vertraut sind. Um auch solche Entwickler beim Entwurf informationsflusssicherer Anwendungen zu unterstützen, sowie die manuelle Implementierung von Teilen solcher Anwendungen zu ermöglichen, muss der Ansatz auch vollautomatische Analysetechniken zur Verfügung stellen. Aktuelle Tools wie JOANA [32], die Korrektheitsgarantien bezüg-

lich ihrer Informationsflussanalysealgorithmen geben, haben jedoch wie alle statische Codeanalysetechniken eine Reihe von Einschränkungen, u.a. auch bezüglich des Umfangs und Komplexität des analysierbaren Codes. Daher sind geeignete Abstraktionstechniken gefordert, die es ermöglichen, den komplexen, plattformspezifischen Code von verteilten Anwendungen bestehend aus mobilen Apps und Webservices zu vereinfachen, damit diese vollautomatisch auf Informationsflussverletzungen geprüft werden können. Gleichzeitig muss sichergestellt werden, dass die Eigenschaften, die für die abstrakte Variante des Codes gezeigt wurden, sich auf die finale Anwendung vererben.

4.3 SCHNITTSTELLE ZUM ENDNUTZER

Die Zielgruppe der mit IFlow entwickelten Anwendungen sind Endnutzer von mobilen Apps und Webservices, die ihre Privatsphäre ernst nehmen. Jedoch bedeutet Privacy nicht nur das Verbergen sensitiver Information, sondern auch die Möglichkeit, die Freigabe solcher Information zu kontrollieren, sowie die Transparenz des Systems bezüglich ihrer weiteren Verwendung [104]. Somit muss ein IFlow-Entwickler nicht nur Informationsflusseigenschaften garantieren, sondern diese dem Nutzer auch verständlich darlegen können. Zudem soll der Nutzer weitere Informationsflussanforderungen selbst definieren können, und die Kontrolle über die Freigabe seiner Daten behalten.

Der Fokus von IFlow liegt mitunter auf mobilen Apps, die sich von anderen sicherheitskritischen Anwendungen wie Webservices oder Smartcard-Applets dadurch unterscheiden, dass sie die direkte Interaktion mit dem Nutzer ermöglichen. Daher muss die Sicherheit und Privatsphäre bei der Mensch-Maschine-Interaktion berücksichtigt werden, die die für die *praktische* Sicherheit von *realen* Anwendungen eine ausschlaggebende Rolle spielt [17, 58, 100]. Dazu zählen die Sicherheit bei der Eingabe geheimer Nutzerinformation und ihr Bezug zu den garantierten Sicherheitseigenschaften, sowie fälschungssichere Benutzeroberflächen und sinnvolle Sicherheitshinweise.

5 TOOLUNTERSTÜTZUNG

Im Rahmen des IFlow-Projekts entstand die prototypische Implementierung einer Reihe von Tools und Bibliotheken, die den Entwickler bei der Modellierung und den Modelltransformationen, der Codegenerierung, der Informationsflussanalyse und -verifikation, sowie dem Deployment unterstützen.

5.1 UML-PROFIL UND -BIBLIOTHEK VON MODELFLOW

Die Modellierungssprache MODELFLOW definiert eine Reihe von IFlow-spezifischen Daten- und Stereotypen sowie Operationen, die für die Spezifikation der Anwendung, der Struktur und des Verhaltens ihrer Komponenten, und ihrer Sicherheitseigenschaften notwendig sind.

Die MODELFLOW-spezifischen Stereotypen sind in dem IFlow-UML-Profil *IFlow.profile* festgelegt, während die Datentypen und Klassenoperationen in der UML-Bibliothek *IFlow.elements* spezifiziert sind, die der Modellierer in sein UML-Modell einbinden muss. Abschnitt 9.2 gibt einen Überblick über die Stereotypen des IFlow-UML-Profils, sowie die in MODELFLOW vordefinierten Datentypen und Operationen.

Als UML-Editor wurde im Rahmen des IFlow-Projekts der UML-Editor *MagicDraw* von No Magic genutzt, der den Export des Anwendungsmodells in den *XMI*-Format des Eclipse Modeling Frameworks für weitere Modelltransformationen unterstützt.

Eclipse Modeling Framework *ist ein Projekt der* Eclipse Foundation *zur Modellierung und Codegenerierung*

5.2 MODELLVALIDIERUNG, MODELLTRANSFORMATIONEN, UND CODEGENERIERUNG

Die Validierung und Transformation des Anwendungsmodells (siehe Abbildung 2) erfolgt in IFlow vollautomatisch, und wurde auf Basis des Eclipse Modeling Frameworks (EMF) implementiert. Hierzu wurde ein Eclipse-Plugin entwickelt, das das MODELFLOW-Anwendungsmodell im *XMI*-Format einliest (1), und in eine Instanz des IFlow-eigenen MEL*-Metamodells auf Basis des *Ecore*-Modells von EMF transformiert (2). Dabei wird das Anwendungsmodell sowie der eingebettete Code der domänenspezifischen Sprache MEL* von IFlow geparst, annotiert, und validiert.

Das MEL*-Metamodell bildet alle Elemente der MODELFLOW-Modellierungssprache ab, und schafft die Grundlage für weitere Modelltransformationen und Codegenerierung. Die Modellelemente sowie

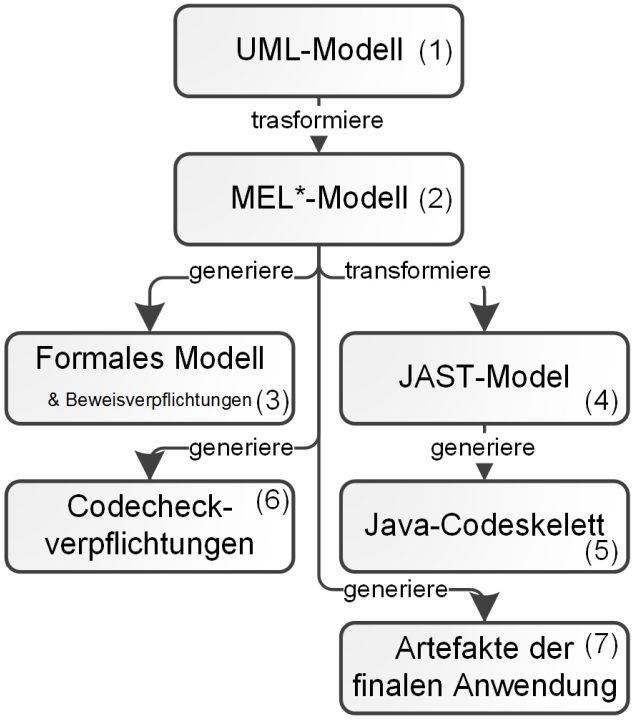


Abbildung 2: Modelltransformationen in IFlow

die Modellierungsrichtlinien von MODELFLOW, die in diesem Transformationsschritt validiert werden, sind in Teil iii im Detail erläutert.

XPand *ist eine Modell-zu-Text-Transformationssprache*

Aus dem MEL*-Modell der Anwendung wird mit XPand ein formales Modell auf Basis einer ASM und algebraischer Spezifikationen, sowie dessen Beweisverpflichtungen generiert, die mit dem interaktiven Beweiser KIV eingelesen und verifiziert werden können (3).

QVTo *(Operational Query/View-/Transform) ist eine Modell-zu-Modell-Transformationssprache*

Zudem wird aus dem MEL*-Modell mit QVTo ein abstrakter Java-Syntaxbaum (engl. *Java Abstract Syntax Tree, JAST*) des Java-Codes generiert, der zur ASM funktional äquivalent ist (4). Anschließend wird aus dem JAST-Modell mit Hilfe einer Modell-zu-Code-Transformation das Java-Codeskelett der Anwendung generiert (5). Die Checkverpflichtungen für das Codeskelett werden mit XPand direkt aus dem MEL*-Modell abgeleitert (6).

Damit das Codeskelett als eine verteilte Anwendung auf den Zielpflattformen lauffähig ist, werden im letzten Schritt zusätzlich notwendige Artefakte wie Android- und Webservice-Konfigurationsdateien mit XPand generiert (7).

Alle Transformationsschritte können aus Eclipse heraus mit Hilfe eines IFlow-Kontextmenüs komfortabel aufgerufen werden. Die Ergebnisse der Modellvalidierung werden dabei in einer Eclipse-Konsole angezeigt.

5.3 IFLOW-BIBLIOTHEK

Das aus einem MODELFLOW-Modell generierte Codeskelett ist nicht sofort lauffähig, sondern dient zunächst zur automatischen Informationsflussanalyse der Anwendung. Dazu nutzt es eine Reihe von Codeabstraktionen, die von echter, plattformspezifischer Funktionalität abstrahieren, aber ihre Informationsflüsse simulieren. Im Rahmen dieser Arbeit ist eine Java-Bibliothek solcher Codeabstraktionen entstanden, die vom Codeskelett genutzt wird.

Es existieren zwei weitere Versionen dieser Bibliothek, die dieselbe Schnittstelle besitzen, und die für die finale, lauffähige Anwendung notwendige, plattformspezifische Funktionalität implementieren. Sie stellen die erforderliche Funktionalität zur Verfügung, um die Anwendungskomponenten des Codeskeletts als mobile Android-Apps und Play!-Webservices zu installieren.

Details zu den implementierten Abstraktionstechniken sind in Abschnitt 17.2 beschrieben.

Android *ist eine Plattform für mobile Anwendungen,* `android.com`

Play! *ist ein Java und Scala-Framework für Webservices,* `playframework.com`

5.4 ANALYSEWERKZEUGE

Zusätzlich zu den in IFlow eingesetzten Verifikations- und Analysetools KIV und JOANA wird das Codeanalyse- und instrumentierungstool *Soot* genutzt. Dazu wurde das Java-Tool *MMChecker* implementiert, das auf Soot aufbaut, und die manuelle Implementierung der im Codeskelett fehlender Funktionalität u.a. bezüglich der Zugriffsverletzungen auf die Zielplattform-API prüft (siehe Unterabschnitt 16.2.3).

Zusätzlich wurde das Java-Tool *DeclMod* entworfen, das den mobilen Code der finalen Anwendung instrumentiert. Das Tool ermöglicht es dem Nutzer, Deklassifikationseigenschaften außerhalb des IFlow-Entwicklungszyklus mit Hilfe statischer Codeanalyse zu prüfen (siehe Unterunterabschnitt 16.2.4.3), und unerwünschte Informationsleaks über Deklassifikationsfunktionen zu verhindern (siehe Unterunterabschnitt 16.2.4.4).

5.5 ANDROID-SOFTWARE

Um dem Endnutzer einer IFlow-App die sichere Dateneingabe zu ermöglichen, wurde die Android-App *SecureInput* entwickelt. Diese setzt eine Reihe von Sicherheitsmechanismen ein, um Phishing- und Man-in-the-Middle-Angriffe zu verhindern, die in Android möglich sind, und das Stehlen sensitiver Benutzereingabe erlauben. Die Anwendung informiert den Nutzer, welche Information angefordert wird, und zeigt ihm die hierfür garantierten Informationsflusseigenschaften an (siehe Unterunterabschnitt 17.2.2.4).

Die Android-App *MyFlows* erlaubt es dem Endnutzer, eigene anwendungsspezifische Informationsflussanforderungen für eine IFlow-App zu definieren. Die Anforderungen werden mit der MODELFLOW-Syntax für Informationsflusseigenschaften festgelegt, und werde für den automatischen Check der IFlow-App im Informationsflussanalysetool-agnostischen Format *RIFL* [25] exportiert (siehe Unterabschnitt 16.2.4).

6 ARCHITEKTUR EINER IFLOW-ANWENDUNG

Eine IFlow-Anwendung besteht aus einer Reihe von Anwendungsmodulen und -komponenten, die in Abbildung 3 gezeigt werden, und im Folgenden erläutert werden.

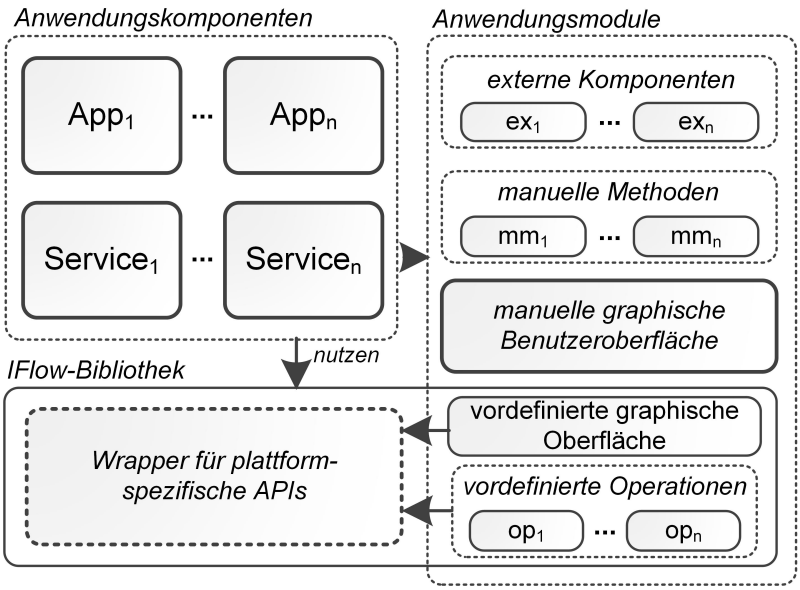


Abbildung 3: Architektur einer IFlow-Anwendung

ANWENDUNGSKOMPONENTEN Als Anwendungskomponente wird der logische Kern einer mobilen App bzw. eines Webservices bezeichnet, der mit MODELFLOW spezifiziert wird. Eine IFlow-Anwendung kann beliebig viele Anwendungskomponenten enthalten, besteht aber üblicherweise aus mindestens einer mobilen App und einem Webservice. Solche Komponenten importieren die IFlow-Bibliothek, und können mehrere Anwendungsmodule nutzen, um die gewünschte Funktionalität zu implementieren.

IFLOW-BIBLIOTHEK Die IFlow-Bibliothek stellt eine Reihe von vordefinierten Klassen und Methoden zur Verfügung, die von jeder MODELFLOW-Anwendung verwendet werden können. Neben vordefinierten Operationen und graphischer Benutzeroberfläche implementiert sie einen Wrapper für plattformspezifische APIs, die den automatischen Codecheck bzw. die Installation der Anwendungskomponenten als mobile Apps und Webservices auf echter Hardware ermöglichen.

ANWENDUNGSMODULE IFlow unterstützt eine Reihe von verschiedenen Anwendungsmodulen, mit denen eine Anwendungskomponente interagieren kann. Dazu zählen die folgenden *anwendungsspezifischen* Module:

- **Manuelle Methoden** sind Methoden, die vom Entwickler händisch implementiert werden müssen, um anwendungs- und plattformspezifische Funktionalität bereitzustellen
- **Externe Komponenten** sind externe Apps und Services, deren Funktionalität nicht auf Modellebene spezifiziert wird
- **Manuelle graphische Benutzeroberfläche** wird vom Entwickler händisch implementiert, und kann Anwendungskomponenten aufrufen

Hinzu kommen die folgenden *anwendungsunabhängigen* Module, die als Teil der IFlow-Java-Bibliothek implementiert sind:

- **Vordefinierte graphische Oberfläche** kann von einer mobilen Anwendungskomponente angezeigt werden, um die Interaktion mit dem Nutzer zu ermöglichen
- **Vordefinierte Operationen** sind mathematische, kryptographische, und plattformspezifische Funktionen, die dem Modellierer von MODELFLOW zur Verfügung gestellt werden

7 VORGEHEN BEI DER ENTWICKLUNG MIT IFLOW

Bei IFlow liegt der Schwerpunkt auf Privacy by Design. Das bedeutet, dass die Privatheit der von der Anwendung verwalteten Informationen schon in frühen Stadien der Entwicklung berücksichtigt werden soll. Insbesondere muss dem Entwickler der Anwendung bereits dann bewusst sein, welche Informationen als sensitiv angesehen werden müssen, und für welche Systemkomponente sie nicht lesbar sein dürfen.

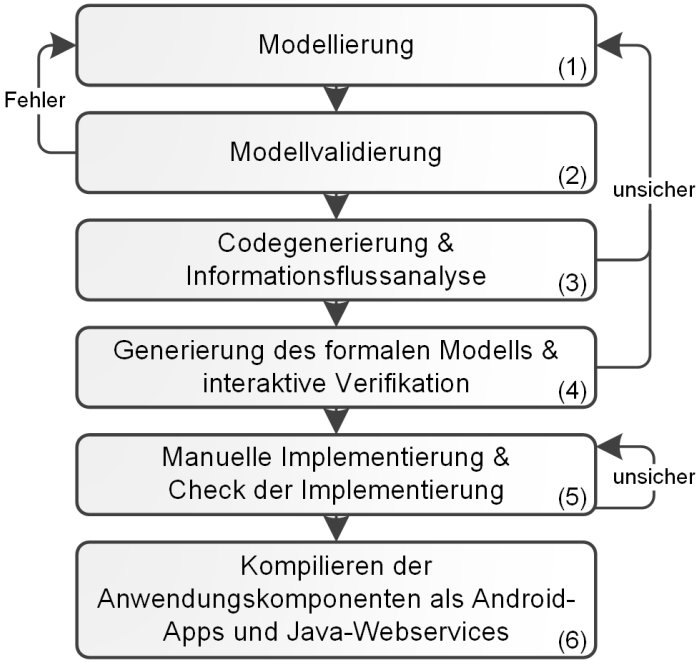


Abbildung 4: Vorgehen bei der Entwicklung mit IFlow

Abbildung 4 zeigt das empfohlene Vorgehen bei der Entwicklung von informationsflusssicheren Anwendungen mit IFlow.

Im ersten Schritt wird das Anwendungsmodell mit UML entworfen, wobei das IFlow-UML-Profil genutzt wird (1).

Danach wird mit dem IFlow-Plugin für Eclipse die Transformation des Anwendungsmodells zum MEL-Modell angestoßen, wodurch es validiert wird (2). Werden hierbei Fehler gemeldet, so muss der Modellierer diese beheben, und das Modell erneut validieren.

Werden bei der Validierung keine Fehler mehr gemeldet, so kann das Codeskelett und die Codecheckverpflichtungen für das Informationsflusskontrolltool JOANA generiert werden (3). Meldet JOANA eine Informationsflussverletzung, so muss der Modellierer zum Schritt

(1) zurückkehren, und das Modell entsprechend korrigieren. Bei komplexen Projekten bietet sich der iterative Prozess an, bei dem neue Anwendungsfunktionen erst dann hinzugefügt werden, wenn die bereits bestehende Anwendung durch automatische Codeanalyse als sicher garantiert wurde.

Wurden Informationsflusseigenschaften modelliert, die nicht automatisch gecheckt werden können, so muss das formale Modell generiert werden, und anhand der generierten Beweisverpflichtungen mit dem interaktiven Beweiser KIV formal verifiziert werden (4). Gelingt dies nicht, so ist die Anwendung unsicher; der Modellierer muss zum Schritt (1) zurückkehren, und das Modell entsprechend korrigieren. Die formale Verifikation der Eigenschaften, die nicht durch Codeanalyse sichergestellt werden können, ist oft zeitaufwändiger, und sollte erst dann gemacht werden, wenn die Anwendung vollständig modelliert und erfolgreich gecheckt wurde.

Wurde das Codeskelett erfolgreich gecheckt und das formale Modell verifiziert, so muss der Entwickler die noch fehlende Funktionalität (falls vorhanden) von Hand implementieren, und diese anhand der generierten Checkverpflichtungen checken (5). Ist der Check nicht erfolgreich, so muss die manuelle Implementierung entsprechend angepasst werden.

Sind alle Checks erfolgreich, so kann das Codeskelett mit der manuellen Implementierung zu finalen Anwendungskomponenten als mobile Apps und Java-Webservices kompiliert werden (6).

8 VERWANDTE ARBEITEN

Moebius stellt mit SecureMDD [65, 66] einen Ansatz zur modell-getriebenen Entwicklung sicherheitskritischer Smartcard-Anwendungen und kryptographischer Protokolle vor, der von Borek auf die Entwicklung von sicheren Webservices erweitert wurde [11]. SecureMDD umfasst die Modellierung, Verifikation, automatisches Model-Checking, und Codegenerierung einer Smartcard- oder Webservice-Anwendung. Der Ansatz und die Modellierungssprache von SecureMDD ähneln dem IFlow-Ansatz sowie dessen Sprache MODELFLOW, jedoch haben die Ansätze unterschiedliche Ansprüche. Während mit SecureMDD die *Sicherheit der modellierten Protokolle gegenüber einem externen Angreifer* formal verifiziert werden kann, indem das komplette Verhalten der Protokollteilnehmer spezifiziert wird, liegt in IFlow der Fokus auf der *Informationsflusssicherheit einer unterspezifizierten verteilten Anwendung* bestehend aus Apps und Services, deren Code händisch ergänzt und vollautomatisch gecheckt werden kann. IFlow-Anwendungen sollen demnach keine unerwünschten Datenlecks gegenüber internen Systemteilnehmern (bzw. externen Beobachtern des Systems) aufweisen.

Jürjens stellt mit UMLSec [46] einen modell-getrieben Ansatz zur Entwicklung sicherer Anwendungen mit UML. Der Fokus liegt dabei auf kryptographischer Sicherheit, jedoch lässt der Ansatz ebenfalls die Spezifikation einfacher Informationsflusseigenschaften durch Modellannotationen zu. UMLSec stellt einen Kern-UML-Profil zur Verfügung, der eine Reihe von Standardeigenschaften unterstützt, jedoch für spezifische Anwendungsdomänen und Sicherheitseigenschaften explizit erweitert werden muss. Die Verbindung zur Codeebene wird u.a. durch modellbasiertes Testen der Implementierung des modellierten Systems geschaffen.

Seehusen [84, 85] integriert ebenfalls Informationsflusssicherheit in einen modell-getriebenen Ansatz. Dabei nutzt er von UML inspirierte Zustandsmaschinen zur Modellierung von sicheren Systemen, sowie Sequenzdiagramme zur Spezifikation von Sicherheitspolicies. Die unterstützten Informationsflusseigenschaften des modellierten Systems bzgl. Geheimhaltung von Systemereignissen (ohne Betrachtung der Deklassifikation) werden auf Modellebene betrachtet, wobei die formale Semantik des Anwendungsmodells auf STAIRS [39] basiert. Die Sicherheitsbetrachtung auf Codeebene beschränkt sich bei dem Ansatz auf Mechanismen zum dynamischen Durchsetzung der modellierten Policy bei der händischen Implementierung des Systems.

Alghathbar et al. [1] beschreiben das Logik-basierte System Flow-UML zur Spezifikation und Verifikation von Informationsflusseigenschaften mit Horn-Formeln für UML-Anwendungsmodelle. Dabei werden die Horn-Formeln gegen den Informationsfluss geprüft, der aus UML-Sequenzdiagrammen abgeleitet wird. Der Ansatz beschränkt sich auf die Sicherheitsbetrachtung von abstrakten Systemen auf Modellebene, sowie auf transitive Informationsflüsse ohne Betrachtung von Deklassifikation.

Teil III

MODELLIERUNG MIT MODELFLOW

9 GRUNDLAGEN

9.1 ZIELSETZUNG UND LÖSUNGSANSATZ

Das primäre Ziel bei der Entwicklung des vorgestellten Ansatzes ist die Spezifikation einer geeigneten Modellierungssprache für verteilte und informationsflusssichere Anwendungen. Eine solche Sprache soll dabei eine Reihe von Anforderungen erfüllen, die im Folgenden aufgezählt und erläutert werden.

Da der Ansatz den Entwickler einer sicheren Anwendung unterstützen soll, muss die dafür ausgelegte Modellierungssprache möglichst intuitiv benutzbar sein, um den Einarbeitungsaufwand minimal zu halten. Dafür ist es von Vorteil, wenn sich die Sprache auf bereits bekannte Modellierungs- und Programmiersprachkonzepte stützt. Dabei soll der Modellierer möglichst einfach den Sicherheitslevel der von der Anwendung verwalteten Informationen festlegen können, um bereits in der Designphase die Informationsflusssicherheit des Gesamtsystems zu berücksichtigen (*Privacy by Design*).

Das Anwendungsmodell muss eine geeignete Abstraktion der finalen Anwendung darstellen, die verteilt auf Smartphones und Webservices ausführbar sein soll. Dabei soll nur das nötigste modelliert werden müssen, um daraus das formale Modell der Anwendung zu generieren, mit dem sinnvolle Sicherheitseigenschaften der Anwendung bewiesen werden können. Andererseits muss das Modell feingranular genug sein, um daraus den (nahezu) vollständigen Code der finalen, verteilten Anwendung generieren zu können, für den die bewiesenen Eigenschaften gelten, wobei möglichst realistische Anwendungen unterstützt werden sollen. Gleichzeitig soll der generierte Code für automatische Informationsflussanalysetools analysierbar sein, wobei sich die Ergebnisse der Analyse auf die finale Anwendung übertragen lassen sollen.

Zuletzt soll der Ansatz und somit die Modellierungssprache benutzerzentrisch sein. Das bedeutet, dass u.a. die modellierten Informationsflusseigenschaften für den Endnutzer lesbar und verständlich sind, während die vom Benutzer eingegebenen Daten im Modell und insbesondere den modellierten Informationsflusseigenschaften als solche annotiert sein müssen.

Im Hinblick auf diese Zielsetzungen wurde die Modellierungssprache MODELFLOW im Rahmen des IFlow-Ansatzes entwickelt. Sie basiert auf dem UML-Metamodell, wird aber durch ein MODELFLOW-spezifisches UML-Profil sowie eine Reihe von vordefinierten Klassen

erweitert. Zudem erhält sie durch das daraus generierbare formale Modell eine Semantik.

Um das dynamische Verhalten der modellierten Anwendung zu spezifizieren, wurde die objektorientierte und domänenspezifische Programmiersprache „Model Extension Language“ (MEL) übernommen und erweitert, die ursprünglich im Rahmen eines Projekts zur Entwicklung kryptographisch sicherer Chipkartenanwendungen entwickelt wurde [65].

9.2 MODELLIERUNGSSPRACHE FÜR INFORMATIONSFLUSSSICHERE SYSTEME

Dieser Abschnitt gibt einen Überblick über die Modellierungssprache MODELFLOW und ihre Elemente. Dazu zählen die unterstützten Diagrammtypen (vgl. Unterabschnitt 9.2.1), Stereotypen (vgl. Unterabschnitt 9.2.2), sowie vordefinierte Datentypen und Operationen (vgl Unterabschnitt 9.2.3 und Unterabschnitt 9.2.4).

Die konkreten Modellierungsrichtlinien, die die Anwendung solcher Elemente vorgeben, werden in den folgenden Kapiteln detailliert beschrieben.

9.2.1 *Diagrammtypen*

Die folgenden UML-Diagrammtypen kommen in MODELFLOW zum Tragen:

Klassendiagramme werden zur Definition von Anwendungskomponenten und -modulen, sowie Daten- und Nachrichtentypen benutzt. Sie stellen die statische Sicht auf die Anwendung dar.

Sequenzdiagramme beschreiben das dynamische Verhalten und die Interaktion der Anwendungskomponenten.

Aktivitätsdiagramme werden in MODELFLOW verwendet, um Funktionen zum Filtern oder Deklassifikation von Daten zu spezifizieren. Zudem werden sie zur Spezifikation von Informationsflusseigenschaften und der Sicherheitsdomänen sowie derer Beziehungen verwendet.

9.2.2 *Stereotypen*

MODELFLOW definiert eine Reihe von Stereotypen, die ein Modellierer nutzen kann, um Komponenten zu definieren, Moduleinschränkungen zu spezifizieren, oder Anwendungsverhalten festzulegen.
Manche dieser Stereotypen haben zusätzliche Attribute (*Tags*), mit denen weitere Eigenschaften festgelegt werden können.

Dieser Abschnitt gibt einen Überblick über die in MODELFLOW unterstützten Stereotypen, und somit über den Funktionsumfang der Modellierungssprache.

9.2.2.1 *Strukturierung des Modells*

Das Modell wird mithilfe von Packages strukturiert, die mit den Stereotypen aus Tabelle 1 annotiert werden. MODELFLOWdefiniert fünf verschiedene Package-Typen. «`ClassDiagram`» wird zur Spezifikation von Komponenten und Datentypen verwendet, «`BehaviorDiagram`» enthält Sequenzdiagramme und Aktivitätsdiagramme, die das Anwendungsverhalten beschreiben, «`DomainDiagram`» speichert das Domänendiagramm, das die Sicherheitsdomänen der Anwendung definiert, «`InformationFlowProperties`» enthält die spezifizierten Informationsflusseigenschaften, während «`IFlowElements`» die vordefinierten Klassen und Operationen speichert.

Alle Modellelemente müssen in einem der annotierten Packages enthalten sein.

STEREOTYP	METAKLASSE	BESCHREIBUNG
«`ClassDiagramm`»	Package	annotiert das Package mit den Klassendiagrammen
«`BehaviorDiagram`»	Package	annotiert das Package mit den Sequenz- und Aktivitätsdiagrammen
«`DomainDiagram`»	Package	annotiert das Package mit dem Domänendiagramm
«`IFlowElements`»	Package	annotiert das Package mit den vordefinierten Klassen und Operationen
«`InformationFlow Properties`»	Package	annotiert das Package mit den Informationsflusseigenschaften

Tabelle 1: Stereotypen für Packages

9.2.2.2 *Komponenten*

Komponenten werden in MODELFLOW als UML-Klassen repräsentiert, die mit einem der Stereotypen aus Tabelle 2 annotiert sind. Es gibt zwei Komponententypen: «`Application`» sowie «`Service`», die jeweils den logischen Kern einer mobilen App bzw. eines Webservices repräsentiert. Eine MODELFLOW-Anwendung muss mindestens eine «`Application`»-Komponente enthalten, da der Endnutzer nicht direkt mit einem Webservice interagieren kann.

STEREOTYP	METAKLASSE	BESCHREIBUNG
«`Application`»	Class	annotiert die App-Komponente
«`Service`»	Class	annotiert die Service-Komponente

Tabelle 2: Stereotypen für Komponenten

9.2.2.3 *Module*

Komponenten können mit diversen Modulen interagieren, die jeweils mit einem Stereotyp aus Tabelle 3 gekennzeichnet werden.

Dazu zählt die *manuelle Methode*, deren Signatur als Klassenoperation mit dem «`uses`»-Stereotyp modelliert wird. Solche Methoden müssen händisch implementiert werden, wobei die Stereotyp-Tags *sources* und *sinks* die plattformspezifischen APIs (und dadurch plattformspezifische Informationsquellen und -senken) einschränken, auf die die Methode zugreifen darf.

Die *vordefinierte Benutzeroberfläche*, die als eine vordefinierte Klasse mit dem «`User`»-Stereotyp repräsentiert wird, erlaubt es dem Entwickler, eine der vordefinierten Benutzerinteraktionen wie Eingabeaufforderung oder Listenauswahl in seiner Anwendung zu nutzen. Dies erspart dem Entwickler den Implementierungsaufwand, verbessert die Präzision der Informationsflussanalyse, und garantiert den Zusammenhang zwischen den modellierten Informationsflusseigenschaften und der Benutzereingabe.

Die *manuelle Benutzeroberfläche* ist eine Anwendungsklasse mit dem «`GUI`»-Stereotyp, die es dem Entwickler erlaubt, eine händisch zu implementierende und anwendungsspezifische Benutzeroberfläche in seine Anwendung einzubinden.

Die *externe App* ist eine Klasse mit dem «`External Application`»-Stereotyp, die das Einbinden einer externen mobilen App ermöglicht, während der *externe Webservice* durch eine Klasse mit dem «`External Service`»-Stereotyp repräsentiert wird.

MODELFLOW unterstützt auch eine Reihe von vordefinierten Methoden (beispielsweise zur Abfrage der aktuellen GPS-Position in einer mobilen Anwendung), die als Klassenoperationen im «`IFlowElements`»-Package modelliert sind.

9.2.2.4 *Informationsflussannotationen*

Um Informationsflusseigenschaften für die modellierte Anwendung zeigen zu können, müssen diese im Modell festgehalten werden. Der Entwickler hat hierfür zwei Werkzeuge: Annotation des Gesamtmodells mit Sicherheitsdomänen wie *„geheim“* und *„öffentlich“*, sowie die explizite Modellierung von Eigenschaften, die den Informationsfluss zwischen spezifizierten Quellen und Senken erlauben bzw. verbieten.

STEREOTYP	METAKLASSE	BESCHREIBUNG
«`uses`» Tags: *sources*, *sinks*	Operation	beschreibt eine Klassenoperation, die händisch implementiert wird. *sources* und *sinks* beschreiben Informationsquellen und -senken aus den vordefinierten Klassen *PredefinedSources* und *PredefinedSinks*, auf die die Operation zugreifen darf
«`User`»	Class	annotiert die vordefinierte Klasse, die die vordefinierte Benutzeroberfläche repräsentiert
«`GUI`»	Class	annotiert die Klasse, die die manuell implementierte Benutzeroberfläche repräsentiert
«`ExternalApplication`» Tags: *noResult*, *optionalResult*	Class	annotiert die Klasse, die eine externe mobile App repräsentiert. *noResult* und *optionalResult* beschreiben, ob die App eine (optionale) Antwortnachricht verschickt
«`ExternalWebService`»	Class	annotiert die Klasse, die einen externen Webservice repräsentiert

Tabelle 3: Stereotypen für Anwendungsmodule

Sicherheitsdomänen helfen dem Entwickler dabei, bereits bei der Spezifikation der Anwendungsfunktionalität ihre Informationsflusssicherheit zu berücksichtigen, und sind notwendig für die formale Verifikation. Sie werden in einem Domänendiagramm als Aktivitätsknoten spezifiziert. Der erlaubte Informationsfluss bzw. die Interferenz-Relation zwischen den Domänen wird mithilfe von gerichteten Kontrollflusskanten definiert. Dabei ist eine solche Relation standardmäßig *transitiv*, außer wenn sie mit dem «`intransitive`»-Stereotyp annotiert wurde.

Explizite Informationsflusseigenschaften sprechen konkret über die für den Endnutzer relevanten Quellen und Senken von sensitiven Daten. Sie werden insbesondere für den automatischen Informationsflusscheck verwendet. Informationsflusseigenschaften werden als

Mengen von Informationsflussquellen und -senken repräsentiert, die auf Komponenten, Komponentenattribute, bzw. die Benutzereingabe Bezug nehmen können und als UML-*Send Signal Actions* bzw. -*Accept Event Actions* modelliert werden. Wird eine Quelle oder Senke mit dem «`otherSources`»- bzw. «`otherSinks`»-Stereotyp annotiert, so repräsentiert sie alle Quellen bzw. Senken, die in der Informationsflusseigenschaft nicht explizit vorkommen.

Ob der Informationsfluss zwischen einer Quelle und Senke stattfinden darf, wird mit den Stereotypen «`allowedFlow`» bzw. «`noFlow`» festgehalten. Diese annotieren die UML-*Control Flows*, die die Quellen und Senken in einer Informationsflusseigenschaft verbinden. Darf der Informationsfluss nur über eine konkrete Deklassifikationsmethode erfolgen, so muss diese im Klassendiagramm mit dem «`declassify`»-Stereotyp annotiert werden. Der mit «`allowedFlow`» erlaubte Informationsfluss muss dann über einen mit «`via`» annotierten Aktivitätsknoten erfolgen, der auf diese Methode verweist. Ihr Stereotyp-Tag *filter* legt dabei fest, ob es sich dabei um eine bewiesen sichere Filterfunktion handelt.

Benutzernachrichten sowie Datentypklassen, die die Benutzereingabe repräsentieren, können mit einer Klartextbeschreibung mit Hilfe des «`labeled`»-Stereotyps annotiert werden.

9.2.2.5 *Datentypannotationen*

MODELFLOW unterstützt eine Reihe von speziellen Datentypen, die durch die Stereotypen aus Tabelle 5 gekennzeichnet werden.

Darunter fallen die vordefinierten Listen von plattformspezifischen Informationsflussquellen und -senken (wie etwa das Dateisystem oder die vom mobilen Gerät verwalteten, anwendungsunabhängigen Kontaktdaten), die im «`IFlowElements`»-Package als UML-*Enumerations* repräsentiert sind. Diese sind mit dem «`PredefinedSources`»- bzw. «`PredefinedSinks`»-Stereotyp annotiert.

Annotiert man ein Attribut einer Klasse mit dem «`key`»-Stereotyp, so werden Listenattribute von diesem Datentyp als eine Map interpretiert.

Spezielle Datentypen, die als Nachrichtenklassen verwendet werden, erben von einer abstrakten Klasse, die entweder mit dem «`Message`»- oder dem «`Usermessage`»-Stereotyp annotiert ist. Dabei sind die Nachrichtenklassen, die von einer «`Message`»-Klasse erben, anwendungsspezifisch und werden zur Kommunikation zwischen Komponenten verwendet. Nachrichtenklassen, die von einer «`Usermessage`»-Klasse erben, sind vordefiniert und repräsentieren die Benutzerinteraktionsmöglichkeiten mit einer vordefinierten Benutzeroberfläche. Klassen, die Authentifikationsdaten für den Zugriff auf zugriffsbeschränkte Maps enthalten, werden mit dem Stereotyp «`AuthData`» versehen. Der Datentyp des Schlüssels einer solchen Map muss hierzu dieser Klasse entsprechen.

STEREOTYP	METAKLASSE(N)	BESCHREIBUNG
«intransitive»	ControlFlow	erlaubt den intransitiven Informationsfluss zwischen zwei Domänen
«noFlow»	ControlFlow	verbietet den Informationsfluss zwischen Informationsquelle und -senke
«allowedFlow»	ControlFlow	erlaubt den Informationsfluss zwischen Informationsquelle und -senke
«otherSinks»	AcceptEventAction	bezeichnet alle in der Informationsflusseigenschaft nicht explizit aufgelisteten Quellen
«otherSources»	SendSignalAction	bezeichnet alle in der Informationsflusseigenschaft nicht explizit aufgelisteten Senken
«via» Tag: *filter : Boolean*	CallBehaviorAction	annotiert den Verweis auf eine Deklassifikationsmethode, über die Information fließen darf. Der Tag *filter* legt fest, ob es sich dabei um eine bewiesen sichere Filterfunktion handelt
«declassify»	Operation	annotiert eine Klassenoperation als eine Deklassifikationsmethode
«labeled» Tag: *label : String*	Class, Message, Property	annotiert ein Modellelement mit einem Klartext (*label*), der in Informationsflusseigenschaften und bei Benutzereingaben angezeigt wird
«PredefinedSources»	Enumeration	annotiert eine Enumeration mit vordefinierten, plattformspezifischen Quellen
«PredefinedSinks»	Enumeration	annotiert eine Enumeration mit vordefinierten, plattformspezifischen Senken

Tabelle 4: Stereotypen für Informationsfluss

STEREOTYP	METAKLASSE(N)	BESCHREIBUNG
«key»	Property	annotiert ein Klassenattribut als den Schlüssel einer Map
«Message»	Class	annotiert eine abstrakte Klasse als Nachrichtendatentyp
«Usermessage»	Class	annotiert eine abstrakte Klasse als Nachrichtendatentyp für die Kommunikation mit der vordefinierten Benutzeroberfläche
«AuthData»	Class	annotiert eine Klasse als Authentifikationsdaten, die beim Zugriff auf eine zugriffsbeschränkte Map benötigt werden

Tabelle 5: Stereotypen für Datentypen

9.2.3 *Vordefinierte primitive Datentypen*

MODELFLOW definiert vier primitive Datentypen, die der Entwickler nutzen kann, um etwa Klassenattribute oder Operationsparameter zu typisieren. Diese sind *Integer*, *Boolean*, *String*, und *Double*, abgebildet in Abbildung 5: Mit *Integer* können ganzzahlige Werten repräsentiert werden, mit dem *Double*-Datentyp Gleitkommazahlen, mit *Boolean* boolesche Werte, und mit *String* Zeichenketten. Der Standardwert der Variablen oder Attribute von diesen Datentypen kann Tabelle 6 entnommen werden.

Abbildung 5: Primitive Datentypen

9.2.4 *Vordefinierte Klassen und Operationen*

MODELFLOW definiert auch eine Reihe von komplexen Datentypen und Operationen, die bei der Anwendungsmodellierung verwendet werden können und in Abbildung 6 abgebildet sind.

DATENTYP	BESCHREIBUNG	STANDARDWERT
Integer	speichert ganzzahlige Werte	`0`
Double	speichert eine Gleitkommazahl	`0`
Boolean	speichert einen booleschen Wert	`false`
String	speichert eine Zeichenkette	„“ (leerer String)

Tabelle 6: Primitive Datentypen und ihre Beschreibung

- *IFlowData* ist eine abstrakte Klasse, von der alle vordefinierten sowie anwendungsspezifischen Datentypen automatisch erben.
- *User* repräsentiert die vordefinierte Benutzeroberfläche. Ihre Attribute *input*, *singleSelection*, *multipleSelection*, und *confirmRelease* bilden jeweils die Benutzereingabe, -auswahl(en), und -bestätigung ab.
- *GPSPos* repräsentiert eine Ortsangabe, wobei ihre Attribute *latitude* und *longitude* jeweils die geographische Länge bzw. Breite repräsentieren.
- *IFlowUtil* stellt eine Reihe von Operationen zur Verfügung, die der Entwickler in seinem Modell nutzen kann.

MATHEMATISCHE OPERATIONEN Die folgenden in MODELFLOW unterstützten mathematischen Operationen entsprechen den gleichnamigen mathematischen Funktionen in Java:

- *round(Double x) : Integer* rundet die Zahl x auf eine Ganzzahlige Zahl
- *abs(Double x) : Double* gibt den absoluten Wert der Zahl x zurück
- *toRadians(Double x) : Double* gibt den Wert der Zahl x in Radiant zurück
- *sqrt(Double x) : Double* gibt die Wurzel der Zahl x zurück
- *atan2(Double x, Double y) : Double* gibt den Wert der Arkusfunktion mit den Parametern x und y zurück
- *sin(Double x) : Double* gibt den Wert der Sinusfunktion mit dem Parameter x zurück
- *cos(Double x) : Double* gibt den Wert der Kosinusfunktion mit dem Parameter x zurück
- *toString(Double x) : String* gibt die String-Repräsentation der Zahl x zurück

LISTENOPERATIONEN MODELFLOW bietet eine Unterstützung für Listen- und Map-Datentypen und stellt eine Reihe von Operationen zur Verfügung, mit denen solche Listen und Maps manipuliert werden können. Die folgenden Operationen können auf eine Variable oder ein Attribut mit dem Listentyp angewandt werden:

- *add(e : IFlowData)* fügt ein Element *e* zur Liste hinzu
- *contains(e : IFlowData) : Boolean* überprüft, ob ein Element *e* in der Liste vorkommt
- *get(i : Integer) : IFlowData* gibt ein Element aus der Liste an Position *i* zurück
- *isEmpty() : Boolean* überprüft, ob die Liste leer ist
- *remove(e : IFlowData)* entfernt das Element *e* aus der Liste
- *set(i : Integer, e : IFlowData)* ersetzt das Element an Position *i* mit Element *e*
- *size() : Integer* gibt die Anzahl der Elemente in der Liste zurück

Die folgenden Operationen können auf eine Variable oder ein Attribut mit dem Maptyp angewandt werden:

- *containsKey(key : IFlowData) : Boolean* überprüft, ob ein Element mit dem Schlüssel *key* in der Map vorhanden ist
- *put(key : IFlowData, value : IFlowData)* setzt das Element *value* mit dem Schüssel *key* in der Map
- *get(key : IFlowData) : IFlowData* gibt das Element mit dem Schlüssel *key* zurück
- *size() : Integer* gibt die Anzahl der Elemente in der Liste zurück

PLATTFORMSPEZIFISCHE OPERATIONEN

- Die Operation *startGPSTracking()* beginnt das Aufzeichnen der aktuellen Position, und speichert diese in einer internen Liste der App, die diese Operation aufruft.
- Die Operation *stopGPSTracking() : GPSPos[*]* gibt diese Liste zurück und stoppt die Aufzeichnung.

VER- UND ENTSCHLÜSSELUNG

- *SymmKey*, *PublicKey*, *PrivateKey* erben von der abstrakten Oberklasse *Key*, und repräsentieren jeweils einen symmetrischen, öffentlichen, und privaten kryptographischen Schlüssel.

- *Encrypted* repräsentiert symmetrisch oder hybrid verschlüsselte Daten.

Bei hybrider Verschlüsselung wird der Klartext mit einem zufälligen Schlüssel symmetrisch verschlüsselt, der seinerseits mit einem öffentlichen Schlüssel verschlüsselt wird

- *SymmEncryption* ist eine Hüllklasse für einen symmetrischen Schlüssel, der in dem Attribut *symmKey : SymmKey* der Hüllklasse gespeichert wird. Sie stellt dem Modellierer die symmetrische Verschlüsselungsoperation *encrypt(data : IFlowData) : Encrypted* sowie die symmetrische Entschlüsselungsoperation *decrypt(enc: Encrypted) : IFlowData* zur Verfügung.
- *Decryptor* ist eine Hüllklasse für ein Schlüsselpaar, das in den Attributen *privateKey : PrivateKey* und *publicKey : PublicKey* der Hüllklasse gespeichert wird. Sie stellt die Operation *decrypt(enc : Encrypted) : IFlowData* zur Verfügung, die die übergebenen Daten mit dem geheimen Schlüssel *privateKey* hybrid entschlüsselt. Das Ergebnis der *decrypt*-Operation muss immer an ein Klassenattribut zugewiesen werden. Außerdem gibt sie mit der Operation *getEncryptor() : Encryptor* eine Instanz der *Encryptor*-Klasse zurück, die ihren öffentlichen Schlüssel enthält.
- *Encryptor* ist eine Hüllklasse für einen öffentlichen Schlüssel, der in dem Attribut *publicKey : PublicKey* der Hüllklasse gespeichert wird. Sie stellt die Operation *decrypt(data : IFlowData) : Encrypted* zur Verfügung, die die übergebenen Daten mit dem öffentlichen Schlüssel *publicKey* hybrid verschlüsselt.
- *Nonce* repräsentiert eine kryptographische Nonce, also eine Zufallszahl, die im System nur einmal verwendet werden darf.
- Die Operation der *IFlowUtil*-Klasse *generateDecryptor() : Decryptor* generiert ein asymmetrisches Schlüsselpaar, und gibt es als Instanz der *Decryptor*-Klasse zurück.
- Die Operation *generateEncryption() : SymmEncryption* generiert einen symmetrischen Schlüssel, und gibt ihn als eine Instanz der *SymmEncryption*-Klasse zurück.
- Die Operation *generateNonce() : Nonce* generiert eine neue Nonce und gibt diese zurück.

SIGNIERUNG

- *Signature* repräsentiert eine kryptographische Signatur.
- *Signer* ist eine Hüllklasse für ein asymmetrisches Schlüsselpaar, das in ihren Attributen *signKey : PrivateKey* und *verifyKey : PublicKey* gespeichert wird. Sie stellt die Operation *sign(data : IFlowData)* zur Verfügung, der die übergebenen Daten mit dem geheimen Schlüssel *signKey* signiert. Außerdem gibt sie mit *getVerifier() : Verifier* eine Instanz der *Verifier*-Klasse zurück, die ihren öffentlichen Schlüssel enthält.

- *Verifier* ist eine Hüllklasse für einen öffentlichen Schlüssel, der in ihrem Attribut *verifyKey : PublicKey* gespeichert ist. Sie stellt die Operation *verify(sig : Signature, data : IFlowData) : Boolean* zur Verfügung, die die Signatur *sig* von den Daten *data* mit dem öffentlichen Schlüssel *verifyKey* überprüft.
- Die Operation der *IFlowUtil*-Klasse *generateSigner() : Signer* generiert ein asymmetrisches Schlüsselpaar zur Signierung, und gibt es als Instanz der *Signer* zurück.

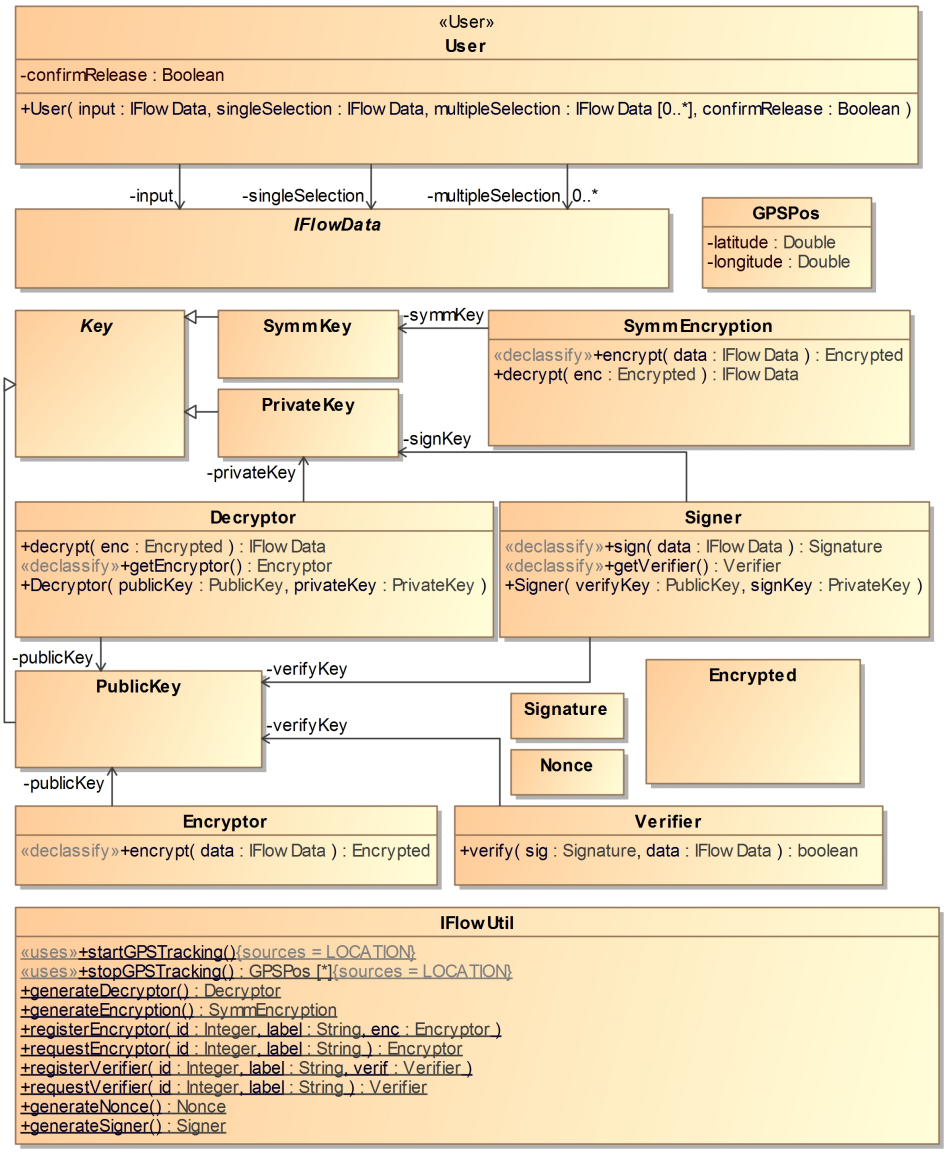


Abbildung 6: Vordefinierte Datentypen und Operationen

BENUTZERNACHRICHTEN Zur Interaktion mit dem Anwendungsmodul *vordefinierte Benutzeroberfläche*, die durch die vordefinierte Klas-

se *User* repräsentiert wird, werden vordefinierte Nachrichtendatentypen genutzt. Diese erben von einer mit dem «`UserMessage`»-Stereotyp annotierten Klasse und sind in Abbildung 7 abgebildet.

- *GetSingleSelection* lässt die Benutzeroberfläche eine Liste von Daten anzeigen, aus der der Benutzer ein Element auswählen kann. Die Liste wird als das Listenattribut *selection* vom generischen Typ *IFlowData* modelliert. Eine Antwortnachricht muss ein Attribut vom komplexen Datentyp besitzen, der dem Typ des konkreten Parameters von *GetSingleSelection* entspricht.

- *GetMultipleSelection* lässt die Benutzeroberfläche eine Listen von Daten anzeigen, aus der der Benutzer mehrere Elemente auswählen kann. Die Liste wird als das Listenattribut *multipleSelection* vom Typ *IFlowData* modelliert. Eine Antwortnachricht muss ein Attribut vom anwendungsspezifischen Listendatentyp besitzen, der dem Typ des konkreten Parameters von *GetMultipleSelection* entspricht.

- *GetInput<Type>* lässt die Benutzeroberfläche eine Eingabeaufforderung anzeigen. Die Eingabefelder entsprechen dabei den Attributen des Datentyps, der zwischen den spitzen Klammern angegeben wird. Eine Antwortnachricht muss ein Attribut vom anwendungsspezifischen Datentyp besitzen, der dem Typ zwischen den spitzen Klammern entspricht.

- *Confirm* lässt die Benutzeroberfläche eine textuelle Meldung anzeigen, die vom Benutzer bestätigt werden muss.

- *ConfirmRelease* lässt die Benutzeroberfläche ein Dialog zum Bestätigen einer Deklassifikation anzeigen. Das Klassenattribut *what* vom generischen Typ *IFlowData* enthält die zu deklassifizierende Information, während das Attribut *receiver* vom primitiven Typ *String* die Beschreibung des Empfängers beinhaltet.

- *Show* lässt die Benutzeroberfläche Informationen anzeigen, die im Klassenattribut *data* vom generischen Typ *IFlowData* enthalten ist.

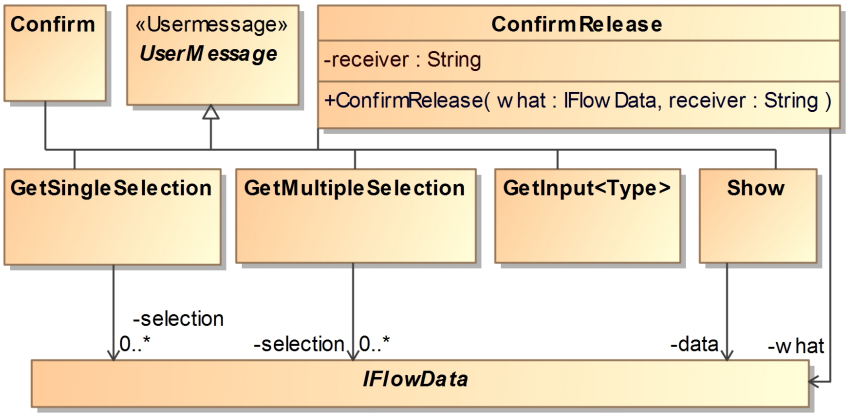


Abbildung 7: Vordefinierte Benutzernachrichten

10 MODELLIERUNG DER STATISCHEN SICHT

In diesem Abschnitt wird erläutert, wie der statische Teil einer Anwendung mit MODELFLOW modelliert wird. Die statische Sicht erfasst dabei Anwendungskomponenten und -datentypen, die für die Komponentenkommunikation notwendige Nachrichtenklassen, sowie Anwendungsmodule wie etwa manuelle Methoden, externe Anwendungen, und die Benutzerschnittstelle. Damit wird die Struktur und Aufbau der Anwendung definiert, wobei hierfür ausschließlich UML-Klassendiagramme verwendet werden. Alle solche Diagramme und die dazugehörigen Klassen werden in einem UML-Paket mit dem «`ClassDiagram`»-Stereotyp abgelegt. Im Folgenden werden die MODELFLOW-Modellierungsrichtlinien vorgestellt, an die sich der Entwickler halten muss, um den statischen Teil seiner Anwendung zu spezifizieren. Diese Richtlinien sind notwendig, um aus dem Anwendungsmodell analysierbaren und lauffähigen Code sowie ein gültiges formales Modell generieren zu können.

Abschnitt 10.1 erklärt die Modellierung von komplexen Datentypen und stellt die vordefinierten Datentypen vor, Abschnitt 10.2 zeigt die Spezifikationsrichtlinien für Anwendungskomponenten, während Abschnitt 10.1 beschreibt, wie Klassenoperationen definiert werden.

10.1 KOMPLEXE DATENTYPEN

Zusätzlich zu den primitiven Datentypen *Boolean*, *Integer*, *Double*, und *String* (siehe Unterabschnitt 9.2.3), sowie den vordefinierten komplexen Datentypen wie *GPSData* und *Encryptor* (siehe Unterabschnitt 9.2.4), kann der Modellierer eigene, anwendungsspezifische komplexe Datentypen definieren. Solche Datentypen werden als UML-Klassen modelliert, und können eine beliebige Anzahl von Attributen und Operationen enthalten. Diese Klassen erben implizit von der abstrakten, vordefinierten *IFlowData*-Klasse (siehe Unterabschnitt 9.2.4). Der Klassenname muss dabei eindeutig sein.

ATTRIBUTE UND ASSOZIATIONEN Ein komplexes Datentyp kann in MODELFLOW beliebig viele Attribute haben. Attribute vom einem primitiven Datentyp werden als UML-Attribute modelliert, während solche vom komplexen Typ sowohl als UML-Attribute (kompakte Darstellung) als auch als unidirektionale UML-Assoziationen modelliert werden können. Zyklische Datentypabhängigkeiten sind nicht erlaubt.

Solche Attribute haben einen Namen, einen Typ, sowie eine Multiplizität. Der Name entspricht hierbei entweder dem Namen des UML-Attributs, oder dem Namen des navigierbaren Endes der UML-Assoziation, während der Typ als Datentyp des UML-Attributs bzw. als UML-Klasse am Ende der UML-Assoziation modelliert wird, zu der hin-navigiert wird. Der Name muss innerhalb eines Datentyps eindeutig sein. Die Multiplizität bestimmt, ob es sich um eine Liste bzw. Map handelt, und wird als die Multiplizität des UML-Attributs oder des navigierbaren UML-Assoziationsendes festgelegt. Ist diese als `*`, d.h. unbeschränkt modelliert, so handelt es sich bei diesem Attribut um eine Liste bzw. Map; anderenfalls ist die Multiplizität nicht spezifiziert oder auf `1` beschränkt. Ist die Multiplizität unbeschränkt, der Datentyp des Attributs komplex, und hat eines der Attribute dieses Datentyps den Stereotyp *«key»*, so handelt es sich um eine Map, wobei der *«key»*-Attribut den Schlüssel dieser Map darstellt (siehe Tabelle 5).

OPERATIONEN Ein komplexer Datentyp kann eine oder mehrere Operationen enthalten, die von den Anwendungskomponenten aufgerufen werden können. Sie werden als UML-Operationen modelliert, die beliebig viele Eingabeparameter und maximal einen Rückgabeparameter haben dürfen. Die Parameter werden als UML-Parameter modelliert und haben entsprechend jeweils einen Namen, eine Richtung (`in` bzw. `return`), und einen (komplexen oder primitiven, vordefinierten oder anwendungsspezifischen) Typ. (Rückgabe-)Parameter können zusätzlich eine unbeschränkte Multiplizität `*` besitzen, was einem Listendatentypen entspricht.

MODELFLOW unterstützt Operationen, die von drei verschiedenen Typen von Methoden definiert werden:

- *Konstruktoren*, die die Reihenfolge der UML-Assoziationen vorgeben
- MEL*-Methoden*, deren Spezifikation vollständig mit MODELFLOW modelliert wird
- *Manuelle Methoden*, die händisch implementiert werden müssen

KONSTRUKTOREN Im dynamischen Teil des Anwendungsmodells können komplexe Datentypen durch den Aufruf ihres Konstruktors instantiiert werden. Der Name des Konstruktors entspricht dem Namen des Datentyps. Jeder komplexe Datentyp besitzt einen impliziten Standardkonstruktor, der keine Parameter entgegennimmt, und eine neue Instanz von diesem Datentyp zurückgibt. Dabei erhalten alle Attribute des neuen Objekts mit einem primitiven Datentyp einen Standardwert, der Tabelle 6 entnommen werden kann.

Hat der Datentyp höchstens eine UML-Assoziation, so besitzt er zusätzlich einen Standardkonstruktor mit Parametern, deren Namen

und Typen den Attributen des Datentyps entspricht. Die Parameterreihenfolge entspricht dabei der Reihenfolge der modellierten UML-Attribute mit der anschließenden UML-Assoziation.

Hat der Datentyp mehr als eine Assoziation, so kann die Reihenfolge der UML-Assoziationen nicht dem Modell entnommen werden. Daher benötigt der Datentyp einen expliziten Konstruktor, der alle seine UML-Attribute und -Assoziationen als Parameter repräsentiert und ihre Reihenfolge festlegt.

Der implizite oder modellierte Konstruktor weist die übergebenen Parameter den Attributen seiner Klasse zu. Dieses Verhalten ist implizit und muss nicht modelliert werden.

MEL*-METHODEN Zusätzlich zu den Konstruktoren kann ein komplexer Datentyp Operationen enthalten, die vollständig auf Modellebene spezifiziert werden. Die Spezifikationssprache solcher Methoden ist MEL* (siehe Abschnitt 11.1 für Details).

MEL*-Methoden können verwendet werden, um beliebige Funktionalität zu implementieren, die nicht plattformspezifisch ist, und mit MEL* ausgedrückt werden kann. Dazu gehört z.B. das Filtern von Daten, oder das Anonymisieren von privaten Informationen, die als Parameter an die Methode übergeben werden. Die Methode darf lediglich auf ihre Ein- und Ausgabeparameter zugreifen. Dabei handelt es sich oft um *Deklassifikation* von Information, d.h., die Rückgabe der Operation wird als weniger geheim eingestuft als ihre Parameter. Solche Deklassifikationsmethoden müssen mit dem «`declassify`»-Stereotyp versehen werden; ihre Sicherheit kann formal bewiesen werden.

MANUELLE METHODEN Der Modellierer kann zusätzlich Operationen deklarieren, deren Implementierung nicht Teil des Anwendungsmodells ist. Diese Implementierung wird als *manuelle Methode* bezeichnet, und die dazugehörige Operation erhält den Stereotyp «`manual`». Es handelt sich dabei um eines der unterstützten Anwendungsmodule, deren statische Modellierung in Unterabschnitt 10.3.1 ausführlich beschrieben wird.

NACHRICHTENDATENTYPEN Nachrichten, die zwischen Anwendungskomponenten ausgetauscht werden, haben in MODELFLOW einen eigenen Nachrichtendatentyp. Diese werden als UML-Klassen modelliert und können wie herkömmliche komplexe Datentypen Attribute besitzen (vgl. Abschnitt 10.1), dürfen jedoch mit Ausnahme von Konstruktoren keine weiteren Operationen definieren. Sie spezifizieren den Typ der Nachricht, während ihre Attribute den Nachrichteninhalt modellieren.

Zur Kommunikation mit der vordefinierten Benutzeroberfläche werden die Benutzernachrichtendatentypen verwendet, die in Unterab-

schnitt 9.2.4 aufgezählt und erläutert wurden. Zur Kommunikation der Anwendungskomponenten untereinander oder mit Anwendungsmodulen wie externen Apps und Services werden anwendungsspezifische Nachrichtendatentypen als Klassen modelliert, die von einer Oberklasse mit dem «`Message`»-Stereotyp erben (vgl. Unterabschnitt 9.2.2). Diese Oberklasse darf ihrerseits nicht selbst als Nachrichtendatentyp verwendet werden.

DATENTYPBESCHREIBUNG Um die Bedeutung eines Datentyps bzw. dessen Attribute für den Endnutzer zu dokumentieren, können die entsprechenden UML-Klassen, -Attribute und -Assoziationen mit einer Kurzbeschriftung versehen werden. Dies ist in dem Fall notwendig, wenn der Datentyp bei der Kommunikation mit der vordefinierten Benutzeroberfläche als Basis für eine Eingabemaske genutzt wird. Die Beschriftung des Datentyps wird dem Nutzer bei der Eingabeaufforderung angezeigt, um zu erklären, welche Informationen gerade abgefragt werden. Die Beschriftung der Attribute entspricht der Beschriftung der einzelnen Eingabefelder, die von diesen Attributen repräsentiert werden.

Zur Beschriftung wird der Stereotyp «`labeled`» genutzt, dessen Stereotyp-Tag *label* den Beschriftungstext enthalten soll. Wird der Stereotyp auf die UML-Klasse angewandt, die den Datentyp repräsentiert, so muss ihr Tag die Beschreibung der Information enthalten, die durch den Datentyp strukturiert wird. Wird der Stereotyp auf die UML-Assoziation bzw. -Attribut dieses Datentyps angewandt, so enthält ihr Tag die Beschreibung der Teilinformation, die dieses Attribut repräsentiert.

Da der Datentyp bei mehreren Eingabeaufforderungen verwendet werden kann, muss die Beschriftung der Klasse bzw. ihrer Attribute möglichst kontextunabhängig ausfallen. Die Beschriftung, die den konkreten Kontext der Eingabeaufforderung beschreibt, wird im Sequenzdiagramm vorgenommen (vgl. Unterabschnitt 11.2.3).

10.2 ANWENDUNGSKOMPONENTEN

Als Anwendungskomponente wird der logische Kern einer mobilen Anwendung bzw. eines Webservices bezeichnet, der vollständig mit MODELFLOW spezifiziert wird. Eine Komponente wird im statischen Teil des Modells als eine UML-Klasse repräsentiert. Handelt es sich um eine mobile Anwendung, so erhält die Klasse den «`Application`»-Stereotyp, während Klassen, die Webservices repräsentieren, mit dem Stereotyp «`Service`» annotiert werden müssen.

Für die Modellierung von Komponenten gelten größtenteils dieselben Richtlinien wie für komplexe Datentypen. Komponenten können Attribute haben, die ihren internen, persistenten Speicher repräsentieren. Solche Attribute werden als UML-Attribute (für Kom-

ponentenattribute mit primitiven oder komplexen Datentyp) oder -Assoziationen (für Komponentenattribute mit komplexen Datentyp) modelliert (vgl. Abschnitt 10.1).

Komponenten können Operationen deklarieren, die durch manuelle Methoden oder MEL*-Methoden implementiert werden (vgl. Abschnitt 10.1). Explizite Konstruktoren sind dagegen nicht notwendig, da Komponenten nicht mit MEL* instantiiert werden dürfen. Stattdessen erhalten die Komponentenattribute beim Deployen der Anwendung Standardwerte, die Tabelle 6 entnommen werden können.

10.3 ANWENDUNGSMODULE

Eine lauffähige Anwendung besteht selten lediglich aus dem logischen Kern einer mobilen App bzw. eines Webservices. Teile der finalen Anwendung, die nicht vollständig im Anwendungsmodell spezifiziert sind, werden in MODELFLOW als *Anwendungsmodule* bezeichnet. Dazu gehören u.a. manuelle Methoden, externe Webservices und Apps, sowie die graphische Oberfläche.

Oft liegt die Implementierung solcher Module zum Zeitpunkt der Modellierung noch nicht vor. Da diese jedoch für die Sicherheit der Gesamtanwendung wichtig sein kann, müssen auf Modellebene Einschränkungen bezüglich der Implementierung modelliert werden, an die sie sich halten muss. Je nach Modultyp kann die finale Implementierung bezüglich dieser Einschränkungen analysiert werden.

Im Folgenden wird auf die Modellierung solcher Module im Detail eingegangen.

10.3.1 *Manuelle Methoden*

Bei MODELFLOW-Anwendungen kann es sich um größere Systeme handeln, die mit der Plattform, auf der sie ausgeführt werden, interagieren können. Der Ansatz erlaubt es dem Entwickler, einzelne Methoden, die im Anwendungsmodell deklariert, aber nicht spezifiziert sind, händisch zu programmieren. Diese werden in MODELFLOW *manuelle Methoden* genannt, und können bei der Informationsflussanalyse unabhängig von dem Codeskelett der Anwendung betrachtet werden (siehe Unterabschnitt 16.2.3). Eine manuelle Methode stellt somit ein *Anwendungsmodul* dar, mit dem das Codeskelett über klar definierte Schnittstellen (Ein- und Ausgabeparameter der Methode) interagiert. Zusätzlich zu komplexen Algorithmen, deren Sicherheit bezüglich Informationsdeklassifikation nicht verifiziert werden muss, kann in manuellen Methoden beispielsweise das Auslesen der aktuellen Standortposition des Nutzergeräts, das Schreiben und Lesen des Dateisystems, oder Versenden von SMS implementiert werden.

Obwohl MODELFLOW manche dieser Funktionalitäten durch vordefinierte Methoden unterstützt (vgl. Unterabschnitt 9.2.4), und die

Modellierungssprache durch weitere solche Methoden erweitert werden kann, kann es dennoch sinnvoll sein, Teile der Anwendung händisch zu implementieren. Zum einen sollte das Codeskelett, der auf Informationsflussverletzungen automatisch gecheckt wird, möglichst minimal gehalten werden. Dies ist wichtig, da das eingesetzte Informationsflussanalysetool nur beschränkt große Anwendungen checken kann ([32]). Zudem sinkt dadurch die Wahrscheinlichkeit für Fehlalarme bei der Informationsflussanalyse.

Operationen, die durch manuelle Methoden implementiert werden sollen, müssen mit dem «`uses`»-Stereotyp versehen werden (vgl. Unterunterabschnitt 9.2.2.4). Die Stereotyp-Tags *reads* und *writes* bezeichnen dabei die Plattform-spezifischen Quellen und Senken, auf die die Methode zugreifen darf. Abgesehen von solchen Quellen und Senken darf die Implementierung nur auf ihre Ein- und Ausgabeparameter zugreifen. Für nähere Details zu den Informationsflussanotationen siehe Kapitel 12. Handelt es sich zudem um eine Deklassifikationsmethode, so muss die Operation zusätzlich mit dem Stereotyp «`declassify`» annotiert werden.

10.3.2 *Vordefinierte Operationen*

MODELFLOW stellt eine Reihe von vordefinierten Operationen und Datentypen zur Verfügung. Dabei handelt es sich um plattformspezifische (z.B. zum Auslesen des GPS-Sensors) sowie kryptographische Operationen (z.B. zum Verschlüsseln von Daten): eine detaillierte Auflistung findet sich in Unterabschnitt 9.2.4. Die Operationen können im dynamischen Teil des Anwendungsmodells vom Modellierer aufgerufen werden.

10.3.3 *Vordefinierte graphische Benutzeroberfläche*

Um Informationen auf dem Bildschirm des Nutzergeräts anzuzeigen, oder Nutzerdaten abzufragen, kann der Anwendungsentwickler auf eine vordefinierte Benutzeroberfläche zurückgreifen, die von MODELFLOW zur Verfügung gestellt wird. Dabei profitiert der Benutzer von zusätzlichen Sicherheitsmechanismen, die sicherstellen, dass die garantierten Sicherheitseigenschaften sich tatsächlich auf die Eingabe beziehen, die der Benutzer vornimmt (siehe Unterunterabschnitt 16.2.4.5 für Details).

Diese Benutzeroberfläche wird als die UML-Klasse *User* mit dem Stereotyp «`User`» repräsentiert. Die Klasse hat eine Reihe von Attributen, die die Benutzereingabe oder -entscheidung repräsentieren, die im dynamischen Teil des Anwendungsmodells abgefragt werden kann.

Nachrichten an diese Benutzeroberfläche sind ebenfalls vordefiniert (vgl. Unterabschnitt 9.2.4), während die Antwortnachrichten an-

wendungsspezifisch sind und eindeutig sein müssen. Die letzteren werden wie alle anderen anwendungsspezifischen Nachrichten modelliert, und müssen von einer Klasse mit dem Stereotyp «`Message`» erben.

Der Modellierer kann die *User*-Klasse in das Klassendiagramm seiner Anwendung integrieren, um die Nutzung der vordefinierten Benutzeroberfläche im statischen Teil des Anwendungsmodells explizit zu machen.

10.3.4 *Manuelle grafische Benutzeroberfläche*

Ein wichtiger Bestandteil der Apps für mobile Geräte wie Smartphones oder Tablets ist ihre grafische Oberfläche. Aufgrund der kleinen Bildschirmgröße, der Touch-Bedienung, und der Nutzung solcher Apps unterwegs, muss diese besonders intuitiv sein und auf die Anwendung zugeschnitten werden. Eine ansprechende und einfach bedienbare grafische Oberfläche von sicherheitskritischen Anwendungen trägt zudem zur Nutzersicherheit bei [19]. Sie erhöht die Akzeptanz der Anwendung, und gibt dem Nutzer keinen Anlass, auf eine einfachere, aber unsicherere Alternative auszuweichen. Außerdem macht der Nutzer einer solchen Anwendung weniger Fehler bei der Bedienung, die die Sicherheit der Anwendung aushöhlen [100, 101].

Das Designen von Benutzeroberflächen mit UML ist prinzipiell möglich [42], der Schwerpunkt von MODELFLOW liegt jedoch auf der Entwicklung von informationsflusssicherer Anwendungslogik. Daher wird es dem Entwickler in MODELFLOW ermöglicht, zusätzlich zur vordefinierten Benutzeroberfläche (vgl. Unterabschnitt 10.3.3) eine händisch implementierte grafische Oberfläche zu integrieren. Diese wird als eine anwendungsspezifische Klasse mit dem «`GUI`»-Stereotyp im Anwendungsmodell repräsentiert. Sie kann eine Reihe von Attributen besitzen, die die Benutzerentscheidungen oder -eingaben repräsentieren und als UML-Attribute bzw. -Assoziationen modelliert werden. Hierbei gelten dieselben Modellierungsrichtlinien wie bei komplexen Datentypen (vgl. Abschnitt 10.1). Die «`GUI`»-Klasse deklariert keine Operationen, da ihr Verhalten von Hand implementiert wird.

Nachrichten von und an die manuelle grafische Oberfläche werden als Nachrichtendatentypen modelliert, die von der «`Message`»-Klasse erbt (siehe Abschnitt 10.1).

10.3.5 *Externe Apps und Webservices*

Oftmals möchte man als Entwickler einer Anwendung auf bereits vorhandene Apps und Services zugreifen, um z.B. Implementierungsaufwand zu sparen, oder andere Anwendungen zu integrieren. Da

der Code einer solchen externen Komponente oft nicht vorliegt, müssen über ihre konkrete Implementierung Annahmen getroffen werden, um Informationsflussanalyse der finalen Anwendung zu ermöglichen.

Die Integration externer Komponenten wird von MODELFLOW ermöglicht, indem solche externen Apps und Services im Klassendiagram als UML-Klassen repräsentiert werden. Handelt es sich dabei um einen externen Webservice, so erhält die entsprechende Klasse den *«ExternalWebService»*-Stereotyp. Soll eine externe App integriert werden, so erhält die Klasse den Stereotyp *«ExternalApp»*. Solche Klassen können eine Reihe von Attributen enthalten, die den internen Speicher dieser Komponenten repräsentieren. Sie werden nur bei der Modellierung der Kommunikation benötigt, und müssen nicht der tatsächlichen Implementierung der externen Komponenten entsprechen. Für die Modellierung der Attribute gelten dieselben Richtlinien wie bei komplexen Datentypen (Abschnitt 10.1). Externe Komponenten dürfen keine Operationen deklarieren.

In MODELFLOW muss die Inter-App-Kommunikation mit externen Apps nicht zwingend bidirektional sein. Android unterstützt beispielsweise Inter-App-Kommunikation, bei der die Antwortnachricht optional ist. Ist dies der Fall, so erhält die Klasse mit dem *«ExternalApplication»*-Stereotyp den zusätzlichen Stereotyp-Tag *optionalResult*. Ist man sich als Entwickler sicher, dass die externe App keine Antwortnachricht liefert, so erhält die dazugehörige Klasse den Stereotyp-Tag *noResult*. Welche Kommunikationsart die externe App unterstützt, muss ihrer Dokumentation entnommen werden. Wird dies auf Modellebene falsch spezifiziert, so wird die Kommunikation mit dieser App in der finalen Anwendung fehlschlagen.

10.4 ZUGRIFFSKONTROLLE

Vor allem bei Webservices ist es oft notwendig, den Zugriff auf manche Datenspeicherbereiche nur bestimmten (z.B. registrierten) Benutzern zu erlauben. Ein solcher Speicherbereich wird in MODELFLOW durch den Eintrag einer Map abgebildet, wobei der dazugehörige Schlüssel ein Geheimnis darstellen muss; folglich darf nur ein Benutzer, dem das Geheimnis bekannt ist, auf den entsprechenden Map-Eintrag zugreifen. Dadurch können beispielsweise benutzerspezifische Daten in einem solchen Map-Eintrag auf einem Webservice abgelegt werden, die für andere Benutzer nicht zugänglich sind, solange sie das dazugehörige geheime Map-Schlüssel nicht kennen. Ein solcher Schlüssel stellt also die Authentifikationsdaten des Nutzers dar, und kann z.B. aus seinem Benutzernamen und Passwort bestehen.

Die Geheimhaltung der Authentifikationsdaten kann in MODELFLOW *als eine Informationsflusseigenschaft formalisiert werden (siehe Abschnitt 12.2)*

Hierfür wird der Datentyp des Map-Schlüssels (der als ein Attribut mit dem Stereotyp *«key»* modelliert ist, siehe Abschnitt 10.1) mit dem Stereotyp *«AuthData»* markiert. Es muss sich dabei um einen

komplexen Datentyp handeln, der Authentifikationsdaten eines Nutzers abbildet. Für den Zugriff auf eine solche Map gelten eine Reihe von Modellierungsregeln, um unerwünschte Informationsflüsse zu vermeiden, die in Unterabschnitt 11.2.6 aufgezählt werden.

Abbildung 8 zeigt ein Beispiel für zugriffsbeschränkte Benutzerdaten, die ein Bank-Webservice verwaltet. Dabei handelt es sich um den Kontostand des Benutzers (*balance*), der in der Map *accounts* der *Bank*-Komponente gespeichert wird. Um den Kontostand eines bestimmten Nutzers aus der Map auszulesen oder zu aktualisieren, muss beim Zugriff auf die Map mittels der vordefinierten Methoden (siehe Unterabschnitt 9.2.4) der korrekte Schlüssel in Form seines Benutzernamens und Passworts angegeben werden, die als Attribute des *AuthData*-Datentyps modelliert sind.

Für die vollständige Beschreibung der Anwendung siehe Kapitel 20

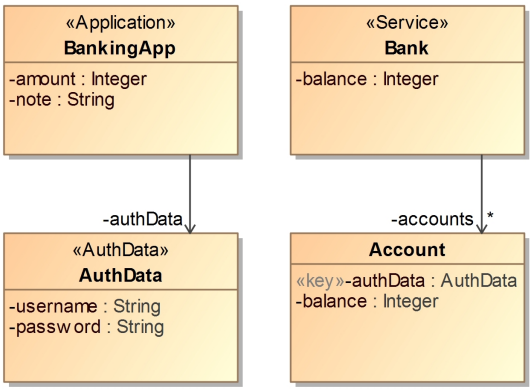


Abbildung 8: Zugriffsbeschränkte Nutzerdaten der Banking-Fallstudie

10.5 STATISCHE SICHT AUF DIE TRAVEL PLANNER-FALLSTUDIE

Die vorgestellten Modellierungsrichtlinien sollen nun anhand der statischen Sicht auf das Modell der Travel Planner-Fallstudie illustriert werden. Es handelt sich hierbei um eine verteilte Anwendung bestehend aus den Apps zum Buchen von Flügen (*TravelPlanner*, TP) und Speichern von Kreditkartendaten (*CreditCardCenter*, CCC), sowie den Webservices einer Reiseagentur (*TravelAgency*, TA) sowie einer Fluglinie (*Airline*, A).

Travel Planner ist eine kompakte und übersichtliche Fallstudie, die die wichtigsten Bestandteile von MODELFLOW demonstriert.

In Abbildung 9 ist das Klassendiagramm der Travel Planner-Anwendung abgebildet, das ihre Anwendungskomponenten, -module, sowie anwendungsspezifische komplexe Datentypen spezifiziert. Abbildung 10 zeigt ein weiteres Klassendiagramm, das die Nachrichtendatentypen dieser Anwendung enthält. Die Diagramme sowie die darin vorkommenden UML-Klassen sind im UML-Package mit dem Stereotyp «`ClassDiagram`» abgelegt.

KOMPLEXE DATENTYPEN Das Anwendungsmodell definiert drei komplexe Datentypen:

- *CreditCardDetails* (vgl. Abbildung 9) repräsentiert die Kreditkarteninformationen des Nutzers, und besitzt drei Attribute vom primitiven Typ *String*: *name* speichert den Namen des Kreditkartenbesitzers, *number* die Kreditkartennummer, und *expiration* das Ablaufdatum der Kreditkarte.
- *FlightOffer* (vgl. Abbildung 9) repräsentiert das Flugangebot von der Fluglinie, und hat zwei Attribute: *id* vom Typ *Integer*, das die Identifikationsnummer des Angebots repräsentiert, und *airline* vom Typ *String*, das die Bezeichnung der Fluglinie enthält
- *RequestData* (vgl. Abbildung 10) bildet die Anfragedaten ab, die zum Abfragen von Flugangeboten verwendet wird. Das Attribut *dat : String* dieser Nachricht speichert dabei das Datum, an dem der Flug stattfinden soll. Diese semantische Bedeutung der Klasse und ihres Attributs ist im Tag *label* des Stereotyps «`labeled`» enthalten, mit dem sie annotiert sind.

ANWENDUNGSKOMPONENTEN Die vier Anwendungskomponenten sind als Klassen *TravelPlanner* und *CreditCardCenter* jeweils mit dem Stereotyp «`Application`», sowie *TravelAgency* und *Airline* jeweils mit dem Stereotyp «`Service`» abgebildet.

KOMPONENTENATTRIBUTE Die Komponente *TravelPlanner* speichert das Flugangebot, das sie von *TravelAgency* erhält, in ihrem Attribut *flightOffer*, das als UML-Assoziation zum komplexen Datentyp *FlightOffer* modelliert wird. *Airline*-Webservice verwaltet seinerseits eine Liste von solchen Flugangeboten, was durch das Attribut *flightOffers* ebenfalls vom Typ *FlightOffer* mit der Multiplizität `*` repräsentiert wird. *CreditCardCenter* speichert die Kreditkarteninformationen des Nutzers im Attribut *ccc* vom Typ *CreditCardData*.

KOMPONENTENOPERATIONEN Das Modell der Travel Planner-Fallstudie definiert vier Operationen zum Verarbeiten von Flugangeboten und Kreditkartendaten.

Die Operation *declassifyCCD(ccd : CreditCardData) : CreditCardData* der *CreditCardCenter*-Komponente nimmt Daten vom Typ *CreditCardData* als Eingabeparameter entgegen und gibt dieselben Daten als Rückgabeparameter zurück. Sie wird zur expliziten Deklassifikation von Kreditkartendaten des Nutzers benutzt, bevor sie an die *Airline*-Komponente geschickt werden. Daher erhält die Deklaration den «`declassify`»-Stereotyp.

Die Operationen *filterOffers(req : RequestData, fos : FlightOffer[*]) : FlightOffer[*]* und *processBooking(id : Integer, ccd : CreditCardDetails)* der Komponente *Airline* müssen händisch implementiert werden,

und erhalten daher den Stereotyp «manual». *filterOffers* wird zum Filtern von Flugangeboten *fos* gemäß der Anfrage *req* verwendet, und gibt eine Liste von Flugangeboten zurück, die der Anfrage entsprechen. *processBooking* bildet die Abwickelung des Bezahlvorgangs mit den Kreditkartendaten *ccd* für das Flugangebot mit der Identifikationsnummer *id* ab.

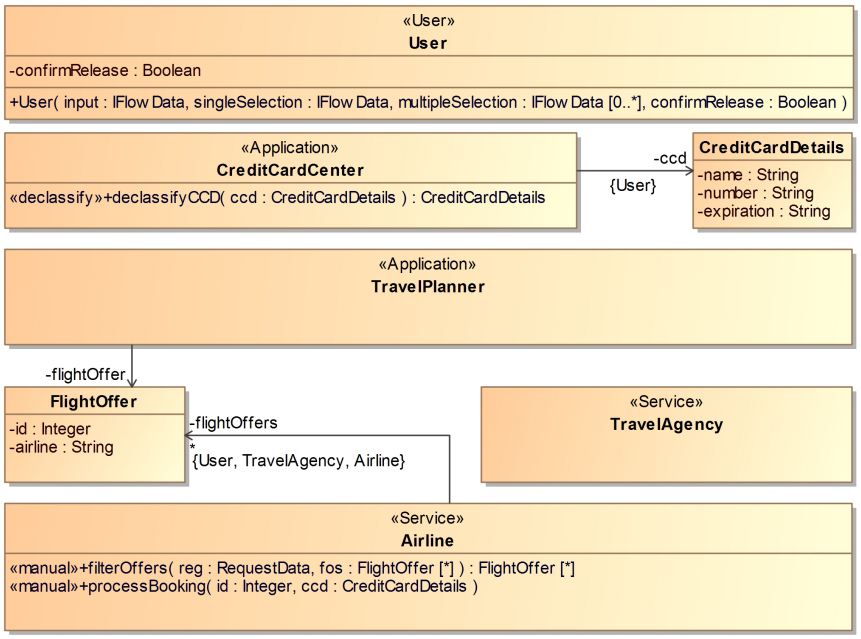


Abbildung 9: Klassendiagramm der Travel Planner-Anwendung

NACHRICHTENDATENTYPEN Das Anwendungsmodell definiert 14 Nachrichtendatentypen, die bei der Kommunikation zwischen den modellierten Komponenten verwendet werden (siehe Abbildung 10).

Die Nachrichten *SelectedFlightOffers*, *BookSelected*, *GetDeclassifiedCCD*, und *RetConfirm* entsprechen dabei den Antwortnachrichten der vordefinierten Benutzeroberfläche auf Standardbenutzernachrichten zur Eingabe von Anfragedaten (*GetInput<RequestData>*), Auswahl eines Flugangebots (*GetSingleSelection*), Bestätigung der Deklassifikation (*ConfirmRelease*), sowie Bestätigung der Buchung (*Confirm*) (vgl. Abbildung 7 für alle in MODELFLOW unterstützten Standardbenutzernachrichten).

Die Nachrichten *RequestFlightOffer* und *GetFlightOffers* werden zur Abfrage von Flugangeboten verwendet, die in den Nachrichten *RetFlightOffers* und *RequestedFlightOffers* zurückgegeben werden.

Die Nachricht *ReleaseCCD* veranlasst die Komponente *CreditCardCenter* dazu, die Kreditkartendaten des Nutzers zu deklassifizieren, die mit der Nachricht *DeklassifiedCCD* an den *TravelPlanner* zurückgegeben werden.

Die Nachrichten *BookFlightOffers*, *PayCommission*, *OkPayCommission*, und *OkConfirmBooking* werden beim Bezahlvorgang zwischen den Komponenten *TravelPlanner*, *Airline*, und *TravelAgency* ausgetauscht.

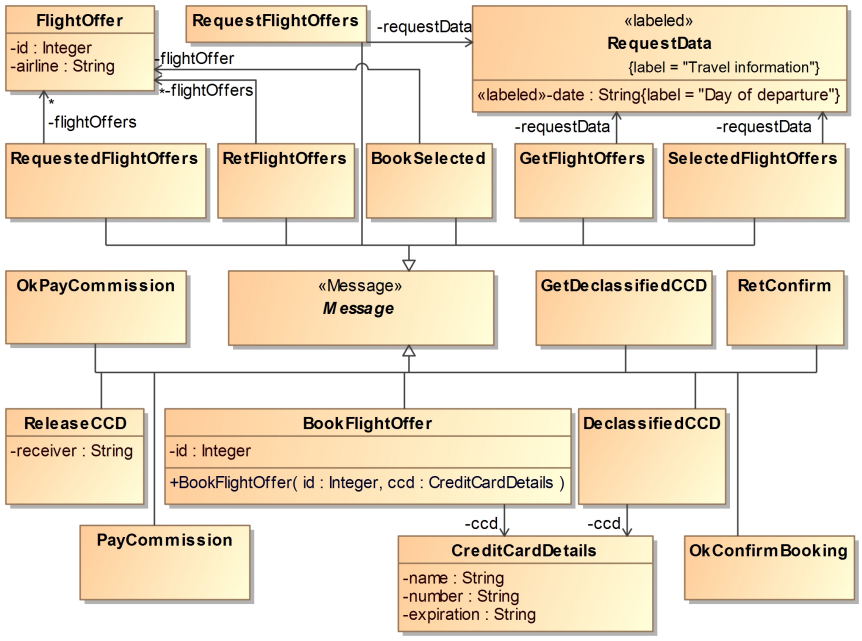


Abbildung 10: Nachrichtendatentypen der Travel Planner-Anwendung

11 MODELLIERUNG DER DYNAMISCHEN SICHT

In diesem Abschnitt wird erläutert, wie der dynamische Teil einer Anwendung mit MODELFLOW modelliert wird. Die dynamische Sicht beschreibt das Verhalten der Anwendung, und erfasst dabei Sequenzdiagramme, die den Programmablauf und Komponentenkommunikation definieren, sowie MEL*-Methoden, die als Aktivitätsdiagramme modelliert werden, und Filter- bzw. Deklassifikationsfunktionen spezifizieren. Die Spezifikation erfolgt einem Dialekt der Model Extension Language [65]. Die resultierenden Diagramme werden im UML-Paket mit dem «`BehaviorDiagram`»-Stereotyp abgelegt.

Im Folgenden werden die MODELFLOW-Modellierungsrichtlinien vorgestellt, an die sich der Entwickler halten muss, um den dynamischen Teil seiner Anwendung spezifizieren zu können. Abschnitt 11.1 beschreibt die MEL*-Sprache für MODELFLOW, Abschnitt 11.2 beschreibt die Modellierung des Anwendungsverhaltens bezüglich der Komponentenkommunikation, während Abschnitt 11.3 die Spezifikation von Filter- bzw. Deklassifikationmethoden erläutert.

11.1 MEL* ALS DIALEKT DER MODEL EXTENSION LANGUAGE

MODELFLOW erlaubt es, das Anwendungsverhalten in einem hohen Detailgrad auf Modellebene zu spezifizieren. Dadurch ist es möglich, den Funktionsumfang der gesamten Anwendung nahezu vollständig im Anwendungsmodell abzubilden. Lediglich wenn bestimmte plattformspezifische Funktionalität gefragt ist, wird der Modellierer veranlasst, die Anwendung durch händisch implementierten Code zu erweitern (vgl. Unterabschnitt 10.3.1 und Unterabschnitt 10.3.4 für Beschreibung der manuellen Anwendungsmodule).

Das Anwendungsverhalten wird in MODELFLOW mit UML-Sequenz- und Aktivitätsdiagrammen, sowie einer integrierten, textuellen Modellierungssprache MEL* festgelegt. MEL* ist somit ein Teilaspekt von MODELFLOW, wobei ihre abstrakte Syntax Teil des MODELFLOW-Metamodells ist.

MODEL EXTENSION LANGUAGE (MEL) Die Grundlage für MEL* ist die Model Extension Language (MEL), die an die Bedürfnisse von MODELFLOW angepasst und maßgeblich erweitert wurde. Bei MEL handelt sich um eine intuitive, objektorientierte Sprache, die UML-Diagramme ergänzt. Ihre Standardbibliothek von Datentypen und Operationen ist auf die Domäne von kryptographischen Protokollen ausgelegt. Die Syntax von MEL ist an Java angelehnt, und eignet sich

besonders zur Spezifikation der Nachrichtenverarbeitung in verteilten Systemen.

GEGENÜBERSTELLUNG VON MEL UND MEL* MEL* ist ein Dialekt der Model Extension Language. Die Sprache ist wertbasiert; beim Vergleich von zwei Objekten werden somit die Werte ihrer Attribute verglichen. Wird ein Objekt an ein anderes zugewiesen, oder als Methodenparameter übergeben, so wird dabei eine Kopie des Objekts erstellt. Dies spielt bei der Informationsflussanalyse des Codes eine wichtige Rolle, da hiermit Seiteneffekte vermieden werden und manuelle Methoden separat vom Codeskelett gecheckt werden können. Zudem ist MEL* Null-frei, d.h., alle Attribute und Variablen erhalten einen Standardwert und Operationen geben im Fehlerfall einen Standardwert zurück. Dies erleichtert ebenfalls die Informationsflussanalyse, da dadurch keine Informationsflüsse über Exception-Handler berücksichtigt werden müssen.

Im Gegensatz zu MEL wird in MEL* die Kommunikation zwischen Komponenten nicht mit Aktivitätsdiagrammen, sondern Sequenzdiagrammen beschrieben. Dies hat den Grund, dass die Anwendungen in MODELFLOW umfangreicher ausfallen können als reine kryptographische Protokolle, für deren Modellierung MEL entworfen wurde. MODELFLOW-Anwendungen werden daher modularisiert betrachtet, wobei bei Modellierung der Komponenteninteraktion von der Funktionalität der Module (wie etwa Methoden, Benutzerschnittstellen, oder externe Komponenten) abstrahiert werden kann. Als Folge eignen sich Sequenzdiagramme als knappe und übersichtliche Alternative zu Aktivitätsdiagrammen, da bei Sequenzdiagrammen die *Interaktion* zwischen Komponenten im Vordergrund steht.

Jedoch kommen auch bei MODELFLOW die Aktivitätsdiagramme zum Tragen. Sie werden verwendet, um Deklassifikations- bzw. Filtermethoden zu spezifizieren, die ihrerseits aus den Sequenzdiagrammen heraus aufgerufen werden können. Diese Spezifikation geschieht mit MEL*, wobei hier die konkrete Syntax von MEL (mit der Erweiterung um Schleifen) verwendet wird.

Die abstrakte Syntax und Semantik von MEL* ist bis auf die hinzugekommene Unterstützung für Schleifen ähnlich zu MEL, und wird in Unterabschnitt 11.1.1 zusammengefasst. Die konkrete Syntax von MEL wurde in MEL* um Unterstützung von Sequenzdiagrammen erweitert, und wird in Unterabschnitt 11.1.2 beschrieben. Zudem hat MEL* eine eigene Standardbibliothek von Datentypen und Operationen, die auf die Anwendungsdomäne von MODELFLOW angepasst wurde. Diese wird in Unterabschnitt 11.1.3 erläutert.

11.1.1 *Abstrakte Syntax und Semantik: eine Zusammenfassung*

Die abstrakte Syntax und Semantik von MEL* sind größtenteils der Syntax und Semantik von MEL nachempfunden, die in [65] beschrieben sind. Die MEL*-Syntax ist als Teil des MODELFLOW-Metamodells definiert, das als Instanz des ECore-Metamodells abgebildet ist. .

ECore ist das Metamodell, das vom Eclipse Modeling Framework spezifiziert wird

Im Folgenden werden die Elemente der abstrakten Syntax von MEL* als *EClass*-Instanzen knapp zusammengefasst, wobei auf die Attribute der in MEL identischen Elemente nicht näher eingegangen wird. In MEL* neu hinzugekommene oder angepasste Elemente werden unterstrichen und samt ihrer Attribute im Detail beschrieben.

- **Typen** als Unterklassen von *MELType*: *IdentMELType* (komplexer Typ), *PrimitiveMELType* (primitiver Typ), *VoidMELType* (Typ für Ausdrücke, die keinen Rückgabewert haben), *MELListType* (Listentyp)

- **Literale** als Unterklassen von *Literal*: *BooleanLiteral* (Boolsches Literal), *StringLiteral* (String-Literal),
  NumberLiteral(value : String, floatingPoint : EBoolean) (Zahl-Literal). Im Gegensatz zu MEL unterstützt MEL* Gleitzahlen, weshalb der Wert (*value*) des Zahl-Literals als *String* gespeichert wird. Handelt es sich um eine ganze Zahl, so hat *floatingPoint* den Wert *true*, anderenfalls ist er *false*.

- **Ausdrücke** als Unterklassen von *MEL*: *ParenthesizedExpr* (geklammerter Ausdruck), *UnaryExpr* (unärer Ausdruck), *BinaryExpr* (Binärer Ausdruck), *LiteralExpr* (Literalausdruck), *LocVarAccess* (Zugriff auf lokale Variable), *FieldAccess* (Zugriff auf Attribut), *SFieldAccess* (Zugriff auf statisches Attribut), *Assignment* (Zuweisung), *CreateExpr* (Objekterzeugung), *MethodCall* (Methodenaufruf), *StatementList* (Liste von Anweisungen), *Receive* (Nachrichtenempfang), *Send* (Nachrichtenversand).
  MELIf(test : MEL, thenPart : MEL, elsePart : MEL) bildet eine bedingte *If*-Verzweigung ab. Der Testausdruck ist durch das *test*-Attribut repräsentiert, während die Attribute *thenPart* und *elsePart* die Ausdrücke repräsentieren, die ausgeführt werden, wenn der Testausdruck den Wert *true* bzw. *false* hat.
  MELWhile(test : MEL, content : MEL) bildet eine *While*-Schleife ab. Das *test*-Attribut speichert den Testausdruck der Schleife. Der Ausdruck, der von der Schleife wiederholt ausgeführt wird, ist durch das Attribut *content* repräsentiert.
  MELFor(init : MEL, test : MEL, update : MEL, body : MEL) repräsentiert eine *For*-Schleife. Das Attribut *init* bildet die Initialisierung des Schleifenzählers ab, *test* ist der Testausdruck der Schleife, *update* repräsentiert die Aktualisierung des Schleifenzählers, während *body* den Rumpf der Schleife enthält.

11.1.2 *Konkrete Syntax*

MEL* kann sowohl innerhalb von Sequenz- als auch Aktivitätsdiagrammen verwendet werden. Aktivitätsdiagramme werden zur Spezifikation von MEL*-Methoden genutzt, die z.B. Deklassifikation- bzw. Filterfunktionen implementieren. Sequenzdiagramme werden hingegen zur Spezifikation der Komponenteninteraktion verwendet, die ihrerseits MEL*-Methoden aufrufen können.

Die konkrete Syntax von MEL* entspricht einer erweiterten Teilmenge der Regeln der Model Extension Language. MEL* wurde um Schleifen, sowie Sequenzdiagrammelemente erweitert. Ihre Regeln sind im Anhang in Abbildung 59 abgebildet (siehe Anhang); die in MEL* neu hinzugekommene Regeln sind unterstrichen. Bei *ReplyMessage*, *CallMessage*, sowie *ToSelfMessage* handelt es sich dabei um den MEL*-Code, der als Name einer Hin- oder Rücknachricht bzw einer Nachricht „an sich selbst“ im Sequenzdiagramm abgebildet werden kann. Bei *ForExpr* handelt es sich um eine *For*-Schleife, die als Ausdruck in einer UML-Action verwendet werden kann. Ihre Argumente entsprechen dabei dem Ausdruck zur Initialisierung des Schleifenzählers, dem Testausdruck der Schleife, sowie dem Ausdruck zur Aktualisierung des Schleifenzählers. Die vollständige Beschreibung der weiteren Regeln findet sich in [65]

11.1.3 *Standardbibliothek*

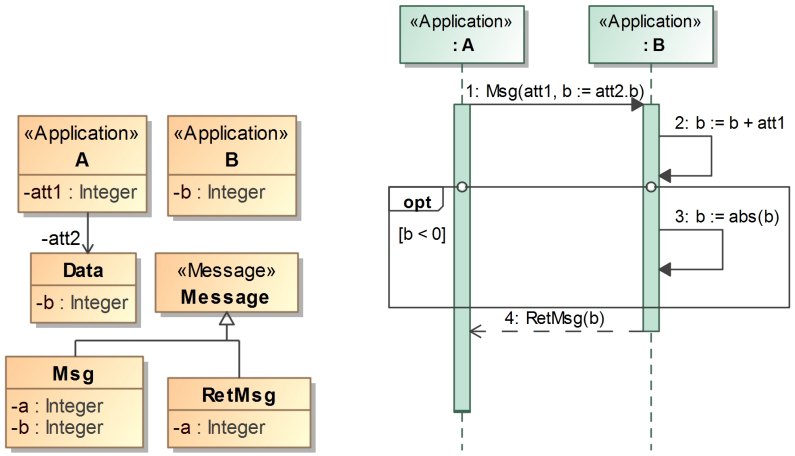
MEL* bietet eine eigene Standardbibliothek von Klassen und Operationen an, die der Entwickler in seiner Anwendung nutzen kann. Diese können verwendet werden, um Listen bzw. Maps zu manipulieren, kryptographische Operationen wie Signierung oder Verschlüsselung auszuführen, oder auf die Sensoren der mobilen Plattform zuzugreifen. Die vordefinierten Klassen und Operationen der MEL*-Standardbibliothek werden im Detail in Unterabschnitt 9.2.4 beschrieben.

11.2 KOMPONENTENINTERAKTION

Die Komponenteninteraktion wird in MODELFLOW mit UML-Sequenzdiagrammen modelliert. Ein Sequenzdiagramm spezifiziert somit einen *Programmablauf*, also den Typ und Reihenfolge von Nachrichten. Die Verarbeitung der Nachrichten wird dabei mit MEL* beschrieben.

Abbildung 11 zeigt das Klassendiagramm sowie einen Ausschnitt eines Sequenzdiagramms einer simplen MODELFLOW-Anwendung, die zur Illustration der Richtlinien zur Modellierung der Komponenteninteraktion verwendet wird. Sie besteht aus zwei Anwendungskomponenten A und B, die zwei Nachrichten austauschen: A schickt

an *B* zwei Ganzzahlen, die *B* addiert, den absoluten Betrag des Ergebnisses in der Variable *b : Integer* speichert, und an *A* zurückschickt.



(a) Klassendiagramm (b) Sequenzdiagramm

Abbildung 11: Illustration des Nachrichtenversands in MODELFLOW

11.2.1 *Komponenten*

Die Komponenten, die im Sequenzdiagramm miteinander interagieren können, sind Anwendungskomponenten (Apps und Webservices), Benutzerschnittstellen, sowie externe Apps und Webservices. Sie werden im Diagramm als UML-Lebenslinien repräsentiert, wobei jeder Komponententyp genau eine Lebenslinie pro Sequenzdiagramm haben darf. Bei Angabe des Typs einer Lebenslinie muss auf die entsprechende UML-Klasse verwiesen werden, die die Komponente im Klassendiagramm repräsentiert (vgl. Modellierung der statischen Sicht auf eine Anwendung in Abschnitt 10.2 und Abschnitt 10.3, sowie Komponenten *A* und *B* in Abbildung 11). Webservice-Komponenten sind dabei zustandslose (*stateless*) Singletons, d.h., es gibt von jeder Webservice-Komponente nur eine Instanz, die nicht die Sitzungsinformationen ihrer Aufrufer speichert.

11.2.2 *Nachrichten*

Nachrichten stoßen *Aktionen* an, die von Komponenten ausgeführt werden (siehe Kapitel 15 für formale Definition einer Aktion). Sie spezifizieren das Verhalten des Systems mit MEL*-Code, der als Name der Nachricht modelliert wird. Es werden drei Nachrichtentypen unterstützt: (1) Initial-Nachrichten zwischen verschiedenen Komponenten (*CallMessage*), (2) Nachrichten von Komponenten an sich selbst (*ToSelfMessage*), sowie (3) Rücknachrichten (*ReplyMessage*). Mit dem

MEL*-Code kann man Nachrichten zwischen Komponenten mit Inhalt füllen, diese Daten auslesen und verarbeiten, auf Komponentenattribute zugreifen, sowie weitere Operationen aufrufen.

Die Nachrichten werden wie folgt modelliert:

1. *CallMessage* ist eine Instanz einer UML-*Synchronous call message*, die zwischen verschiedenen Lebenslinien versandt wird (vgl. Nachricht 1 in Abbildung 11b)
2. *ToSelfMessage* ist eine Instanz einer UML-*Synchronous call message*, deren Sender und Empfänger dieselbe Lebenslinie ist (vgl. Nachrichten 2 und 3 in Abbildung 11b)
3. *ReplyMessage* ist eine Instanz einer UML-*Reply message*, die zwischen verschiedenen Lebenslinien versandt wird (vgl. Nachricht 4 in Abbildung 11b)

Eine *CallMessage* wird zum Versand von Nachrichten verwendet, die im Klassendiagramm als Unterklasse der «`Message`»- bzw. «`Usermessage`»-Klasse modelliert sind. Jede *CallMessage* (mit Ausnahme der vordefinierten Benutzernachrichten) muss eine Nachricht repräsentieren, die zuvor in keinem Programmablauf vorgekommen ist. Die folgenden Komponententypen können eine solche Nachricht empfangen: Anwendungskomponenten, externe Komponenten, sowie die vordefinierte Benutzerschnittstelle.

Der Nachrichtentyp bestimmt eindeutig die Nachrichtenbehandlungsmethode der Empfängerkomponente, vgl. Unterabschnitt 16.2.1

Eine *ReplyMessage* wird zum Versand von Antwortnachrichten verwendet, die im Klassendiagramm als Unterklasse der «`Message`»- bzw. «`Usermessage`»-Klasse modelliert sind. Jede *ReplyMessage* muss eine Nachricht repräsentieren, die zuvor in keinem Programmablauf vorgekommen ist. Die folgenden Komponententypen können eine solche Nachricht empfangen: Anwendungskomponenten, externe Komponenten, sowie die manuelle Benutzerschnittstelle.

Jede *CallMessage* muss mit einer *ReplyMessage* beantwortet werden; einer *ReplyMessage* muss immer eine *CallMessage* vorangehen. Bei der Interaktion mit der vordefinierten Benutzerschnittstelle dürfen nur vordefinierte Benutzernachrichten als *CallMessage* verwendet werden (siehe Unterabschnitt 9.2.4 für Liste der vordefinierten Benutzernachrichten und die Modellierung der Antwortnachrichten).

Der Name der dazugehörigen UML-Message enthält den MEL*-Code, der diese Nachricht durch Aufruf ihres Konstruktors instantiiert, wobei das *create*-Schlüsselwort weggelassen wird (vgl. Unterabschnitt 11.1.2 für MEL*-Syntax). Die folgenden Regeln gelten für den MEL*-Code der *CallMessage* und *ReplyMessage*:

- An den Konstruktor der Nachrichtenklasse können lokale Variablen oder Komponentenattribute der sendenden Komponente als Parameter übergeben werden. Die Anzahl der Parameter entspricht dabei der Anzahl der Attribute der entsprechenden Nachrichtenklasse.

- Als Parameter dürfen nur Identifier oder Zuweisungen verwendet werden, wobei die linke Seite der Zuweisung ein Identifier sein muss.
- Handelt es sich um einen Identifier, der auf Empfängerseite einem Komponentenattribut entspricht, so werden die Daten beim Empfang diesem Attribut zugewiesen.
- Handelt es sich um einen Identifier, der auf Empfängerseite nicht einem Komponentenattribut entspricht, so werden die Daten beim Empfang einer neuen, gleichnamigen lokalen Variable zugewiesen (vgl. Parameter *att1* in Abbildung 11b).
- Handelt es sich um eine Zuweisung, so muss die rechte Seite der Zuweisung einer Variable des Senders entsprechen. Die Daten werden auf Empfängerseite dem Attribut zugewiesen, dessen Name dem Identifier auf der linken Seite der Zuweisung entspricht (vgl. Zuweisung $b := att2.b$ in Abbildung 11b: *A* schickt an *B* die Ganzzahl, die im Feld *b* ihres Komponentenattributs *att2* gespeichert ist, und dem Komponentenattribut *b* von *B* zugewiesen wird). Gibt es kein solches Attribut, so wir eine neue lokale Variable mit diesem Namen angelegt.

Eine *ToSelfMessage* kann eine oder mehrere MEL*-Ausdrücke enthalten (vgl. Unterabschnitt 11.1.2). Darin kann der Modellierer auf die Attribute der Komponente zugreifen, die Sender und Empfänger dieser Nachricht ist. Zudem hat er Zugriff auf lokale Variablen, die beim Nachrichtenempfang automatisch erstellt werden (vgl. Nachrichten 2 und 4 in Abbildung 11b).

11.2.3 *Nachrichtenbeschriftung*

Nachrichten, die als Antwort auf die Benutzernachricht *GetInput<Type>* von der vordefinierten Benutzerschnittstelle versandt werden, können beschriftet werden. Diese Beschriftung beschreibt den Kontext der Eingabeaufforderung, die *GetType<Type>* repräsentiert, und ergänzt somit die Beschriftung des Datentyps *Type*, der als Basis für die Eingabemaske genutzt wird.

Solche Beschriftungen erfüllen die folgenden Aufgaben:

- Sowohl die Beschriftung der Nachricht als auch die des Datentyps werden dem Benutzer während der Eingabeaufforderung angezeigt. Sie erklärt, welche Information vom Nutzer erwartet wird, wobei die Beschriftung des Datentyps erfassen soll, *um welche Daten* es sich handelt, während die Beschriftung der Nachricht festlegt, *in welchem Kontext* diese Daten abgefragt werden.

- Die Beschriftung einer Antwortnachricht kann bei der Modellierung von Informationsflusseigenschaften als Quelle von Information verwendet werden (vgl. Tabelle 7). Da der Endnutzer der Anwendung sowohl ihre modellierten Eigenschaften einsehen kann, als auch die Beschriftung bei der Eingabeaufforderung präsentiert bekommt, auf die die Eigenschaften Bezug nehmen, wird für ihn unmissverständlich klar, welche Informationsflussgarantien für seine Eingabe gelten.

Diese Beschriftung wird durch den Inhalt des *label*-Tags vom Stereotyp «`labeled`» festgehalten, mit der die Nachricht annotiert wird, die als Antwort auf eine *GetInput*<*Type*>-Nachricht modelliert wurde.

11.2.4 *Schleifen und Verzweigungen*

MODELFLOW unterstützt Schleifen und bedingte Verzweigungen, die in Sequenzdiagrammen mit Hilfe von *kombinierten Fragmenten* modelliert werden. Die folgenden drei Fragmente kommen dabei zu Tragen: *alternatives Fragment* (Schlüsselwort: *alt*), *optionales Fragment* (Schlüsselwort: *opt*), sowie *Schleife* (Schlüsselwort: *loop*). Dabei fasst ein *opt*-Fragment Nachrichten als Abläufe zusammen, die ausgeführt bzw. versandt werden sollen, falls die modellierte Bedingung erfüllt ist Ein *alt*-Fragment ist eine Erweiterung des *opt*-Fragments, bei dem zusätzlich Nachrichten zusammengefasst werden können, die im Falle der nicht erfüllten Bedingung ausgeführt bzw. versandt werden sollen. Ein *loop*-Fragment fasst Nachrichten zusammen, die wiederholt ausgeführt werden, solange die modellierte Bedingung erfüllt ist. Als Bedingung kann dabei ein beliebiger boolescher MEL*-Ausdruck verwendet werden.

Die folgenden Modellierungsrichtlinien gelten für alle drei Fragmenttypen:

- Kombinierte Fragmente dürfen beliebig geschachtelt werden.
- Programmabläufe dürfen nicht mit einem kombinierten Fragment beginnen.
- Der Ablauf innerhalb eines kombinierten Fragments muss bei derselben Komponente anfangen und aufhören.
- Der Ablauf innerhalb eines kombinierten Fragments darf nicht bei Anwendungsmodulen wie Benutzerschnittstellen oder externen Komponenten anfangen oder aufhören
- Ein kombiniertes Fragment darf keine *ReplyMessage* enthalten, wenn die dazugehörige *CallMessage* außerhalb des Fragments liegt.

- Der Geltungsbereich einer lokalen Variable endet mit dem Beginn bzw. Ende eines kombinierten Fragments, das eine oder mehrere *CallMessage* bzw. *ReplyMessage* enthält, sowie mit einer *ReplyMessage*.

Abbildung 11b zeigt ein Beispiel für die Modellierung eines *opt*-Fragments. Dabei wird der MEL*-Code der zweiten *ToSelfMessage* (Nachricht 3 im Sequenzdiagramm) nur ausgeführt, wenn der Wert vom Komponentenattribut *b* kleiner Null ist.

Zudem unterstützt MODELFLOW *implizite* Verzweigungen. Wird dem Benutzer eine Benutzeroberfläche angezeigt, so kann er die Eingabeaufforderung abbrechen und zum Anfang des modellierten Programmablaufs zurückkehren. Dies muss nicht explizit modelliert werden, sondern wird bei jeder Nachricht an die vordefinierte Benutzeroberfläche implizit angenommen, und bei der Informationsflussanalyse berücksichtigt (vgl. Unterunterabschnitt 17.2.2.4).

11.2.5 *Programmabläufe*

Ein MODELFLOW-Modell darf mehr als ein Sequenzdiagramm besitzen, um mehrere mögliche Programmabläufe zu spezifizieren. Ist dies der Fall, so muss jedes Sequenzdiagramm mit einer *CallMessage* der manuellen Benutzeroberfläche an eine «`Application`»-Anwendungskomponente beginnen. Dies entspricht einer Benutzerinteraktion, mit der ein durch das Sequenzdiagramm modelliertes Programmablauf angestoßen wird, und ermöglicht dadurch die Entwicklung einer *ereignisbasierten* Anwendung.

Handelt es sich um eine simple Anwendung mit nur einem modellierten Programmablauf, so kann die manuelle Benutzeroberfläche weggelassen werden, und die Komponenteninteraktion muss mit einer Nachricht an die vordefinierte Benutzerschnittstelle beginnen.

Die in diesem Kapitel präsentierte dynamische Sicht auf die Travel Planner-Fallstudie illustriert den Fall mit nur einem Programmablauf (siehe Abschnitt 11.4). Für eine komplexe Anwendung mit einer Vielzahl von modellierten Programmabläufen, siehe z.B. Kapitel 22.

11.2.6 *Einschränkungen*

Um die Korrektheit der Informationsflussanalyse zu gewährleisten, müssen folgende Regeln eingehalten werden.

- Attribute, die Maps mit zugriffsgeschützten Einträgen abbilden (vgl. Abschnitt 10.4), dürfen weder als linke oder rechte Seite einer Zuweisung agieren, noch als Parameter einer Methode verwendet werden. Damit wird sichergestellt, dass der Zugriff auf

die geschützte Einträge nur über die vordefinierten Methoden möglich ist (siehe Unterabschnitt 9.2.4 für die Auflistung der Methoden und Unterunterabschnitt 17.2.1.5 für Behandlung solcher Einträge bei der Informationsflussanalyse).

- Die linke Seite der Zuweisung beim Aufruf der vordefinierten *decrypt*-Methode muss einem Komponentenattribut entsprechen. Damit wird garantiert, dass der Fluss von Klartextinformationen zu einer Komponente bei der Codeanalyse entdeckt wird (vgl. Unterunterabschnitt 16.2.2.1).

- Ein modellierter Programmablauf darf nur einen Aufruf eines konkreten Webservices enthalten, wenn dadurch der interne Zustand des Services (durch Zuweisung an einen seiner Attribute) verändert wird. Dadurch wird sichergestellt, dass bei der Informationsflussanalyse jeder in der Realität mögliche Programmablauf berücksichtigt wird: ein Angreifer kann jederzeit mit dem Webservice kommunizieren, was bei der Informationsflussanalyse auf eine beliebige Aktion des Angreifers vor und nach einem modellierten Programmablauf abgebildet wird (siehe Unterabschnitt 12.2.4 für Modellierung eines Angreifers und Unterunterabschnitt 17.2.1.5 für das Code-Refinement bezüglich der möglichen Programmabläufe).

11.3 MEL*-METHODEN

Als MEL*-Methoden wird die Spezifikation von Operationen auf Modellebene mit Hilfe von Aktivitätsdiagrammen bezeichnet (vgl. Abschnitt 10.1 für Modellierung der dazugehörigen Klassenoperationen, sowie Abschnitt 10.1 für Zweck und Anwendung von MEL*-Methoden).

Die Modellierung solcher Methoden ist angelehnt an die Modellierung von Subdiagrammen, wie sie im SecureMDD-Ansatz definiert wurden (vgl. [65]). Im Gegensatz dazu verwendet man dabei MEL* statt der Model Extension Language, wodurch der Modellierer Zugriff auf die MODELFLOW-Standardbibliothek von Datentypen und Operationen hat, sowie Schleifenkonstrukte verwenden kann.

Jede MEL*-Methode wird dabei durch ein eigenes Aktivitätsdiagramm modelliert, das genau einen Aktivitätsbereich enthält. Der Name des Aktivitätsbereichs ist dabei der Name der Komponente, die die dazugehörige Klassenoperation enthält, konkateniert mit zwei Doppelpunkten, sowie dem Namen der Operation.

Der Methode können Argumente übergeben werden, die im Diagramm als UML-*ActivityParameterNodes* repräsentiert werden, und mit einem UML-*JoinNode* zusammengeführt werden. Der Ausgabeparameter wird ebenfalls als ein *ActivityParameterNode* modelliert. Diese Ein- und Ausgabeparameter müssen den Ein- und Ausgabeparame-

tern der dazugehörigen Klassenoperation entsprechen. Die Methode darf nicht auf Komponentenattribute zugreifen, daher müssen auch diese explizit an die Methode als Parameter übergeben werden.

Das Verhalten der Methode wird mit UML-*ActivityNodes* festgelegt, die MEL*-Code als Name enthalten, während der Programmfluss durch die UML-*ControlFlows* zwischen den *ActivityNodes* modelliert wird.

Verzweigungen werden mit UML-*DecisionNodes* modelliert, die zwei ausgehende *ControlFlows* haben. Die UML-*Guards* auf den *ControlFlows* legen die Bedingungen fest, wann welcher Kontrollfluss ausgeführt werden kann. Wie bei den kombinierten Fragmenten kann hierbei ein beliebiger boolescher MEL*-Ausdruck als Verzweigungsbedingung verwendet werden.

Schleifen werden mit dem MEL*-Sprachkonstrukt *for* spezifiziert (vgl. Regel `ForExpr` der konkreten MEL*-Syntax in Abbildung 59).

Die Modelle der Fallstudien *Distance Tracker*, *ContanctSMSManager* und *Private Taxi* illustrieren die Modellierung und Anwendung solcher MEL*-Methoden (siehe Teil v).

11.4 DYNAMISCHE SICHT AUF DIE TRAVEL PLANNER-FALLSTUDIE

Die vorgestellten Modellierungsrichtlinien sollen nun anhand der dynamischen Sicht auf das Modell der Travel Planner-Fallstudie illustriert werden.

In Abbildung 12 ist das Sequenzdiagramm der Anwendung abgebildet. An dem Programmablauf nehmen die folgenden Komponente teil, die als Lebenslinien repräsentiert sind: vordefinierte Benutzerschnittstelle (*User*), Reiseplanner-App (*TravelPlanner*), App zur Kreditkartenverwaltung (*CreditCardCenter*), Webservice der Reiseagentur (*TravelAgency*), sowie der Webservice der Fluglinie (*Airline*).

Der Programmablauf lässt sich in die folgenden Schritte untergliedern:

1. Abfrage der Reisedaten des Benutzers (Nachrichten 1-2)
2. Abholen passender Flugangebote der Fluglinie über den Webservice der Reiseagentur (Nachrichten 3-7)
3. Auswahl eines Flugangebots durch den Benutzer (Nachrichten 8-9)
4. Abfrage der Kreditkartendaten des Nutzer von der Kreditkarten-App (Nachrichten 10-14)
5. Buchung des ausgewählten Flugs mit den Kreditkartendaten (Nachrichten 15-16)

6. Bezahlung der Kommission an die Reiseagentur, und Buchungsbestätigung (Nachrichten 17-21)

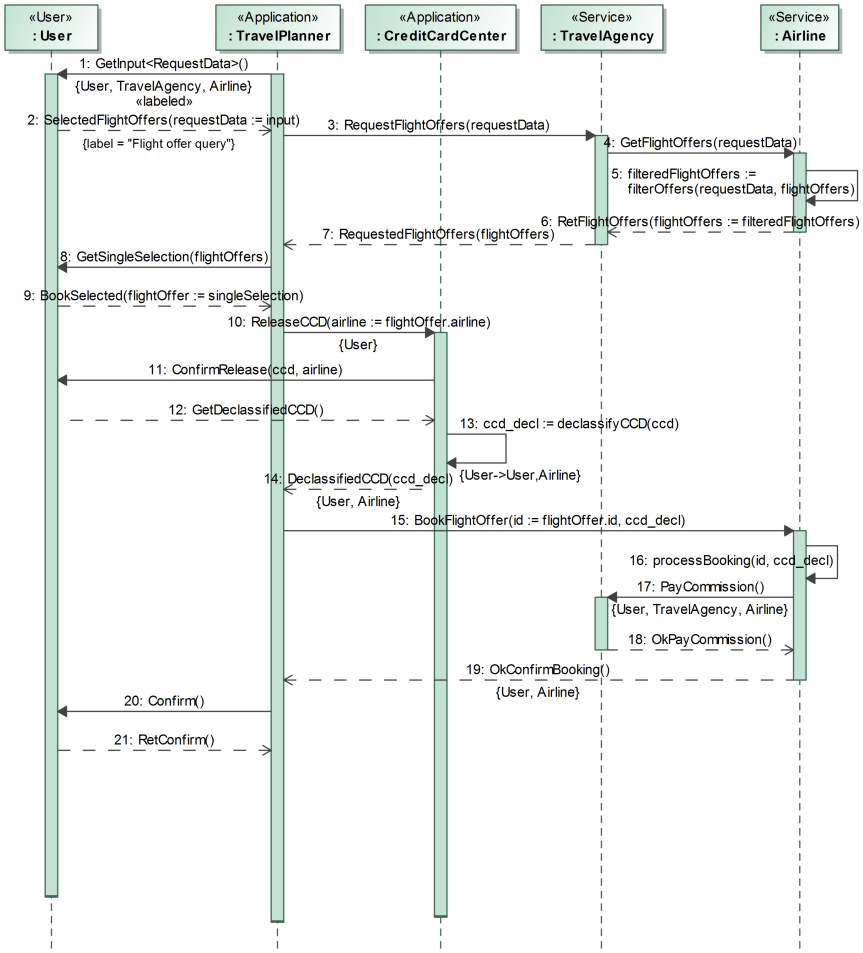


Abbildung 12: Sequenzdiagramm der Travel Planner-Anwendung

(1) ABFRAGE DER REISEDATEN Zur Abfrage der Reisedaten schickt die Anwendungskomponente *TravelPlanner* die vordefinierte Nachricht *GetInput<RequestData>* an die vordefinierte Benutzerschnittstelle. Die Attribute des Datentyps in den spitzen Klammern bestimmen, welche Eingabefelder dem Benutzer angezeigt werden (vgl. Klasse *RequestData* in Abbildung 10). Die Antwortnachricht greift auf das Attribut *input* der *User*-Komponente zu, das die Benutzereingabe repräsentiert, und weist diesen der Variable *requestData* auf Empfängerseite zu. Die Nachricht ist durch den «`labeled`»-Stereotyp mit *„Flight offer query“* beschriftet. Die Beschriftung wird dem Benutzer bei der Eingabeaufforderung angezeigt, und erklärt, wofür die angefragten Informationen verwendet werden.

(2) ABHOLEN DER FLUGANGEBOTE *TravelPlanner* sendet die Benutzereingabe an *TravelAgency*, die sie an *Airline* weiterleitet. Diese filtert die in ihrem Komponententattribut *flightOffers* gespeicherten Flugangebote anhand der Benutzereingabe mit Hilfe der manuellen Methode *filterOffers*. Das Ergebnis *filteredFlightOffers* schickt sie zurück an *TravelPlanner* über die Komponente *TravelAgency*.

(3) AUSWAHL EINES FLUGANGEBOTS *TravelPlanner* zeigt dem Benutzer eine Liste von den erhaltenen Flugangeboten mit Hilfe der Benutzernachricht *GetSingleSelection* zur Auswahl an. Die Antwortnachricht greift auf das Attribut *selection* der *User*-Komponente zu, die die Benutzerauswahl repräsentiert, und weist diese dem Attribut *flightOffer* der *TravelPlanner*-Komponente zu.

(4) ABFRAGE DER KREDITKARTENDATEN *TravelPlanner* schickt an *CreditCardCenter* die Nachricht *ReleaseCCD*, um die Kreditkartendaten zu erhalten. Die Nachricht enthält den Namen der Fluglinie, die im Attribut *airline* des Flugangebots *flightOffer* gespeichert ist.

CreditCardCenter zeigt dem Nutzer mittels vordefinierter Nachricht *ConfirmRelease* ein Bestätigungsdialog mit dem Namen der Fluglinie sowie den Kreditkartendaten an. Wird die Freigabe der Kreditkartendaten an die Fluglinie vom Benutzer bestätigt, wird diese explizit mit der Deklassifikationsoperation *declassifyCCD* deklassifiziert, und das Ergebnis an *TravelPlanner* verschickt. Bricht der Benutzer ab, so wird der Programmablauf an den Anfang (Schritt (1)) zurückgesetzt.

(5) BUCHUNG DES FLUGS Hat der Nutzer die Freigabe seiner Kreditkartendaten an die Fluglinie bestätigt, wird an *Airline* die Nachricht *BookFlightOffer* mit der ID des Flugangebots sowie den Kreditkartendaten verschickt. Die Fluglinie wickelt den Kaufvorgang mittels der manuellen Methode *processBooking* ab.

(6) KOMMISSION UND BUCHUNGSBESTÄTIGUNG Um die Reiseagentur von der erfolgreichen Kundenvermittlung zu informieren, verschickt *Airline* die Nachricht *PayCommission* an *TravelAgency*. Diese Nachricht kann zur Bezahlung einer Kommission an die Reiseagentur genutzt werden, wovon im vorgestellten Modell abstrahiert wird. Sobald *TravelPlanner* die Antwortnachricht *OkConfirmBooking* von *Airline* erhalten hat, wird dem Benutzer eine Buchungsbestätigung mit Hilfe der vordefinierten Benutzernachricht *Confirm* angezeigt. Der Bestätigungstext wird einer Konfigurationsdatei der App entnommen, und kann vom Entwickler vor dem Deployment der Anwendung angepasst werden (vgl. Artefakte der finalen Anwendung in Abbildung 2).

In dieser Fallstudie werden die Kreditkartendaten des Benutzers (oder davon abgeleitete Daten) niemals an die Reiseagentur übermit-

telt, während die Fluglinie die Kreditkartendaten nur nach expliziter Benutzerbestätigung erfährt. Die Formalisierung dieser Eigenschaften wird im folgenden Kapitel beschrieben, und erfolgt unter Anderem mit Hilfe von Nachrichtenannotation in geschweiften Klammern (siehe Nachrichten 1, 10, 13, und 14), die die Sicherheitsdomäne der Nachrichteninhalte bestimmen.

12 INFORMATIONSFLUSSMODELLIERUNG

Dieser Abschnitt beschäftigt sich mit dem Kernthema des vorgestellten Ansatzes zur Entwicklung informationsflusssicherer Systeme: der Modellierung von Informationsflussannotationen und -eigenschaften.

Diese erfüllen eine Reihe von Funktionen:

- Der Entwickler der Anwendung ist bereits in der Designphase veranlasst, die Informationsflusssicherheit zu berücksichtigen. Dies beeinflusst seine Entscheidungen, welche Informationen als geheim angesehen werden sollen, wo sie gespeichert werden dürfen, und wie die Komponenteninteraktion auszusehen hat. So können bereits in frühen Entwicklungsphasen Fehler entdeckt und ausgebessert werden, was auch die nachfolgende Informationsflussanalyse vereinfachen kann.

- Informationsflusseigenschaften werden durch ihre Modellierung formalisiert, und können bei der Informationsflussanalyse der Anwendung verwendet werden. Dadurch ist es für den Modellierer nicht notwendig, mit den der Analyse zugrunde liegenden formalen Methoden vertraut zu sein, um die gewünschten Sicherheitseigenschaften präzise ausdrücken zu können. Somit ist die Modellierungssprache unabhängig von einem bestimmten formalen Modell oder Informationsflussanalysetool.

- Zur Sicherheit und Privatsphäre gehört nicht nur die Geheimhaltung von sensitiven Daten, sondern auch die Transparenz des Systems in Bezug darauf, welche Informationen es sammelt, sowie mit wem (und unter welchen Umständen) sie geteilt werden. Dafür müssen Informationsflusseigenschaften nicht nur formal garantiert werden, sondern auch knapp, präzise, und verständlich darstellbar sein. Nur so können die Benutzer der Anwendung eine informierte Entscheidung treffen, ob und wie sie mit dem System interagieren wollen.

In Abschnitt 12.1 wird die Informationsflusspolicy vorgestellt, die die Sicherheitsdomänen von Daten und Interaktionen sowie ihre Interferenz-Relation festlegen. Sie formalisiert den erlaubten Fluss von Informationen durch das System, und ist notwendig für die formale Verifikation.

Abschnitt 12.2 beschreibt die Modellierung der Informationsflusseigenschaften. Zusätzlich zu der Sicherheitspolicy können sie komplexe Sicherheitsgarantien ausdrücken, indem sie z.B. die Umstände der expliziten Informationsfreigabe formalisieren. Sie bilden die

Grundlage für die automatische Informationsflussanalyse, und können aufgrund ihrer intuitiven Darstellung auch dem Benutzer präsentiert werden.

Sicherheitsdomänen, die Interferenz-Relation, sowie die Informationsflusseigenschaften werden mit Hilfe von Aktivitätsdiagrammen modelliert, die im Paket mit dem Stereotyp «`InformationFlowProperties`» abgelegt werden.

12.1 SICHERHEITSPOLICY

Die Sicherheitspolicy bildet in MODELFLOW die Grundlage für die Spezifikation des erlaubten Informationsflusses innerhalb des modellierten Systems. Sie besteht aus Sicherheitsdomänen, ihrer Interferenz-Relation, sowie den Sicherheitsannotation der Komponentenattribute und -nachrichten. Die Domänen der einfachsten sinnvollen Sicherheitspolicy werden in der Fachliteratur oft *high* und *low* genannt, und entsprechen der „hohen“ bzw. „niedrigen“ Sicherheitsstufe der Information. Die Sicherheitsstufe der öffentlichen Information darf dabei jederzeit erhöht werden, während geheime Information nur in bestimmten Fällen öffentlicher werden darf.

In MODELFLOW ist die Sicherheitspolicy notwendig für die formale Verifikation der Anwendung (siehe Kapitel 15), und optional, falls nur der automatische Informationsflusscheck auf der Codeebene eingesetzt werden soll (siehe Kapitel 16). Unabhängig davon, wie die Sicherheitseigenschaften garantiert werden sollen, wird der Entwickler durch die Modellierung einer Sicherheitspolicy bereits in der Designphase veranlasst, Informationsflusssicherheit der zu entwickelnden Anwendung zu berücksichtigen. Zum Einen beeinflusst die Policy die Daten- und Kommunikationsstruktur der Anwendung, da der Modellierer die Komponentenattribute und Nachrichten sinnvoll annotieren können muss. Zum Anderen können explizit modellierte Informationsflusseigenschaften mit der Sicherheitspolicy verglichen werden, und so auf Plausibilität und Konsistenz überprüft werden.

Abbildung 13 illustriert die Modellierung von Sicherheitdomänen und ihrer Interferenz-Relation anhand der Travel Planner-Fallstudie. Ihre vollständigen und annotierten Klassen- und Sequenzdiagramme sind in Abbildung 9 und Abbildung 12 abgebildet.

12.1.1 *Sicherheitsdomänen*

Eine Sicherheitspolicy definiert eine Menge von Sicherheitsdomänen (manchmal auch Sicherheitslevels bzw. -stufen bezeichnet), die zur Annotation von Komponentenattributen und Nachrichten verwendet werden. Dadurch legt man fest, wie vertraulich die darin gespeicherten oder verschickten Informationen sind bzw. sein dürfen.

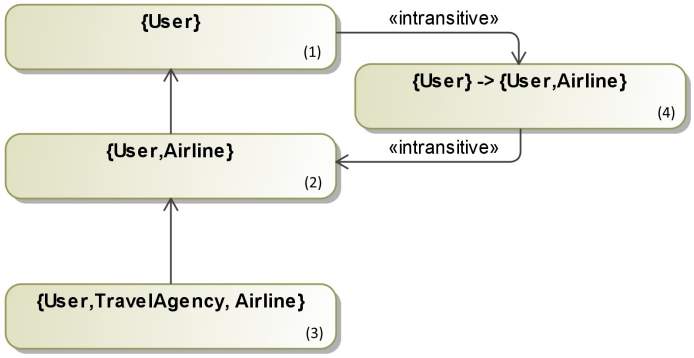


Abbildung 13: Sicherheitsdomänen und ihre Interferenz-Relation in der Travel Planner-Fallstudie

In MODELFLOW werden Sicherheitsdomänen mit einem Aktivitätsdiagramm modelliert. Eine Sicherheitsdomäne wird dabei durch einen UML-Aktivitätsknoten repräsentiert, und ihre Bezeichnung entspricht dem Namen dieses Knotens.

Dem Modellierer ist freigestellt, die Domänenbezeichnungen nach Belieben zu wählen. Aus Gründen der Lesbarkeit bietet es sich jedoch an, eine Domäne nach der Menge von Komponenten zu benennen, die die damit annotierten Daten lesen dürfen. Besteht also eine Anwendung aus der Menge der Komponenten $C = \{User, Airline, TravelAgency\}$, so erhält die Domäne den Namen $d \subseteq C$ (vgl. Domänen (1), (2), und (3) in Abbildung 13).

Sollen Informationen öffentlicher gemacht werden können, so muss hierfür eine explizite Deklassifikationsdomäne eingeführt werden. Ihre Bezeichnung d_{decl} ergibt sich dabei aus der Menge der Komponenten, die auf die Informationen vor der Deklassifikation zugreifen durften, konkateniert mit einem ASCII-Pfeil `->`, sowie der Menge der Komponenten, die die Informationen nach der Deklassifikation lesen können ($d_{decl} \subseteq C \times C$). Kann beispielsweise nach der Deklassifikation zusätzlich zur Komponente *User* die Komponente *Airline* die Daten lesen können, so erhält die Deklassifikationsdomäne den Namen *{User}->{User, Airline}* (vgl. Domäne (4) in Abbildung 13).

12.1.2 *Interferenz-Relation*

Die Sicherheitspolicy legt die Interferenz-Relation (kurz: $\rightsquigarrow$, siehe Kapitel 15 für eine formale Definition) zwischen den Sicherheitsdomänen fest, die den erlaubten Fluss der Information in der Anwendung bestimmt: *interferiert* (bzw. *beeinflusst*) eine Domäne d_1 die Domäne d_2 ($d_1 \rightsquigarrow d_2$), so darf Information von d_1 zu d_2 fließen, aber nicht umgekehrt.

Die Interferenz-Relation (und somit die Richtung, in die Information fließen darf) wird in MODELFLOW durch Kontrollflusskanten

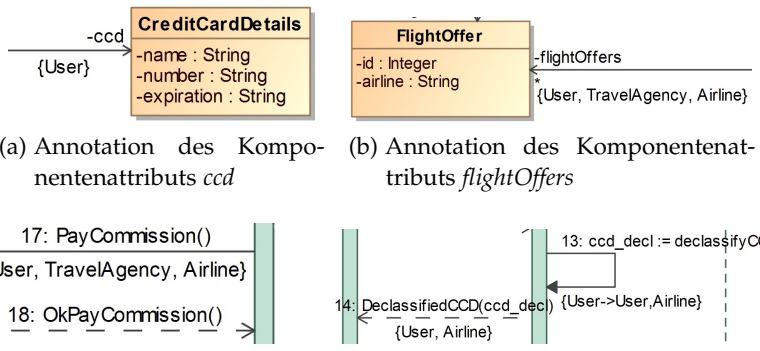
zwischen Aktivitätsknoten, die Domänen repräsentieren, abgebildet. Die Relation ist automatisch reflexiv, und gibt eine direkte Interferenz zwischen zwei Domänen an; sind die Kontrollflüsse zudem nicht mit dem Stereotyp «`interansitive`» markiert, so sind sie implizit transitiv. Es ist ebenfalls möglich, dass zwei Domänen in keinerlei Beziehung zueinander stehen, und somit kein Informationsfluss zwischen ihnen erlaubt ist. Zyklen durch nicht markierte Kontrollflüsse sind dabei nicht erlaubt.

In Abbildung 13 interferiert die Domäne *{User, TravelAgency, Airline}* somit die Domäne *{User, Airline}* sowie implizit die Domäne *{User}*

Um festzulegen, dass Daten öffentlicher gemacht werden können, muss Informationsfluss auch in die umgekehrte Richtung erlaubt werden. In MODELFLOW ist ein solcher inverser Fluss nur über spezielle Deklassifikationsdomänen möglich (vgl. Domäne (4) in Abbildung 13). Beide Kontrollflusskanten von und zur Deklassifikationsdomäne müssen entgegen der Richtung der nicht markierten Kanten modelliert werden, sowie intransitiv sein und daher mit dem Stereotyp «`intransitive`» versehen werden. Deklassifikationsdomänen müssen genau eine ein- und eine ausgehende Kante haben. Damit dürfen Informationen nicht direkt von der Ursprungs- zur Zieldomäne fließen, sondern nur über die Deklassifikationsdomäne.

Die in Abbildung 13 modellierte Sicherheitspolicy drückt somit aus, dass Information immer geheimer werden darf, indem sie u.a. von *{User, Airline}* (2) zu *{User}* (1) fließt; sie kann jedoch nur öffentlich werden, wenn sie von *{User}* (1) zu *{User}->{User, Airline}* (4), und von dort zu *{User, Airline}* (2) fließt. Da die Kanten von und zur Domäne (4) als intransitiv markiert sind, darf Information nicht direkt von (1) zu (2) fließen.

12.1.3 *Sicherheitsannotation*



(a) Annotation des Komponentenattributs *ccd*

(b) Annotation des Komponentenattributs *flightOffers*

(c) Annotation der Nachricht *PayCommission*

(d) Annotation des Deklassifikationsmethodenaufrufs *declassifyCCD*

Abbildung 14: Sicherheitsannotationen in der Travel Planner-Fallstudie

Um erlaubte Informationsflüsse innerhalb einer MODELFLOW-Anwendung zu bestimmen, müssen Komponentenattribute und Sequenzdiagrammnachrichten mit Sicherheitsdomänen annotiert werden. Diese Klassifikation geschieht durch das Anbringen einer UML-Zusicherung an das Attribut bzw. die Sequenzdiagrammnachricht. Die Zusicherung muss dabei den Namen einer modellierten Sicherheitsdomäne tragen.

ANNOTATION DER KOMPONENTENATTRIBUTE Erhält ein Komponentenattribut die Sicherheitsdomäne d, so bestimmt diese die untere Schranke für die Sicherheitsdomäne d_h der Nachricht, die das Attribut lesen darf; es muss also $d \rightsquigarrow d_h$ gelten. Dadurch kann man als Modellierer beispielsweise festlegen, dass der Code einer weniger vertrauenswürdigen (und somit „öffentlicheren“) Komponente die im Attribut gespeicherte Information nicht lesen darf. Zudem bestimmt man mit d die obere Schranke für die Sicherheitsdomäne d_l der Nachricht, die das Attribut schreiben darf ($d_l \rightsquigarrow d$). Damit darf das Attribut keine geheimere Information speichern, als die Domäne d es zulässt.

Abbildung 14a zeigt ein Anwendungsattribut *ccd*, das geheime Informationen speichert (siehe Abbildung 9 für das komplette Klassendiagramm). Das Attribut ist mit der Sicherheitsdomäne *User* annotiert; es darf also nur der Benutzer, bzw. Aktionen mit der Sicherheitsdomäne *{User}* oder *{User}->{User, Airline}* gelesen werden (vgl. Interferenz-Relation in Abbildung 13). Das Attribut *flightOffers* (siehe Abbildung 14a) ist dagegen öffentlicher, und daher mit der Sicherheitsdomäne *{User, TravelAgency, Airline}* annotiert.

ANNOTATION DER NACHRICHTEN Annotiert der Modellierer eine Nachricht *Msg* mit der Domäne d, so legt er dadurch fest, dass nur vertrauenswürdiger(er) Code mit der Domäne d_h die von *Msg* veränderten Variablen und Attribute lesen darf ($d \rightsquigarrow d_h$). Gleichzeitig bestimmt d die obere Schranke für die Domäne d_l der Nachricht, die Variablen und Attribute schreibt, auf die der Code von *Msg* lesend zugreift. Handelt es sich bei *Msg* um eine *Call*- bzw. *Reply*-Nachricht, so bedeutet es insbesondere, dass der Empfänger der Nachricht keine Information erhalten darf, die geheimer als d ist. Dies ist beispielsweise in Abbildung 14c der Fall: mit der Nachricht (17) dürfen an den Empfänger (die *Airline*) lediglich Informationen versandt werden, die mit *{User, TravelAgency, Airline}* (oder einer öffentlicheren Domäne, falls eine solche modelliert worden wäre) annotiert sind.

MEL*-*Code einer Nachricht verändert den Inhalt von Variablen und Attribute durch explizite Zuweisungen sowie das Versenden von Nachrichten, was dem Schreiben der lokalen Variablen des Empfängers entspricht*

Diese Regeln sind ähnlich zu den bekannten Eigenschaften **no-read-up** und **no-write-down** des Bell-LaPadula-Sicherheitsmodells (siehe Kapitel 15).

Der Aufruf einer Deklassifikationsoperation, die im Klassendiagramm mit dem Stereotyp «`declassify`» versehen ist, muss als ei-

ne eigenständige *ToSelf*-Nachricht modelliert werden, und mit einer Deklassifikationsdomäne d_{decl} annotiert werden (mit $d_h \rightsquigarrow d_{decl}$ und $d_{decl} \rightsquigarrow d_l$). Dadurch legt der Modellierer fest, dass die an die Deklassifikationsoperation als Parameter übergebene, geheime Information genau die Sicherheitsdomäne d_h haben muss, während das Ergebnis der Operation die öffentlichere Domäne d_l erhält. Anders ausgedrückt, dürfen nur Nachrichten mit der Domäne d_h Variablen und Attribute schreiben, die als Parameter übergeben werden, während das Rückgabeparameter nur von einer Nachricht mit der Domäne d_l gelesen werden können. So muss in Abbildung 14d das Attribut *ccd* in Nachricht (13) explizit deklassifiziert werden, bevor sie mit der öffentlicheren Nachricht *DeclassifiedCCD* (14) versandt werden darf.

AUTOMATISCHE DOMÄNENPROPAGATION In MODELFLOW ist es nicht notwendig, jede Nachricht explizit mit einer Sicherheitsdomäne zu versehen, da sie in vielen Fällen automatisch aus den vorhandenen Annotationen berechnet werden kann. So erhält eine nicht annotierte Nachricht automatisch die Domäne der im Programmablauf vorhergehender Nachricht. Dadurch ist es in einem modellierten Programmablauf lediglich notwendig, die erste Nachricht, sowie die Nachrichten, die eine andere Domäne als die vorherige haben sollen, zu annotieren. Letzteres ist beispielsweise bei *ToSelf*-Nachrichten notwendig, die Deklassifikationsmethoden aufrufen. In Abbildung 14c ist die Nachricht (18) nicht explizit annotiert, und erhält daher automatisch die Domäne *{User, TravelAgency, Airline}* der vorhergehender Nachricht (17).

Die Annotation der Komponentenattribute ist in MODELFLOW optional, da ihre Domäne oft aus der Annotation der Nachrichten berechnet werden kann, die diese schreiben bzw. lesen. Falls sich jedoch die Domänen dieser Nachrichten nicht interferieren, so kann die Domäne des Attributs nicht berechnet werden, was bei der Modellvalidierung dem Entwickler gemeldet wird.

12.2 INFORMATIONSFLUSSEIGENSCHAFTEN

Durch die in Abschnitt 12.1 vorgestellte Sicherheitspolicy lässt sich der erlaubte Informationsfluss innerhalb der Anwendung ausdrücken. Jedoch ist es damit nicht möglich, komplexere Informationsflusseigenschaften, die bei realistischen Anwendungen oft notwendig sind, zu modellieren. Dazu zählt, *unter welchen Umständen* geheime Informationen freigegeben werden dürfen, und *welche Komponente* diese Freigabe vornehmen darf.

Die Policy eignet sich zudem nicht dazu, dem Endnutzer der Anwendung präsentiert zu werden. Für jemanden, der mit den Grundlagen der Informationsflusstheorie nicht vertraut ist, ist die Policy nicht intuitiv verständlich, zudem man das gesamte Anwendungsmodell

und dessen Annotationen untersuchen muss, um daraus die einzelnen Informationsflusszusicherungen wie „Fluglinie erfährt nicht meine Kreditkartendaten“ abzuleiten.

Daher bietet MODELFLOW die Möglichkeiten, solche intuitiven Informationsflusseigenschaften einzeln zu modellieren. Damit ist es auch möglich, die Umstände und den Initiator einer Deklassifikation auszudrücken, sowie den erlaubten oder verbotenen Informationsfluss zwischen Komponenten, ihren Attributen, der Benutzereingabe, sowie verschiedenen Nutzern der Anwendung zu modellieren. Somit sind die MODELFLOW-Informationsflusseigenschaften ausdrucksstärker als eine Sicherheitspolicy, die Sicherheitsdomänen und ihre Interferenz-Relation festlegt. Der Fokus solcher Eigenschaften liegt darauf, was mit der für den Benutzer wichtigen Information passieren darf, ohne dass abstrakte Konzepte wie Sicherheitsdomänen und Interferenz-Relation hinzugezogen werden müssen.

Solche Eigenschaften werde mit Hilfe von UML-Aktivitätsdiagrammen modelliert. Jede Eigenschaft ist dabei durch einen strukturierten Aktivitätsknoten repräsentiert, die als Namen ihre intuitive Beschreibung in Textform erhält. Die Diagramme modellieren *Quellen* von Informationen als *Send Signal Actions*, während die *Senken* von Information als *Accept Event Actions* modelliert werden. Der Name der jeweiligen *Action* spezifiziert die konkrete Informationsflussquelle oder -senke, die entweder eine Komponente, ein Komponentenattribut, ein plattformspezifisches Datum, oder eine bestimmte Benutzereingabe sein kann (vgl. Tabelle 7). Eine Eigenschaft darf entweder eine *Send Signal Action* sowie mehrere *Accept Event Actions*, eine *Accept Event Action* sowie mehrere *Send Signal Actions*, oder genau eine *Accept Event Action* sowie genau eine *Send Signal Action* enthalten.

Eine *Action* kann auch mehrere konkrete Quellen und Senken zusammenfassen, indem diese im Namen der *Action* durch Strichpunkt getrennt aufgeführt werden.

12.2.1 *Transitive Nichtinterferenz-Eigenschaften*

In MODELFLOW ist es möglich, Informationsflusseigenschaften zu modellieren, die durch eine transitive Interferenz-Relation ausgedrückt werden können. Dies entspricht in MODELFLOW einer Sicherheitspolicy, die keine intransitiven Kanten hat, und somit keine Deklassifikation durch besondere Domänen erlaubt. In der Informationsflusstheorie redet man dabei von *transitiver Nichtinterferenz* (was irreführend sein kann, da man sich dabei auf die Transitivität der Interferenz-Relation bezieht [78]).

Die für die Verifikation *solcher Eigenschaften generierte Sicherheitspolicy kann sehr wohl neue, intransitive Kanten enthalten, um implizite Deklassifikation zu ermöglichen (siehe Unterunterabschnitt 15.2.1.2 für Details)*

Eigenschaften, die durch eine transitive Interferenz-Relation ausgedrückt werden können, werden im Folgenden *transitive Nichtinterferenz*-Eigenschaften genannt. Eigenschaften, die durch eine intransi-

TYP	SYNTAX	BESCHREIBUNG
Komponente	*Component*	Name der UML-Klasse, die *Component* repräsentiert
Komponentenattribut	*Component.att*	Name des Attributs *att* der UML-Klasse, die *Component* repräsentiert
Benutzereingabe	*„label“*	Wert des «`labeled`»-*label*-Tags, mit dem die Antwort auf eine Benutzernachricht markiert ist (siehe Unterabschnitt 11.2.3 für Diskussion)
Plattformzpezifische Quelle	*PredefinedSources.x*	Element *x* der vordefinierten Enumeration *PredefinedSources* (vgl. Abbildung 18)
Plattformzpezifische Senke	*PredefinedSinks.y*	Element *y* der vordefinierten Enumeration *PredefinedSinks* (vgl. Abbildung 18)

Tabelle 7: Erlaubte Namen einer Quelle bzw. Senke: konkrete Syntax

tive Interferenz-Relation ausgedrückt werden müssen, werden dagegen *intransitive Nichtintereferenz*-Eigenschaften genannt.

INFORMATIONSFLUSSSPEZIFIKATION Um auszudrücken, dass zwischen einer Informationsquelle und -senke kein Informationsfluss stattfindet, modelliert man einen strukturierten Aktivitätsknoten, und fügt ihm eine *Send Signal Action* sowie eine *Accept Event Action* hinzu, die jeweils die gewünschte Quelle und Senke repräsentieren. Diese werden durch einen Kontrollfluss von Quelle zu Senke verbunden, der mit dem Stereotyp «`noFlow`» markiert wird. Soll der Informationsfluss erlaubt sein, so muss die Kontrollflusskante mit dem Stereotyp «`allowedFlow`» markiert werden.

Abbildung 15 zeigt eine Eigenschaft der Travel Planner-Fallstudie, die den Informationsfluss der Kreditkartendaten, die im Attribut *ccd* der Komponente *CreditCardCenter* gespeichert sind, zur Komponente *TravelAgency* verbietet. Sie entspricht der intuitiven Sicherheitseigenschaft „Die Reiseagentur erfährt niemals die Kreditkartendaten des Nutzers“.

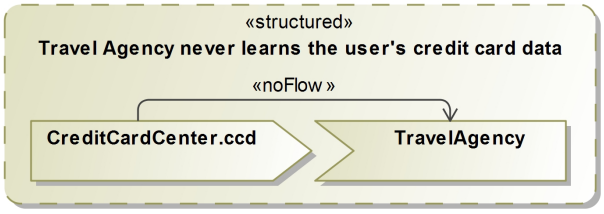


Abbildung 15: Verbotener Informationsfluss

ZUSAMMENFASSEN VON QUELLEN UND SENKEN Die Eigenschaft kann auch auf alle in der Anwendung möglichen Quellen bzw. Senken verweisen, die in der strukturierten Aktivität nicht explizit modelliert wurden. Diese werden mit einer *Accept Event Action* zusammengefasst, die mit dem Stereotyp «`otherSources`» bzw. «`otherSinks`» markiert wird. Dies ist beispielsweise dann nützlich, wenn der Informationsfluss zu allen bis auf eine Senke verboten werden soll: die Quelle wird durch einen mit «`noFlow`» markierten Kontrollfluss mit einer «`otherSinks`»-Senke verbunden, und erhält zudem einen «`allowedFlow`»-Kontrollfluss zur konkreten Senke, zu der der Fluss erlaubt ist.

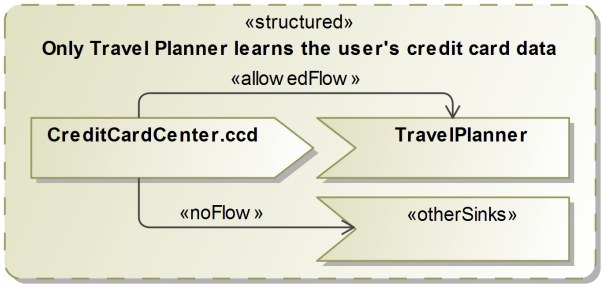


Abbildung 16: Verbotener Informationsfluss mit Ausnahmen

Abbildung 16 zeigt eine Eigenschaft der Travel Planner-Fallstudie, die den Informationsfluss der Kreditkartendaten, die im Attribut *ccd* der Komponente *CreditCardCenter* gespeichert sind, nur zur Komponente *TravelPlanner* erlaubt. Sie ist von der modellierten Anwendung nicht erfüllt.

STANDARDSPEZIFIKATION Die Informationsflüsse von und zu plattformspezifischen Quellen und Senken ist standardmäßig verboten, und muss durch «`allowedFlow`»-Kanten explizit erlaubt werden. Nur der Zugriff auf das Netzwerk durch modellierte Komponentenkommunikation ist standardmäßig erlaubt, und muss explizit durch eine «`noFlow`»-Kante zu *PredefinedSinks.NETWORK* bzw. von *PredefinedSources.NETWORK* verboten werden.

Der standardmäßige Verbot hat eine Reihe von Vorteilen: die Lesbarkeit der Informationsflusseigenschaften wird erhöht, da ihre An-

zahl überschaubar bleibt, der Modellierungsaufwand bleibt gering, und die Anwendung ist bezüglich des Informationsflusses standardmäßig von ihrer Umgebung isoliert. Der Benutzer bei einer mit MODELFLOW modellierten Anwendung also davon ausgehen, dass diese nicht auf anwendungsunabhängige Informationen wie seinen Standort oder den Inhalt der SD-Karte zugreift, wenn dies nicht explizit angegeben wird.

12.2.2 *Deklassifikationseigenschaften*

Eigenschaften bei einer transitiven Interferenz-Relation sind in realistischen Anwendungen oft zu einschränkend. Manchmal möchte man den Informationsfluss erlaubten, wenn die Information zuvor anonymisiert, gefiltert, oder die Informationsfreigabe explizit vom Benutzer bestätigt worden ist. Dies entspricht einer Sicherheitspolicy mit einer intransitiven Interferenz-Relation (*intransitive noninterference*), die die Deklassifikation von Information erlaubt.

INTRANSITIVE NICHTINTERFERENZ-EIGENSCHAFTEN

In MODELFLOW ist es möglich, Informationsflusseigenschaften zu modellieren, die den Informationsfluss zwischen einer Quelle und Senke verbieten, falls er nicht durch eine Deklassifikationsmethode stattfindet. Dafür modelliert man wie in Unterabschnitt 12.2.1 beschrieben die gewünschte Quelle und Senke, und verbietet den „direkten“ Informationsfluss (d.h., den Informationsfluss, der nicht durch eine Deklassifikationsmethode stattfindet) zwischen den beiden mittels einer mit «`noFlow`» markierter Kontrollflusskante. Zusätzlich dazu modelliert man eine Deklassifikationsmethode als einen Aktivitätsknoten, die den Namen dieser Funktion enthält (dabei muss es sich um eine Klassenoperation handeln, die mit dem Stereotyp «`declassify`» markiert ist, optional mit dem vorangestellten Namen dieser Klasse). Dieser Knoten muss mit dem Stereotyp «`via`» vermerkt sein, dessen Tag *filter* festlegt, ob es sich dabei um eine bewiesen sichere Filter- bzw. Anonymisierungsfunktion handelt (vgl. Unterunterabschnitt 15.2.2.2) Durch das Verbinden der Quelle mit dem Deklassifikationsknoten, sowie des Knotens mit der Senke durch mit «`allowedFlow`» markierten Kontrollflüssen, legt der Modellierer fest, dass der Informationsfluss von Quelle zur Senke über diese Funktion (und nur über diese Funktion) erlaubt ist.

Abbildung 17 zeigt eine Eigenschaft der Travel Planner-Fallstudie, die besagt, dass die Benutzerkreditkarten, die im Attribut *ccd* der Komponente *CreditCardCenter* nicht an die Fluglinie (*Airline*) fließen dürfen, außer, sie werden zuvor durch den Aufruf der Methode *declassifyCCD* deklassifiziert.

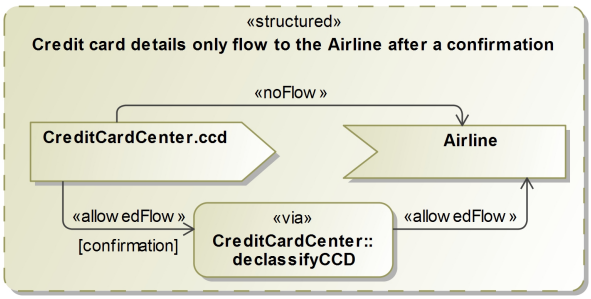


Abbildung 17: Erlaubter Informationsfluss nach Bestätigung und Deklassifikation

UMSTÄNDE DER DEKLASSIFIKATION Es ist oft nicht ausreichend, nur die Methode festzulegen, die die geheimen Daten deklassifizieren soll. In MODELFLOW ist es daher möglich, auch die Umstände, bei welchen die Deklassifikation stattfinden darf, einzuschränken. Insbesondere kann man ausdrücken, dass die Deklassifikation nur nach einer expliziten Bestätigung durch den Nutzer ausgeführt werden darf, die im Sequenzdiagramm als eine *ConfirmRelease*-Nachricht an die vordefinierte Benutzerkomponente modelliert wird. Dies entspricht der *when*-Dimension der „relativen" Klasse der Deklassifikation, wie sie von Sabelfeld et al. definiert wurde [81]; durch die Eigenschaft wird relativ zur Ausführung anderer Aktionen im System festgelegt, *wann* die Deklassifikation stattfinden darf.

In der Informationsflusseigenschaft wird dies durch einen UML-*Guard* festgelegt, der das Schlüsselwort *confirmation* enthält, und an den `«allowedFlow»`-Kontrollfluss von der Quelle zur Deklassifikationsaktion angebracht wird.

So drückt die Eigenschaft in Abbildung 17 aus, dass die Kreditkartendaten des Nutzers nur nach seiner expliziten Bestätigung für die Fluglinie freigegeben werden dürfen.

INITIATOR DER DEKLASSIFIKATION Zusätzlich zur *when*-Dimension lässt sich auch die *where*-Dimension der Deklassifikation mit MODELFLOW festlegen. Diese drückt nach Sabelfeld et al. aus, *wo* im System die Deklassifikation stattfinden darf, und kann diesen Ort auf vertrauenswürdige Teile des Systems beschränken [81].

Die Deklassifikationseigenschaft kann explizit die Komponente spezifizieren, die diese Deklassifikation vornimmt, indem der Name der Komponente dem Namen der Deklassifikationsmethode vorangestellt wird (getrennt durch zwei Doppelpunkte), die die Methode deklariert und aufruft.

Dadurch wird für den Benutzer klargestellt, in welcher App bzw. in welchem Webservice die Deklassifikation stattfindet. So kann er sich bei der Eigenschaft in Abbildung 17 sicher sein, dass seine Kreditkartendaten lokal auf seinem Gerät deklassifiziert werden, bevor sie

an die Fluglinie verschickt werden. Diese Eigenschaft kann schon bei der Modellvalidierung geprüft werden, indem der Aufrufer der Deklassifikation anhand der modellierten Sequenzdiagramme mit dem geforderten Initiator der Deklassifikation verglichen wird.

ERGEBNIS DER DEKLASSIFIKATION Auch die *what*-Dimension der Deklassifikation lässt sich in IFlow ausdrücken. Nach [81] wird dabei spezifiziert, *welche* bzw. *wie viel* Information durch die Deklassifikation freigegeben wird.

Die Deklassifikationseigenschaft kann mit dem *Tag filter* des Stereotyps «`via`» festlegen, dass es sich dabei um eine Filter- bzw. Anonymisierungsfunktion handelt. Dies besagt intuitiv, dass ein öffentlicher Beobachter der Ausgabe der Funktion vernachlässigbar wenig über die Eingabe lernt: die Information wird somit ausreichend *gefiltert* bzw. *anonymisiert*, bevor sie freigegeben wird.

Handelt es sich bei der Deklassifikationsfunktion beispielsweise um eine, die die Distanz zwischen einer Liste von geographischen Positionen berechnet, so kann sich der Benutzer sicher sein, dass die dadurch freigegebenen Daten (nahezu) keinen Rückschluss auf diese Positionen zulassen (vgl. Kapitel 21).

12.2.3 *Plattformspezifische Eigenschaften*

Aktuelle mobile Geräte stellen eine Reihe von anwendungsunabhängigen Informationsquellen und -senken zur Verfügung. Dazu zählen Sensoren (wie etwa Kameras, Mikrophone, oder GPS-Empfänger), Dateisysteme (wie etwa der interne Speicher bzw. SD-Karten) und viel mehr. Diese werden bei MODELFLOW als Attribute der vordefinierten Klassen *PredefinedSources* und *PredefinedSinks* repräsentiert (vgl. Abbildung 18), und können als Quelle bzw. Senke in den modellierten Informationsflusseigenschaften vorkommen (siehe Tabelle 7 für Syntax).

Der Zugriff auf solche Quellen und Senken geschieht durch manuelle bzw. vordefinierte Methoden; die entsprechende Klassenoperation muss dazu im Klassendiagramm mit dem Stereotyp «`uses`» markiert werden. Mit dem Tag *reads* dieses Stereotyps kann der Modellierer eine Liste von vordefinierten Quellen angeben, auf die die Methode lesend zugreift, indem dem Tag die entsprechenden Attribute der Klasse *PredefinedSources* zugewiesen werden. Mit dem Tag *writes* wird dagegen eine Liste von vordefinierten Senken angegeben, auf die die Methode schreibend zugreift. Dazu wird dem Tag mit einer Liste der Attribute der *PredefinedSinks*-Klasse befüllt. Verweist eine Informationsflusseigenschaft auf eine vordefinierte Quelle, so werden damit die Rückgabewerte aller spezifizierten Operationen impliziert, die auf diese Quelle lesend zugreifen. Verweist die Eigenschaft

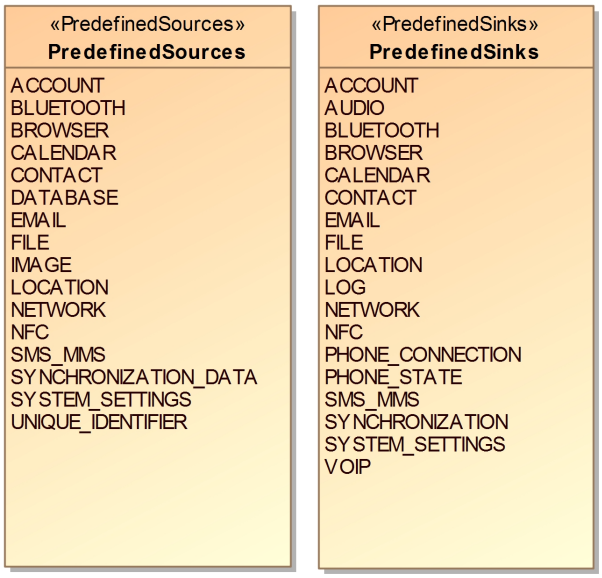


Abbildung 18: Vordefinierte, plattformspezifische Quellen und Senken

auf eine vordefinierte Senke, so sind damit die Eingabeparameter der Operationen gemeint, die auf diese Senke schreibend zugreifen.

Handelt es sich um eine vordefinierte Methode, so stellt der anwendungsunabhängige Code des IFlow-Frameworks sicher, dass auf keine weiteren Quellen und Senken zugegriffen wird. Eine manuelle Methode muss dazu auf Codeebene auf API-Zugriffe hin analysiert werden (vgl. Unterabschnitt 16.2.3).

12.2.4 *Informationsfluss zwischen Anwendungsnutzern*

Eine mit MODELFLOW modellierte Anwendung kann Webservices enthalten, die aus dem Internet erreichbar sein müssen, und von mehreren Anwendern gleichzeitig genutzt werden können. Dies gilt nicht nur für legitime Benutzer der Anwendung, die mit Hilfe der modellierten Apps mit den Webservices derselben Anwendung kommunizieren, sondern auch für Angreifer, die sich nicht an das modellierte Anwendungsverhalten halten müssen. Da dies in der Realität zu unerwünschten Informationsflüssen führen kann, selbst wenn es keine Informationsflussverletzungen *innerhalb* der modellierten Anwendung mit nur einem Nutzer gibt, muss dies besonders berücksichtigt, und die entsprechenden Garantien explizit modelliert werden.

In MODELFLOW ist es möglich, Eigenschaften über den Informationsfluss zwischen verschiedenen Nutzern der Anwendung auszudrücken. Als Nutzer ist dabei der Besitzer eines mobilen Geräts bezeichnet, auf dem eine oder mehrere Apps der betrachteten IFlow-

Anwendung installiert ist, bzw. ein Angreifer, der mit Webservices dieser Anwendung interagiert.

Da in der Realität mehrere Nutzer miteinander kooperieren, oder sogar selbst als Angreifer agieren können, um der Anwendung die Informationen eines bestimmten legitimen Anwenders zu entwenden, werden sie in MODELFLOW als eine Einheit betrachtet, und von einem konkreten, legitimen Nutzer unterschieden. Diese Gruppe von Nutzern und Angreifern kann in einer Informationsflusseigenschaft als eine namenlose Quelle bzw. Senke modelliert werden, die mit dem Stereotyp «`otherUser`» markiert ist, während alle anderen Quellen und Senken sich auf Informationen beziehen, die für den legitimen Benutzer zugänglich sind (wie etwa der Speicher seines eigenen mobilen Geräts, oder Webservice-Daten, für die er die Zugriffsberechtigung hat). Solche Nutzer können Daten untereinander austauschen, sowie die öffentlichen Schnittstellen der modellierten Webservices in beliebiger (und somit auch modellierter) Reihenfolge und mit beliebigen Daten aufrufen. Hierbei wird angenommen, dass sie die Authentifikationsdaten des legitimen Nutzers nicht kennen, die für den Zugriff auf entsprechend zugriffsgeschützte Daten notwendig sind, die von Webservices verwaltet werden (siehe Abschnitt 10.4 für Modellierungsrichtlinien solcher zugriffsgeschützten Daten), sowie keinen Zugriff auf das mobile Gerät des legitimen Nutzers haben (vgl. Unterabschnitt 17.3.2 für Diskussion der Gerät- und Plattformsicherheit).

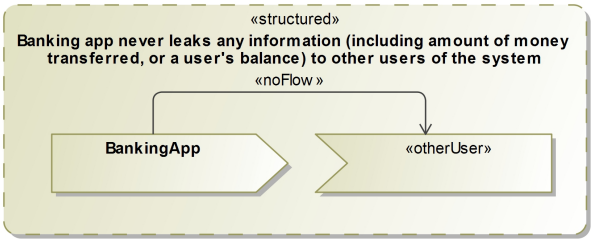


Abbildung 19: Verbotener Informationsfluss der von *BankingApp* verwalteten Daten eines Nutzers zu anderen Nutzern des Systems

Banking ist eine simple Bankenanwendung (siehe Kapitel 20 für die Erläuterung und das Anwendungsmodell der Fallstudie), die es einem Benutzer erlaubt, mit Hilfe der mobilen *BankingApp*-Anwendung sein Online-Konto mit Punkten aufzuladen (Abbildung 30), Punkte abzubuchen (Abbildung 31), oder sein Kontostand abzufragen (Abbildung 32). Die Nutzerkonten werden durch einen *Bank*-Webservice verwaltet, der den Zugriff auf ein Konto nur nach einer erfolgreichen Authentifikation erlaubt (für Modellierung der Benutzerauthentifikation siehe Abschnitt 10.4).

Abbildung 19 zeigt eine mit MODELFLOW modellierte Eigenschaft, die besagt, dass die von *BankingApp* verwalteten Informationen wie

Überweisungsbeträge, Authentifikationsdaten, oder der Kontostand eines konkreten Nutzers nicht zu anderen Nutzern des Systems fließen dürfen. Diese Eigenschaft ist von der modellierten Anwendung erfüllt.

Abbildung 33 illustriert den Programmablauf zur Abfrage des Kontostandes in einer Variation der *Banking*-Anwendung, für die die Eigenschaft **nicht** erfüllt ist. Darin weist der Modellierer den Kontostand eines Kunden dem nicht zugriffsgeschützten Service-Attribut *balance* statt einer lokalen Variable zu. Ruft ein Angreifer (bzw. ein Kunde, der die falschen Authentifikationsdaten angibt) dieselbe Service-Schnittstelle auf, so erfährt er den Kontostand des letzten Aufrufers als Inhalt der Antwortnachricht *RetBalance*.

12.2.5 *Konsistenz und Widerspruchsfreiheit*

Bei der automatischen Modelltransformation wird sichergestellt, dass die modellierten Informationsflusseigenschaften untereinander widerspruchsfrei sind, sowie mit der modellierten Sicherheitspolicy übereinstimmen.

Die Widerspruchsfreiheit wird garantiert, indem aus allen modellierten und Standard-Eigenschaften eine Informationsflusstabelle abgeleitet wird; die Spalten entsprechen dabei allen modellierten und vordefinierten Informationsflusssenken, während die Zeilen alle modellierten und vordefinierten Informationsflussquellen repräsentieren. Die Tabelleneinträge spezifizieren, ob der Informationsfluss zwischen der Quelle und Senke verboten, erlaubt, oder nur nach Deklassifikation erlaubt sind. Die Tabelle wird zunächst durch implizierte Standardeigenschaften gefüllt (siehe Abbildung 12.2.1), wonach einzelne Einträge durch modellierte Eigenschaften überschrieben werden. Soll ein bereits überschriebener Eintrag erneut durch einen inkompatiblen Eintrag überschrieben werden, so gibt es zwischen den modellierten Informationsflusseigenschaften einen Widerspruch, der vom Modellierer aufgelöst werden muss.

Zudem wird bei der Modelltransformation validiert, dass die modellierten Informationsflusseigenschaften mit der Sicherheitspolicy übereinstimmen, falls eine Sicherheitspolicy spezifiziert worden ist. Dazu wird bei transitive Nichtinterferenz-Eigenschaften geprüft, dass wenn der Fluss zwischen einer konkreten Quelle q und Senke s verboten ist, sich auch ihre Domänen nicht interferieren dürfen:

$$\mathrm{dom}(q) \not\rightsquigarrow \mathrm{dom}(s) \tag{1}$$

Da dadurch auch Flüsse über Deklassifikationsmethoden und somit Deklassifikationsdomänen verboten sind, muss die Nichtinterferenzrelation in Gleichung 1 mit einer Variante der modellierten Policy

übereinstimmen, bei der alle intransitiven Kanten transitiv gemacht wurden.

Bei intransitive Nichtinterferenz-Eigenschaften, die den Fluss von einer Quelle q zu Senke s nur über die Deklassifikationsfunktion decl erlauben, muss für $v, w \in D$ gelten:

Die zusätzlichen Domänen v, w werden benötigt, da die Kanten von und zu $\mathrm{dom}(\mathrm{decl})$ intransitiv sind

$$\begin{aligned} \mathrm{dom}(q) &\not\rightsquigarrow \mathrm{dom}(s) \wedge \\ \mathrm{dom}(q) \rightsquigarrow v &\rightsquigarrow \mathrm{dom}(\mathrm{decl}) \wedge \\ \mathrm{dom}(\mathrm{decl}) \rightsquigarrow w &\rightsquigarrow \mathrm{dom}(s) \end{aligned} \tag{2}$$

Da auch nur Informationsflüsse über die konkrete Deklassifikationsfunktion decl erlaubt sind, und Flüsse über weitere Deklassifikationsfunktionen mit einer anderen Deklassifikationsdomäne verboten sind, muss die Interferenz-Relation in Gleichung 2 mit einer Variante der modellierten Policy übereinstimmen, bei der die intransitiven Kanten (außer die von und zu $\mathrm{dom}(\mathrm{decl})$) transitiv gemacht wurden.

Da die modellierte Sicherheitspolicy immer restriktiver als jede solcher Varianten ist, ist die Anwendung auch immer dann bezüglich der modellierten Eigenschaften sicher, wenn sie die modellierte Sicherheitspolicy nicht verletzt, und die Eigenschaften in keinem Widerspruch dazu stehen. Dadurch ist es auch möglich, dass die Anwendung eine konkrete Sicherheitseigenschaft garantiert, aber die modellierte Sicherheitspolicy nicht erfüllt.

Ist der Informationsfluss zwischen einer plattformspezifischen Quelle bzw. Senke und einer Anwendungskomponente verboten, so kann bereits bei der Modelltransformation validiert werden, ob diese Komponente Methoden aufruft, die Zugriff auf diese Quelle bzw. Senke hat. Ist dies der Fall, so ist die modellierte Eigenschaft nicht erfüllt.

Schlägt die Validierung fehl, so muss der Modellierer die Anwendung bzw. ihre Informationsflusseigenschaften oder Sicherheitspolicy entsprechend abändern.

13 VERWANDTE ARBEITEN

MODELFLOW ist an die Modellierungssprache von SecureMDD [65] angelehnt, und nutzt ein Dialekt dessen domänenspezifischen Sprache MEL (vgl. Abschnitt 11.1), sowie dessen Komponentenmodell (vgl. Abschnitt 10.2). SecureMDD modelliert das komplette Verhalten der sicherheitskritischen Anwendungen, erlaubt jedoch die Integration externer Webservices [12]. Der Fokus von SecureMDD liegt auf anwendungsspezifischen Sicherheitseigenschaften von Protokollen, die mit OCL spezifiziert werden, während IFlow die intuitive Modellierung von Informationsflusseigenschaften ermöglicht.

Heldal et al. [40, 41] stellen ein UML-Profil vor, der das dezentrale Annotationsmodell (*decentralized label model, DLM* [69]) zur Informationsflussspezifikation abbildet. DLM ist die Grundlage für die Java-Erweiterung *Jif* zur Informationsflusskontrolle [68]; durch die Annotation von UML-Elementen mit Jif-Labeln ist es möglich, Jif-Code zu generieren, der mit dem Jif-Compiler und -Laufzeitumgebung auf Informationsflussverletzungen geprüft werden kann. Jedoch beschreibt [41] solche Codegenerierung als zukünftige Arbeit, während [36, 49, 75] zeigen, dass Informationsflussannotationen im Stil des DLM hochgradig nichttrivial sind.

UMLSec [46] ist ein UML-Profil zur Modellierung sicherer Systeme. Der Fokus liegt auf kryptographischer Sicherheit, das Profil unterstützt jedoch die simplen Informationsflussannotationen wie «`no down-flow`», sowie die Sicherheitsdomäne *{high}*, womit der Fluss geheimer Information zu einem öffentlichen Beobachter verboten werden kann. Dies entspricht einer transitiven Sicherheitspolicy in IFlow.

Der Modell-getriebene Ansatz zur Informationsflusskontrolle von Seehusen [84, 85] spezifiziert verbotene Informationsflüsse mit UML-Sequenzdiagrammen, beschränkt sich dabei jedoch auf transitive Policies.

Auch der UML-basierte Ansatz von Alghathbar et al. [1] betrachtet nur transitive Informationsflüsse, modelliert diese jedoch nicht mit UML, sondern spezifiziert sie mit Horn-Formeln.

Teil IV

VERIFIKATION UND CODEANALYSE

14 ÜBERBLICK

Das Ziel des IFlow-Ansatzes ist die Entwicklung von informationsflusssicheren Anwendungen. Dies wird durch die Anwendung verschiedener Modellierungs-, Analyse-, und Verifikationstechniken ermöglicht.

Der Fokus der vorliegenden Arbeit liegt auf den automatisierten Analysetechniken. Diese ermöglichen es dem Entwickler, den Großteil der in IFlow formalisierbaren Eigenschaften für seine Anwendung zu garantieren, ohne dass Kenntnis von formalen Methoden vorausgesetzt werden muss.

Die Beschreibung der im Ansatz eingesetzten formalen Methoden, sowie des formalen Modells und Frameworks baut auf den im Rahmen des IFlow-Projekts publizierten Arbeiten auf [91–93]. Das formale Modell definiert die formale Semantik von MODELFLOW, und kann für Verifikation von Informationsflusseigenschaften genutzt werden, bei denen die automatischen Techniken scheitern.

Abbildung 20 zeigt eine Übersicht über die in IFlow für Analyse und Verifikation benötigten Artefakte und Tools. Aus dem UML-Modell (1) wird eine interne Repräsentation des MEL*-Modells (2) erstellt, wobei die modellierten Informationsflusseigenschaften bereits während der automatischen Modelltransformationen vervollständigt und auf Konsistenz überprüft werden.

Das MEL*-Modell dient als Grundlage für das monolithische Codeskelett bestehend aus den Anwendungskomponenten sowie den Abstraktionen der Module und der Plattformen (3, 4), und kann anhand der generierten Analyseverpflichtungen (8) automatisch auf Informationsflussverletzungen geprüft werden.

Das Codeskelett wird anschließend in einzelne Anwendungskomponentpakete verteilt und mit den konkreten Implementierungen der Module sowie der Plattformframeworks verknüpft (5), um als mobile Apps und Webservices kompiliert zu werden. Der händisch implementierte Code wird hierbei im Kontext der App bzw. des Webservices anhand der generierten Checkverpflichtungen (9) auf Zugriffsverletzungen geprüft, um bei der Analyse bzw. Verifikation nicht berücksichtigte Informationsflüsse auszuschließen.

Das aus dem MODELFLOW-Modell generierte formale Modell (6) baut auf einer abstrakten Zustandsmaschine und algebraischen Spezifikationen auf, die das Verhalten und die Informationsflüsse des Codeskeletts abbilden. Es kann verwendet werden, um die aus den

modellierten Informationsflusseigenschaften generierte sowie manuell spezifizierten Beweisverpflichtungen (7) zu verifizieren.

Im Kontext der prototypischen, konkreten Implementierung des Ansatzes werden für die Analyse und Verifikation das Eclipse Modeling Framework [90] (2), Joana [30] (8), Soot [57] (9), und KIV [3] (7) eingesetzt.

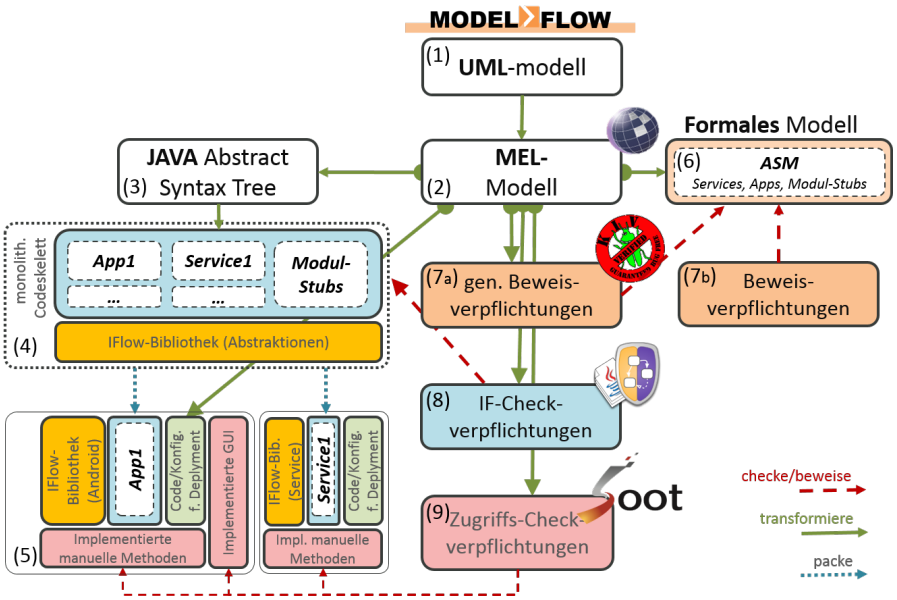


Abbildung 20: Die in IFlow für Analyse und Verifikation benötigten Artefakte und Tools

Im Folgenden werden die eingesetzten Techniken beschrieben, die je nach Klasse der modellierten Eigenschaft verwendet werden können, um die Infromationsflusssicherheit der Anwendung zu garantieren:

- Eigenschaften bei transitiver Nichtinterferenz:
  - automatische, PDG-basierte Informationsflussanalyse (8), siehe Unterunterabschnitt 16.2.2.1
  - (optional bei falsch positiven Ergebnissen der automatischen Analyse) formale Verifikation der transitiven Nichtinterferenz (7a), siehe Unterunterabschnitt 15.2.2.1 und Unterunterabschnitt 15.2.2.2
- Eigenschaften bei intransitiver Nichtinterferenz
  - (bei bewiesen sicheren Deklassifikationsmethoden) automatische, PDG-basierte Informationsflussanalyse des (angepassten) Codeskeletts (8), siehe Unterunterabschnitt 16.2.2.2
  - formale Verifikation der intransitiven Nichtinterferenz (7a), siehe Unterunterabschnitt 15.2.2.2

  - (optional für benutzerspezifizierte, komponentenlokale Eigenschaften) automatische Informationsflussanalyse von instrumentiertem Android-Code (8), siehe Unterunterabschnitt 16.2.4.3

- Umstände der Deklassifikation (*wann* Deklassifikation stattfinden darf)
  - formale Verifikation der Abläufe der abstrakten Zustandsmaschine (7a), siehe Unterunterabschnitt 15.2.2.2

- Initiator der Deklassifikation (*wo* Deklassifikation stattfinden darf)
  - automatische Validierung des Anwendungsmodells (2), siehe Unterabschnitt 12.2.5

- Deklassifikationseigenschaften komplexer Filterfunktionen
  - formale Verifikation des abstrakten Programms der Funktion (7b), siehe Unterunterabschnitt 15.2.2.2

- Plattform-spezifische Zugriffsbeschränkungen
  - statische Kontrollflussanalyse des Bytecodes (9), siehe Unterabschnitt 16.2.3

Abschließend umreißt Kapitel 17 das Verhältnis zwischen dem formalen Modell und dem Codeskelett, beschreibt im Detail die Refinementbeziehung zwischen dem Codeskelett und der finalen Anwendung, und diskutiert den Kontext, in dem die finale Anwendung ausgeführt werden darf bzw. muss.

15 VERIFIKATION

Der IFlow-Ansatz ermöglicht die interaktive Verifikation der unterstützten Informationsflusseigenschaften. Hierzu wird automatisch ein formales Modell der mit MODELFLOW modellierten Anwendung generiert, das mit KIV [3] eingelesen und bzgl. der spezifizierten Eigenschaften verifiziert werden kann. Dieses Kapitel erläutert die formalen Grundlagen des IFlow-Ansatzes.

In Abschnitt 15.1 werden zunächst die Grundlagen der modellbasierten Informationsflusstheorie von Rushby vorgestellt, auf der das formale Modell von IFlow aufbaut. In Abschnitt 15.2 wird erläutert, wie formale Verifikation in IFlow eingesetzt wird. Dazu zeigt Unterabschnitt 15.2.1 das formale Modell von IFlow und dessen Beziehung zur Informationsflusstheorie von Rushby. Unterunterabschnitt 15.2.1.2 erklärt das Prinzip der Deklassifikation, das für viele Anwendungen notwendig ist, während Unterunterabschnitt 15.2.2.1 auf die Unwinding-Theoreme von Rushby und van der Meyden eingeht. Abschließend skizziert Unterunterabschnitt 15.2.2.2 das formale Framework zur Verifikation der Deklassifikationsfunktionen.

15.1 NICHTINTERFERENZ BEI RUSHBY

In der Vergangenheit wurde eine Reihe von Informationsflusseigenschaften vorgeschlagen, um Sicherheitsanforderungen hinsichtlich der Vertraulichkeit von Daten oder Systemaktionen zu formalisieren [61]. Eine davon ist die *Nichtinterferenz*, die die Intuition der mit MODELFLOW modellierten Eigenschaften erfasst, und daher zur Formalisierung und Verifikation solcher Eigenschaften in IFlow verwendet wird.

Der Begriff der Nichtinterferenz (oder *Nicht-Beeinflussung*, engl. *noninterference*) wurde ursprünglich von Goguen und Meseguer eingeführt [31]. Sie definieren Nichtinterferenz als die Eigenschaft, die dann für ein gegebenes System gilt, wenn die Aktionen einer Gruppe von Nutzern keine Auswirkung darauf haben, was eine andere Gruppe von Nutzern sehen kann. Hierfür erhalten die Aktionen jeweils eine Sicherheitsdomäne, und die Eigenschaft wird als eine Nichtinterferenz-Relation $\not\rightsquigarrow$ zwischen diesen Domänen formalisiert. Intuitiv drückt diese Relation die Richtung aus, in der die Information innerhalb des Systems nicht fließen darf.

Rushby stellt in [80] ein Framework vor, das das die Nichtinterferenz-Eigenschaft für ein Transitionssystem formalisiert, und in IFlow verwendet wird.

15.1.0.1 *Systemmodell*

Das von Rushby betrachtete Systemmodell ist ein Transitionssystem M. Im Folgenden wird die Variante vorgestellt, die Rushby zur Formalisierung des Zugriffskontrollmechanismus verwendet. Dieses führt eine interne Struktur der Systemzustände in Form von Speicherstellen ein, die sich in unserem formalen Modell intuitiv auf Komponentenattribute und Variablen abbilden lassen (siehe Unterabschnitt 15.2.1 für Details).

Das Systemmodell besteht dabei aus folgenden Komponenten:

- Menge S von strukturierten *Systemzuständen* mit dem Ausgangszustand $s_0 \in S$
- Menge N von *Speicherstellen* (*names* bei Rushby)
- Menge V von *Speicherwerten*
- Funktion $\mathtt{contents} : S \times N \to V$, die den Wert $v \in V$ in der Speicherstelle $n \in N$ im Zustand $s \in S$ festlegt
- Menge A von *Systemaktionen*
- Menge O von *Ausgaben*
- Funktion $\mathtt{step} : S \times A \to S$, die den nächsten Systemzustand $s' \in S$ festlegt, wenn die Aktion $a \in A$ im Zustand $s \in S$ ausgeführt wurde
- Funtion $\mathtt{run} : S \times A^* \to S$, die den nächsten Systemzustand $s' \in S$ beschreibt, wenn die Aktionssequenz $\alpha \in A^*$ im Zustand $s \in S$ ausgeführt wurde
- Funktion $\mathtt{output} : S \times D \to O$, die die Ausgabe $o \in O$ festlegt, die die Domäne $d \in D$ im Zustand $s \in S$ liest

15.1.0.2 *Transitive Nichtinterferenz-Relation*

Die Nichtinterferenz-Eigenschaft drückt aus, dass ein öffentlicher Beobachter nicht die Informationen erfahren darf, die von geheimen Aktionen stammen. Um diese Aussage formalisieren zu können, definiert Rushby zudem

- die Menge D von Sicherheitsdomänen wie etwa *public* oder *secret*,
- die Funktion $\mathtt{dom} : A \to D$, die die Sicherheitsdomäne $u \in D$ der Aktion $a \in A$ festlegt,
- die Funktion $\mathtt{observe} : D \to \mathcal{P}(N)$, die festlegt, welche die Menge von Speicherstellen aus N die Domäne $d \in D$ *beobachten* kann,

- die Funktion $\mathtt{alter} : D \to \mathcal{P}(N)$, die festlegt, welche die Menge von Speicherstellen aus N die Domäne $d \in D$ *verändern* kann,
- die reflexive Interferenzrelation $\rightsquigarrow$: $D \times D$, die mit $u \rightsquigarrow v$ den Informationsfluss von Sicherheitsdomäne $u \in D$ zur Domäne $v \in D$ erlaubt,
- ihre Komplementrelation $\not\rightsquigarrow = (D \times D) \setminus \rightsquigarrow$,
- die Äquivalenzrelation $\approx_d$ auf S, die besagt, dass zwei Zustände $s_1, s_2 \in S$ für die Domäne $d \in D$ äquivalent sind, wenn alle Werte der Speicherbereiche, die d beobachten kann, gleich sind: $s_1 \approx_d s_2 :\Leftrightarrow \forall l \in \mathtt{observe}(d) : s_1(l) = s_2(l)$
- die Funktion $\mathtt{purge} : A^* \times D \to A^*$, die für eine Domäne $v \in D$ aus der Aktionssequenz $\alpha \in A^*$ alle Aktionen $a \in A$ löscht, für die gilt $\mathtt{dom}(a) \not\rightsquigarrow v$.

Nach Rushby gilt die Aussage $u \not\rightsquigarrow v$ dann, wenn der öffentliche Beobachter mit der Domäne v nicht feststellen kann, ob in einem Systemablauf α geheime Aktionen mit der Domäne u ausgeführt wurden. Werden also aus α mit der Funktion $\mathtt{purge}(\alpha, v)$ alle geheimen Aktionen $a \in A$ mit $\mathtt{dom}(a) \not\rightsquigarrow v$ gelöscht, so muss gelten, dass der Benutzer nicht anhand der für ihn sichtbaren Ausgabe zwischen α und $\mathtt{purge}(\alpha, v)$ unterscheiden kann:

$$\mathtt{output}(\mathtt{run}(s_0, \alpha), v) = \mathtt{output}(\mathtt{run}(s_0, \mathtt{purge}(\alpha, v)), v) \tag{3}$$

Das System gilt folglich als sicher, wenn Gleichung 3 erfüllt ist, d.h., wenn öffentliche Beobachtungen nicht von geheimen Aktionen abhängen. Dies kann mit Hilfe der *Unwinding-Bedingungen* bewiesen werden, die in Unterunterabschnitt 15.2.2.1 näher erläutert werden, wofür die Funktionen *observe* und *alter* benötigt werden.

15.1.0.3 *Intransitive Nichtinterferenz-Relation*

Die oben beschriebene Definition bezieht sich auf die *transitive* Version der Nichtinterferenz-Relation, wie sie auch von Goguen und Meseguer [31] formalisiert wurde. Für viele Anwendungen ist sie jedoch zu einschränkend: implementiert etwa das System eine Funktion zur Überprüfung des Benutzerpassworts $\mathtt{checkPass}(\mathtt{pass})$, so hängt das Ergebnis der Funktion, das für einen öffentlichen Beobachter einsehbar sein muss, von dem geheimen Benutzerpasswort ab. Soll also geheime Information mit der Domäne u durch eine Aktion a mit $\mathtt{dom}(a) = w$ zur Domäne v *deklassifiziert*, d.h., öffentlicher gemacht werden können, aber der direkte Fluss der geheimen Information zur Domäne v verboten sein, so wird die *intransitive* Variante der Nichtinterferenz-Relation benötigt (vgl. Gleichung 4).

$$u \rightsquigarrow w \wedge w \rightsquigarrow v \wedge u \not\rightsquigarrow v \tag{4}$$

Rushby formalisiert diese intransitive Nichtinterferenz-Eigenschaft, indem er in Gleichung 3 die Funktion purge so modifiziert, dass diese nur die geheime Aktionen von u löscht, die nicht von Aktionen von w gefolgt werden.

15.2 VERIFIKATION IN IFLOW

Um die Informationsflusseigenschaften einer IFlow-Anwendung interaktiv verifizieren zu können, werden aus dem Anwendungsmodell ein formales Modell sowie die Beweisverpflichtungen generiert. Anschließend kann das formale Modell mit KIV [3] eingelesen und bezüglich der generierten Beweisverpflichtungen verifiziert werden.

15.2.1 *Formales Modell einer IFlow-Anwendung*

Dieser Abschnitt gibt einen Überblick über das formale Modell einer IFlow-Anwendung [29], welches die formale Semantik von MODELFLOW definiert. Die Elemente des formalen Modells werden am Beispiel der *TravelPlanner*-Fallstudie erläutert (siehe Abschnitt 10.5 für die statische und Abschnitt 11.4 für die dynamische Sicht auf die Anwendung).

Siehe Abschnitt 5.2 für Details zu den Transformationen

Das Modell wird aus dem MEL*-Zwischenmodell abgeleitet, aus dem auch das Codeskelett generiert wird. Unterabschnitt 16.2.1 beschreibt die Struktur des Codeskeletts, während Abschnitt 17.1 die Refinement-Beziehung zwischen dem formalen Modell und dem Codeskelett diskutiert.

DATENTYPEN Primitive MODELFLOW-Datentypen werden im formalen Modell wie gefolgt abgebildet: *Integer* entspricht einer unbegrenzten Ganzzahl, *String* einer algebraisch spezifizierten Zeichenfolge, *double* einer endlichen Dezimalzahl, und *Boolean* einem Boolean. Komplexe Datentypen werden dabei durch algebraische Spezifikationen abgebildet.

Listing 1 zeigt die Spezifikation der komplexen Datentypen der *Travel-Planner*-Fallstudie: *CreditCardDetails*, *FlightOffer*, und *RequestData*.

Siehe Abbildung 9 für das vollständige Klassendiagramm von TravelPlanner

KOMPONENTEN Anwendungskomponenten, die vordefinierte und manuelle grafische Oberflächen, sowie externe Apps und Services werden auf Instanzen der Sorte *Agent* abgebildet (vgl. Listing 2). Da sie mehrfach instantiiert werden können, erhalten sie zudem eine natürliche Zahl als eine eindeutige Identifikationsnummer.

Eine Komponente enthält zudem die folgenden Speicherstellen, auf deren Wert mit der Funktiom s zugegriffen werden kann:

```
1 data specification
2                   using string−ops
3
4         CreditCardDetails = mkCreditCardDetails(. .name :
              string; . .number : string; . .expiration : string);
5         FlightOffer = mkFlightOffer(. .id : int; . .airline : string);
6         RequestData = mkRequestData(. .date : string);
```

Listing 1: Spezifikation von komplexen Datentypen der *Travel-Planner*-Fallstudie

```
1 data specification
2                   using string−data, nat
3
4 Agent =  mkAirline(. .id : nat) with isAirline
5         | mkCreditCardCenter(. .id : nat) with
              isCreditCardCenter
6         | mkTravelAgency(. .id : nat) with isTravelAgency
7         | mkTravelPlanner(. .id : nat) with isTravelPlanner
8         | mkUser(. .id : nat) with isUser
```

Listing 2: Spezifikation von Komponenten der *Travel-Planner*-Fallstudie

1. **Komponentenattribute** werden als Tupel aus dem Namen des Attributs sowie einer Agenteninstanz spezifiziert. So entspricht etwa das Attribut *ccd* der Komponente *CreditCardCenter* aus der Fallstudie *TravelPlanner* dem Tupel $ccd \times CreditCardCenter(n)$.

2. Zudem erhält jede Komponente für jede empfangene Nachricht eine eigene **Mailbox**-Variable, die ebenfalls als ein solcher Tupel spezifiziert wird. Diese Mailbox speichert eine empfangene Nachricht, bis sie von der Komponenten verarbeitet wird. So speichert etwa $Mailbox(GetFlightOffers) \times Airline(o)$ die Nachricht *GetFlightOffers* an die Singleton-Instanz der Airline-Anwendungskomponente der *TravelPlanner*-Fallstudie.

   Siehe Abbildung 12 für das vollständige Sequenzdiagramm von TravelPlanner

3. **Zustandsvariablen** werden wie auch Mailboxen nicht explizit modelliert, sondern automatisch aus dem Sequenzdiagramm abgeleitet. Sie garantieren die im Sequenzdiagramm modellierte Abfolge der Aktionen.

4. Modellierte **lokale Variablen** werden als Tupel aus dem Namen der Variable und der Agenteninstanz abgebildet.

So wird das Parameter *requestData* der Nachricht *GetFlightOffers*, die in *TravelPlanner* an die *Airline*-Komponente versandt wird, zum Tupel *requestData* × *Airline*.

KOMPONENTENVERHALTEN UND -INTERAKTION Das Systemverhalten wir im formalen Modell von IFlow durch eine ASM [13] abgebildet. Die Regeln der ASM verändern ihren Zustand, der dem Mapping s von Speicherstellen aller modellierten Komponente auf die Werte in diesen Speicherstellen entspricht. Im einfachsten Fall entspricht ein ASM-Schritt dem modellierten Verhalten einer Komponente zwischen dem Empfang einer *CallMessage* bzw. *ReplyMessage*, und dem Versand der nächsten *CallMessage* bzw. *ReplyMessage*. Die MEL*-Anweisungen wie Zuweisungen, Methodenaufrufe, und arithmetische Operationen werden dabei auf entsprechende Anweisungen eines abstrakten Programms abgebildet (für Details siehe [65]).

Eine ASM-Regel, die den Empfang einer Nachricht abbildet, überprüft, ob die dazugehörige Mailbox eine Nachricht enthält. Ist dies der Fall, wird die Mailbox geleert, und die modellierten Anweisungen werden ausgeführt. Der Versand der nächsten Nachricht entspricht dabei dem Schreiben der Nachricht in die dazugehörige Mailbox. Die ASM führt in jedem Schritt zufällig eine ihrer Regeln aus.

Eine solche ASM-Regel der *Travel-Planner*-Fallstudie für den Empfang der Nachricht *GetFlightOffer* ist in Listing 3 abgebildet. Abbildung 21 zeigt den dazugehörigen Ausschnitt aus dem vollständigen Sequenzdiagramm (vgl. Abbildung 12).

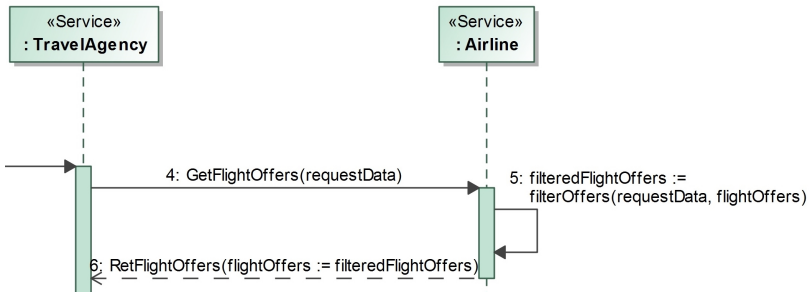


Abbildung 21: Ausschnitt aus dem Sequenzdiagramm der TravelPlanner-Fallstudie

Der Empfang der Nachricht 4 in Abbildung 21 entspricht dabei den Zeilen 3-5 in Listing 3. Zeile 3 überprüft die dazugehörige Mailbox *Mailbox(GetFlightOffers)* der *Airline*, und Zeile 4 leert diese. Zeile 5 weist den Inhalt der Nachricht automatisch der lokalen Variable *requestData* zu. Zeilen 6-8 entsprechen dem Aufruf der manuellen Methode *filterOffers* und der Zuweisung ihres Rückgabewerts an die Variable *filteresFlightOffers* in Nachricht 5 des Sequenzdiagramms. Abschließend schreiben Zeilen 9-10 diese Variable als Inhalt der Antwortnachricht *RetFlightOffers* in die dazugehörige Mailbox *Mailbox(RetFlightOffers)* der *TravelAgency*.

Siehe Unterabschnitt 11.2.2 zu Details über automatische Generierung von lokalen Variablen

```
1  getFlightOffers−Airline ≡
2      let msg = s(Mailbox(GetFlightOffers) × Airline(o)) in {
3          if (isGetFlightOffersMessage(msg)) then {
4            s(Mailbox(GetFlightOffers) × Airline(o)) := null;
5            s(requestData × Airline(o)) := msg.requestData;
6            s(filteredFlightOffers × Airline(o)) :=
7              filterOffers(s(requestData × Airline(o)),
8                           s(flightOffers × Airline(o)));
9            s(Mailbox(RetFlightOffers) × TravelAgency(o)) :=
10             RetFlightOffers(s(filteredFlightOffers × Airline(o)));
11 }}
```

Listing 3: Die ASM-Regel *getFlightOffers-Airline*

Vordefinierte und manuelle Methoden (wie *filterFlightOffers* in Abbildung 21) werden im formalen Modell als algebraische Funktionen abgebildet, deren Signatur der modellierten Methode entspricht. MEL*-Methoden werden dabei zu abstrakten Programmen übersetzt.

Wurden zwischen einer *CallMessage* und einer *ReplyMessage* Nachrichten mit verschiedenen Domänen modelliert, so werden diese in mehrere ASM-Regeln unterteilt, wobei hintereinander auszuführende Nachrichten mit derselben Domäne zu einer Regel zusammengefasst werden. Die korrekte Reihenfolge solcher Regeln wird durch das Überprüfen, Leeren, und Schreiben von automatisch generierten Zustandsvariablen garantiert (siehe auch Unterunterabschnitt 15.2.1.2).

Abgesehen von manuellen und vordefinierten Methoden werden unterspezifizierte Anwendungsmodule wie die graphische Benutzeroberfläche und externe Komponenten im formalen Modell wie im Code abstrahiert, wobei solche Abstraktionen die Informationsflüsse ihrer konkreten Implementierung überapproximieren. Details sowie Pseudocode dieser Abstraktionen wird in Unterabschnitt 17.2.2 gezeigt und im Detail diskutiert. Unterunterabschnitt 17.2.1.5 erläutert dabei die Überapproximation und Abbildung der in der finalen Anwendung möglichen Programmabläufe. Solche Abstraktionen werden als weitere Regeln der ASM abgebildet.

15.2.1.1 *Instantiierung des Frameworks von Rushby*

Um die in IFlow unterstützten Nichtinterferenz-Eigenschaften zu formalisieren, instantiiert das formale Modell von IFlow das Framework von Rushby wie gefolgt:

ZUSTÄNDE UND SPEICHERSTELLEN Die Zustände des Transitionssystems von Rushby sind in Speicherstellen strukturiert. Diese Speicherstellen werden in der ASM des formalen Modells in IFlow als (1) Komponentenattribute, (2) Mailboxen, (3) lokale Variablen, und (4)

Zustandsvariablen abgebildet. Die Funktion contents wird in IFlow durch die Funktion s abgebildet, die den Wert in einer Speicherstelle zurückgibt.

AKTIONEN Eine Aktion des Transitionssystems von Rushby entspricht in IFlow einer Regel der ASM. Im einfachsten Fall entspricht ein ASM-Schritt dem modellierten Verhalten einer Komponente zwischen dem Empfang einer *CallMessage* bzw. *ReplyMessage*, und dem Versand der nächsten *CallMessage* bzw. *ReplyMessage* (siehe Unterabschnitt 15.2.1 für Details zu ASM-Schritten). Die Menge der Domänen D sowie die Zuordnung der Domänen zu Aktionen durch die Funktion dom wird aus dem Anwendungsmodell übernommen (siehe Abschnitt 12.1 für Sicherheitsannotationen in MODELFLOW), und durch automatisch generierte Domänen erweitert (siehe Unterunterabschnitt 15.2.1.2 zur impliziten Deklassifikation).

observe UND alter Die Menge der von den Funktionen observe(d) und alter(d) zurückgegebenen Speicherstellen entspricht allen Speicherstellen, die von allen Aktionen mit der angegebenen Domäne d beobachtet bzw. verändert werden:

$$\mathtt{observe}(d) = \bigcup \{\mathtt{observe}(a) : \mathtt{dom}(a) = d\}$$
$$\mathtt{alter}(d) = \bigcup \{\mathtt{alter}(a) : \mathtt{dom}(a) = d\}$$

Die Funktionen observe(a) und alter(a) werden dabei automatisch aus dem MEL*-Code des Anwendungsmodells generiert.

output Die Funktion output(s, d) ist in IFlow fix und entspricht den Werten aller Speicherstellen, die die Domäne d beobachten kann:

$$\mathtt{output}(s, d) = \{s(l) : l \in \mathtt{observe}(d)\}$$

purge In IFlow kommt die intransitive Variante der purge-Funktion zum Einsatz (siehe Unterunterabschnitt 15.1.0.3).

15.2.1.2 *Implizite Deklassifikation*

Bei der Generierung der ASM aus dem Anwendungsmodell muss zusätzlich eine Besonderheit beachtet werden. Enthält das Anwendungsmodell zwei aufeinander folgende Aktionen a_1 und a_2 mit $\mathtt{dom}(a_1) \not\rightsquigarrow \mathtt{dom}(a_2)$, wobei a_2 keine Variablen oder Komponentenattribute liest, die a_1 schreibt, so ist die Anwendung intuitiv sicher. Dies ist beispielsweise in der Travel Planner-Fallstudie der Fall: die Nachricht 17 (*PayComission*) in Abbildung 22 liest nichts, was die die Nachricht 16 schreibt. Außer der Tatsache, dass zuvor die Aktion *processBooking* ausgeführt wurde (was auch ein Blick in das Anwendungsmodell verraten würde), erfährt der Empfänger der *PayCommission* keine weitere Information.

Siehe Abbildung 12 für das vollständige Interaktionsdiagramm und Abbildung 13 für die modellierte Interferenzrelation

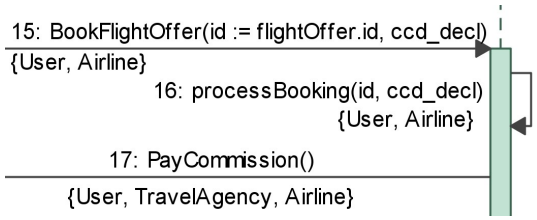


Abbildung 22: Ausschnitt aus dem Sequenzdiagramm der TravelPlanner-Fallstudie

Die ASM des generierten formalen Modells garantiert die modellierte Reihenfolge ihrer Schritte jedoch durch das Schreiben und Abfragen der Mailboxen und der zusätzlichen, automatisch generierten Zustandsvariablen (siehe Unterabschnitt 15.2.1 für Details). Ein solcher Zustand wird in Abbildung 22 von der aus Nachricht 16 generierten Aktion a_{16} gesetzt, der von der aus Nachricht 17 generierten Aktion a_{17} überprüft wird. Da die modellierte Sicherheitspolicy jedoch den Informationsfluss von $dom(a_{16})$ zu $dom(a_{17})$ verbietet, kann die Unwinding-Bedingung *AOI* (siehe Unterunterabschnitt 15.2.2.1) nicht bewiesen werden.

Um auch ein solches System als sicher garantieren zu können, ohne zusätzliche, explizite Deklassifikationsdomänen und -aktionen modellieren zu müssen, werden diese automatisch aus dem Modell berechnet. Dazu wird in Abbildung 22 zwischen a_{16} und a_{17} eine neue Aktion a_{decl} eingefügt, die eine neue Deklassifikationsdomäne $dom(a_{decl})$ mit

$$dom(a_{16}) \rightsquigarrow dom(a_{decl}) \wedge$$
$$dom(a_{decl}) \rightsquigarrow dom(a_{17}) \wedge$$
$$dom(a_{16}) \not\rightsquigarrow dom(a_{17})$$

erhält. Dadurch schreibt a_{16} zunächst den Zustand von a_{decl}, die diesen laut der neuen Interferenz-Relation lesen darf. a_{decl} schreibt ihrerseits den Zustand von a_{17}, die diesen nun ebenfalls lesen und überprüfen kann. Durch diese *implizite Deklassifikation* bildet die ASM nach wie vor die modellierte Abfolge der Nachrichten ab, und verletzt zugleich nicht die Nichtinterferenz-Relation.

15.2.2 *Formalisierung und Verifikation der IF-Eigenschaften*

15.2.2.1 *Sicherheitspolicy*

Durch die Instantiierung des Frameworks von Rushby kann der Entwickler verifizieren, dass die IFlow-Anwendung bezüglich ihrer modellierten Sicherheitspolicy (und somit bezüglich der modellierten transitiven und intransitiven Nichtinterferenz-Eigenschaften) sicher ist. Dazu werden die Sicherheitsdomänen D, ihre intransitive Interferenz-Relation $\rightsquigarrow$, sowie die Abbildung der Aktionen auf ihre Domä-

nen dom aus der mit MODELFLOW modellierten Sicherheitspolicy abgeleitet.

Siehe Abschnitt 12.1 für Details zur Modellierung der Sicherheitspolicy

Nach Rushby [80] und van der Meyden [63] ist ein System dann sicher, wenn die folgenden vier Unwinding-Bedingungen gelten:

RMA *steht für* Reference Monitor Assumption. *In einer Betriebssystemarchitektur definiert ein Referenzmonitor Anforderungen an ein Zugriffskontrollmechanismus*

- **RMA1**: $s_1 \approx_d s_2 \rightarrow \mathrm{output}(s_1, d) = \mathrm{output}(s_2, d)$

  Sind zwei Zustände für eine Domäne gleich, so ist auch die Ausgabe für diese Domäne gleich. Dies gilt in IFlow automatisch, da output nur die Werte der Speicherstellen zurückgibt, die diese Domäne beobachten kann.

- **RMA2**: $s_1 \approx_{\mathrm{dom}(a)} s_2 \wedge s_1(l) = s_2(l) \wedge l \in \mathrm{alter}(\mathrm{dom}(a)) \rightarrow \mathrm{step}(s_1, a)(l) = \mathrm{step}(s_2, a)(l)$ [63]

  Die Bedingung fordert, dass die Menge observe, durch die $\approx$ definiert ist, nicht zu klein ist [63, 91]: wenn eine Aktion eine Veränderung an einer Speicherstelle bewirken kann, dann muss die Domäne der Aktion auch die Speicherstelle beobachten können, deren Wert diese Veränderung verursacht hat. In IFlow gilt dies automatisch, da observe aus dem Modell berechnet wird.

- **RMA3**: $\mathrm{step}(s, a)(l) \neq s(l) \rightarrow l \in \mathrm{alter}(\mathrm{dom}(a))$

  Die Bedingung garantiert, dass die Menge alter korrekt gewählt wurde: verändert eine Aktion den Wert einer Speicherstelle, dann muss dies durch alter erlaubt sein. In IFlow gilt dies automatisch, da alter aus dem Modell berechnet wird.

AOI *steht für* Alter/Observe respects Inteference

- **AOI**: $\mathrm{alter}(d_1) \cap \mathrm{observe}(d_2) \neq \emptyset \rightarrow d_1 \rightsquigarrow d_2$

  Wird eine Speicherstelle von d_1 verändert, die von d_2 beobachtet wird, so muss d_1 d_2 interferieren.

Die Beweisverpflichtungen *RMA2* und *RMA3* können in großen Systemen wegen der step-Funktion sehr teuer werden. Aufgrund des modellgetriebenen Ansatzes muss in IFlow jedoch lediglich *AOI* bewiesen werden, während *RMA1-3* automatisch gelten.

15.2.2.2 *Formalisierung der Informationsflusseigenschaften*

Siehe Abschnitt 12.2 für Klassifikation und Modellierung von IF-Eigenschaften

Zusätzlich zur allgemeinen Sicherheitspolicy kann der Entwickler für jede modellierte Informationsflusseigenschaft beweisen, dass die Anwendung diese erfüllt. Da solche Eigenschaften nicht nur den Informationsfluss zwischen Aktionen, sondern auch Speicherstellen wie Komponentenattribute einschränkt, wird die neue Funktion $\mathrm{dom} : N \rightarrow D$ eingeführt:

> $\mathrm{dom} : N \rightarrow D$ ordnet einer Speicherstelle $l \in N$ die Sicherheitsdomäne $d \in D$ zu

Für jede Speicherstelle werden zusätzlich zwei neue Beweisverpflichtungen eingeführt, die an die Bell-La-Padula-Eigenschaften [7] angelehnt sind:

- **no-read-up**: $l \in observe(d) \rightarrow dom(l) \rightsquigarrow d$
- **no-write-down**: $l \in alter(d) \rightarrow d \rightsquigarrow dom(l)$

no-read-up besagt intuitiv, dass eine Aktion keine geheimeren Speicherstellen lesen darf, als ihre eigene Domäne es zulässt. *no-write-down* legt dagegen fest, dass eine Aktion keine öffentlicheren Speicherstellen schreiben darf, als ihre eigene Domäne es zulässt.

Obwohl bei der Modelltransformation sichergestellt wird, dass die modellierten Informationsflusseigenschaften nicht der definierten Sicherheitspolicy widersprechen, und sie somit auch dann gelten, wenn die Anwendung diese Sicherheitspolicy erfüllt, kann eine Eigenschaft auch gelten, wenn die Sicherheitspolicy verletzt ist. Dies liegt daran, dass jede Informationsflusseigenschaft eine eigene Variante der modellierten Sicherheitspolicy impliziert, die mehr Informationsflüsse zulässt als das Original.

Siehe Unterabschnitt 12.2.5 zur Konsistenz und Widerspruchsfreiheit der IF-Eigenschaften

TRANSITIVE NICHTINTERFERENZ-EIGENSCHAFTEN Verbietet eine transitive Informationsflusseigenschaft den Informationsfluss von einer Quelle q zur Senke s, so wird diese Eigenschaft als die folgende Beweisverpflichtung formalisiert:

$$dom(q) \not\rightsquigarrow' dom(s)$$

Da in der originalen Interferenz-Relation der direkte Fluss von q zu s zwar verboten, aber über eine Deklassifikationsfunktion $decl$ erlaubt sein könnte, würde dies nicht der Intuition der modellierten Eigenschaft entsprechen, die *jeglichen* Informationsfluss von q zu s verbietet. Daher wird eine Variante $\rightsquigarrow'$ der originalen Interferenzrelation $\rightsquigarrow$ benutzt, die ihre transitive Hülle bildet:

$$\rightsquigarrow' = \rightsquigarrow^+$$

Folgende Regeln gelten bei der Übersetzung der MEL*-Quellen und-Senken zu Speicherstellen und Aktionen des formalen Modells:

- Eine Quelle oder Senke, die ein MODELFLOW-Komponentenattribut referenziert, wird auf die entsprechende Speicherstelle abgebildet, die dieses Attribut im formalen Modell repräsentiert
- Eine Senke, die eine MODELFLOW-Komponente referenziert, wird auf die Mailboxen abgebildet, die für den Empfang der an diese Komponente gerichteten Nachrichten zuständig sind, sowie auf Attribute, an die das Ergebnis der *decrypt*-Operation zugewiesen wird

Vergleiche dazu Unterunterabschnitt 16.2.2.1 für die Übersetzung solcher Quellen und Senken zu Attributen und Parametern des Codeskeletts

- Eine Quelle, die eine MODELFLOW-Komponente referenziert, wird auf alle ihre (modellierten) Attribute, sowie auf auf die Mailboxen abgebildet, in die diese Komponente schreibt, um Nachrichten an andere Komponenten zu verschicken.
- Eine vordefinierte, plattformspezifische Senke wird auf die Aktionen abgebildet, die solche Methoden aufrufen, die auf diese Senke zugreifen (modelliert als Tag *sink* des «`uses`»-Stereotyps, vgl. Unterunterabschnitt 9.2.2.4)
- Eine vordefinierte, plattformspezifische Quelle wird auf die Aktionen abgebildet, die solche Methoden aufrufen, die auf diese Quelle zugreifen (modelliert als Tag *source* des «`uses`»-Stereotyps, vgl. Unterunterabschnitt 9.2.2.4)
- Quellen und Senken, die mit den Stereotypen «`otherSinks`» und «`otherSources`» annotiert sind, werden bereits beim Generieren der Informationsflusstabelle auf entsprechende Komponenten und ihre Attribute abgebildet (vgl. Unterabschnitt 12.2.1)
- Eine Quelle, die eine mit *„label“* beschriftete Benutzereingabe repräsentiert, wird auf die Mailbox abgebildet, in die die Nachricht reingeschrieben wird, die aus einer mit *„label“* annotierten Sequenzdiagrammnachricht generiert wurde (vgl. Unterabschnitt 11.2.3)
- Eine Quelle oder Senke, die einen anderen Systemnutzer repräsentiert, wird auf das Attribut *sl* der *OtherUser*-Klasse abgebildet, welches dessen internen Zustand repräsentiert (vgl. Unterunterabschnitt 17.2.1.5)

Die Eigenschaft gilt dann, wenn für die Speicherstellen des Systems die Eigenschaften *no-read-up* und *no-write-down* gelten, die Sicherheitspolicy aus Unterunterabschnitt 15.2.2.1 mit der neuen Interferenz-Relation $\rightsquigarrow'$ erfüllt ist, und die Beweisverpflichtung der Eigenschaft nicht im Widerspruch zu dieser Relation steht.

INTRANSITIVE NICHTINTERFERENZ-EIGENSCHAFTEN Verbietet eine intransitive Informationsflusseigenschaft jeden Informationsfluss von einer Quelle q zur Senke s (außer den Fluss durch die Deklassifikationsfunktion d), so wird diese Eigenschaft als die folgende Beweisverpflichtung formalisiert:

Der Fluss über die Domäne d wird durch die Interferenzrelation zugelassen, siehe unten

$$\mathrm{dom}(q) \not\rightsquigarrow' \mathrm{dom}(s)$$

Die originale Interferenz-Relation könnte auch Flüsse von q zu s die über andere Deklassifikationsfunktionen als d zulassen. Dies entspricht jedoch nicht der intuitiven Eigenschaft, dass *jeder* Fluss von q zu s über d stattfinden muss.

Daher wird die Variante $\rightsquigarrow'$ von $\rightsquigarrow$ benutzt, die aus der Original-Sicherheitspolicy generiert wird, in der alle intransitiven Kanten außer die von und zu $\mathrm{dom}(d)$ transitiv gemacht wurden. Hierfür werden die Kanten von und zu $\mathrm{dom}(d)$ der Menge $\rightsquigarrow$ abgezogen , aus dem Resultat wird die transitive Hülle gebildet, wonach die abgezogenen Kanten wieder hinzugefügt werden:

Die Modellrichtlinien garantieren, dass alle Kanten von und zu $\mathrm{dom}(d)$ *intransitiv sind*

$$\rightsquigarrow' = (\rightsquigarrow \setminus \bigcup_{a,b \in D} \{(a, \mathrm{dom}(d)), (\mathrm{dom}(d), b)\})^{+} \cup \rightsquigarrow$$

Die Eigenschaft gilt dann, wenn für die Speicherstellen des Systems die Eigenschaften *no-read-up* und *no-write-down* gelten, die Sicherheitspolicy aus Unterunterabschnitt 15.2.2.1 mit der neuen Interferenz-Relation $\rightsquigarrow'$ erfüllt ist, und die Beweisverpflichtung der Eigenschaft nicht im Widerspruch zu dieser Relation steht.

UMSTÄNDE DER DEKLASSIFIKATION Eigenschaften bezüglich der Umstände einer Deklassifikation können ebenfalls interaktiv verifiziert werden. Um auszudrücken, dass eine Deklassifikation immer nur nach dem Ausführen einer bestimmen Aktion (wie etwa Benutzerbestätigung) stattfinden darf, wird für jede solche Aktion ein Zähler z generiert. Dieser wird automatisch durch die Ausführung einer solchen Aktion erhöht.

Aus der modellierten Deklassifikationseigenschaft, die die möglichen Umstände der Deklassifikation einschränkt, wird eine neue Beweisverpflichtung generiert. Diese drückt aus, dass der Wert des Zählers z_d (entspricht der bisherigen Anzahl der Aufrufe der Deklassifikationsaktion a_d) für jeden möglichen Ablauf der ASM *kleiner oder gleich* sein muss als der Zähler z_u (entspricht der Anzahl der Ausführung der Aktion, die den notwendigen Umstand der Deklassifikation a_u–wie etwa Benutzerbestätigung–abbildet):

Siehe Unterabschnitt 12.2.2 für Modellierung von Deklassifikationseigenschaften

$$s(z_u) = s(z_d) \vee$$
$$s(z_u) = s(z_d) + 1$$

ERGEBNIS DER DEKLASSIFIKATION Die Formalisierung der Eigenschaft, dass ein öffentlicher Beobachter aus dem Ergebnis einer Deklassifikationsfunktion (nahezu) keine Information über ihre Eingabe schließen kann, ist abhängig vom Zweck, Kontext, und der Implementierung dieser Funktion.

Stenzel definiert dazu in [93] ein Schema für Formalisierung und Verifikation von komplexen Deklassifikationsfunktionen, die unbegrenzte Schleifen, komplexe Datentypen, und komplexe mathematische Funktionen enthalten können. Dies entspricht der Art von Funktionen, die mit MEL* spezifiziert werden können.

Siehe Abschnitt 11.3 für Details zur Modellierung solcher Methoden

Sei eine Funktion ff gegeben, die mit MEL* spezifiziert wurde, und in einer Deklassifikationsfunktion als eine sichere Filterfunktion festgelegt wurde. Gleichung 5 zeigt das Schema für die Eigenschaft

von ff, die bei geeignet gewählten Prädikaten assumption, property, enoughDifferences, sowie der Konstante c bewiesen werden muss, damit ff als sicher gilt.

$$\begin{aligned} \forall in, out.\mathbf{ff}(in) = out \wedge \mathbf{assumption}(in, out) \rightarrow \\ \exists set.size(set) > c \wedge \\ (\forall a.a \in set \rightarrow \mathbf{ff}(a) = out \wedge \mathbf{property}(a) \wedge \\ (\forall a, b.a \in set \wedge b \in set \wedge a \neq b \rightarrow \mathbf{enoughDifferences}(a, b)) \end{aligned} \tag{5}$$

Intuitiv teilt diese Eigenschaft die Eingaben der Funktion ff in Äquivalenzklassen ein, so dass verschiedene Eingaben aus derselben Äquivalenzklasse zur selben Ausgabe führen, wobei diese Äquivalenzklasse mindestens c Elemente haben muss. Selbst wenn ein öffentlicher Beobachter diese Ausgabe sieht, erfährt er nicht, welche der c möglichen Eingaben dazu geführt hat. Dabei legt assumption die Äquivalenzklasse, und property die Eingaben von ff fest, für die die Eigenschaft gelten soll.

In manchen Fällen ist es möglich, eine Menge von verschiedenen Eingaben zu konstruieren, die zur selben Ausgabe führen, aber dennoch genug Gemeinsamkeiten haben, um daraus die Eingabe abzuleiten. Daher muss enoughDifferences für jedes Paar von Eingaben aus der Äquivalenzklasse definieren, dass diese ausreichend verschieden sein müssen.

Da assumption, property, enoughDifferences, und c für jede Deklassifikationsfunktion verschieden sowie abhängig von ihrem Kontext sind, und dabei recht komplex ausfallen können, sind sie kein Teil der intuitiven Informationsflusseigenschaften von MODELFLOW, sondern müssen vom Entwickler bei der Verifikation festgelegt werden. Ein konkretes Beispiel, das das Schema instantiiert, ist in Kapitel 21 zu finden.

16 CODEANALYSE

Obwohl die Verifikation des generierten formalen Modells in IFlow bezüglich der transitiven bzw. intransitiven Nichtinterferenzeigenschaft durch den modellgetriebenen Ansatz vereinfacht wird, ist davon auszugehen, dass die vollautomatische Informationsflussanalyse des generierten Codes die bevorzugte Variante für die meisten App- und Webservice-Entwickler bleibt. Unterabschnitt 16.1.1 stellt die theoretischen Grundlagen der verwendeten sprachbasierten Analysetechniken vor, die dies ermöglichen.

Unterabschnitt 16.2.2 beschreibt die Verwendung des PDG-basierten Informationsflusskontrolltools Joana in IFlow. Unterabschnitt 16.2.3 stellt die Kontrollflussanalyse des manuell implementierten Codes auf Zugriffe der Java- bzw. Android-API vor. Unterabschnitt 16.2.4 zeigt die Schnittstelle zur Spezifikation der Informationsflusseigenschaften durch den Endnutzer sowie die Instrumentierung und Analyse einer IFlow-Anwendungskomponente.

16.1 GRUNDLAGEN

In den folgenden Abschnitten werden die Grundlagen der Informationsflussanalyse mit Programmabhängigkeitsgraphen vorgestellt, die in IFlow zum automatischen Check der modellierten Informationsflusseigenschaften eingesetzt wird, sowie die Berechnung der plattformspezifischen Informationsflussquellen und -senken, die für diesen Check notwendig sind.

16.1.1 *Informationsflussanalyse mit Programmabhängigkeitsgraphen*

PROGRAMM- UND SYSTEMABHÄNGIGKEITSGRAPHEN Hammer stellt in [35] eine vollautomatische Informationsflussanalyse von Java-Bytecode mit Hilfe von Programmabhängigkeitsgraphen (PDGs) bzw. Systemabhängigkeitsgraphen (SDGs) vor. PDGs repräsentieren Kontrollabhängigkeiten in Programmen, und wurden von Ferrante et al. in [28] definiert. Sie enthalten alle Knoten des Kontrollflussgraphen des Programms, die jedoch nicht durch Kontrollfluss-, sonder Daten- und Kontrollabhängigkeitskanten verbunden sind. Dabei existiert zwischen zwei Anweisungen eine Kontrollabhängigkeit, falls die erste Anweisung (wie etwa die Überprüfung eines Schleifenprädikats) direkt die Ausführung der zweiten Anweisung (wie etwa einer, die von der Schleife umschlossen ist) kontrolliert. Eine Datenabhängigkeit zwischen zwei Anweisungen existiert dann, wenn die erste

Anweisung einen Wert berechnet, der von der zweiten Anweisung verwendet wird.

Um multiprozedurale Programme als solche Abhängigkeitsgraphen abzubilden, führten Howitz et al. [45] die Systemabhängigkeitsgraphen ein, worin die Prozeduren als eigene Prozedurabhängigkeitsgraphen (ebenfalls PDGs) repräsentiert werden, die durch interprozedurale Kanten verbunden sind. Interprozedurale Abhängigkeiten bestehen aus Aufrufabhängigkeiten, die den Aufruf der Prozedur abbilden, Eingabe- und Ausgabeparameterabhängigkeiten, die die formalen und tatsächlichen Parameter der Prozedur verbinden, sowie zusammenfassende Abhängigkeiten, die die kontextsensitive Abhängigkeit zwischen tatsächlichen Ein- und Ausgabeparametern eines Aufrufs darstellen.

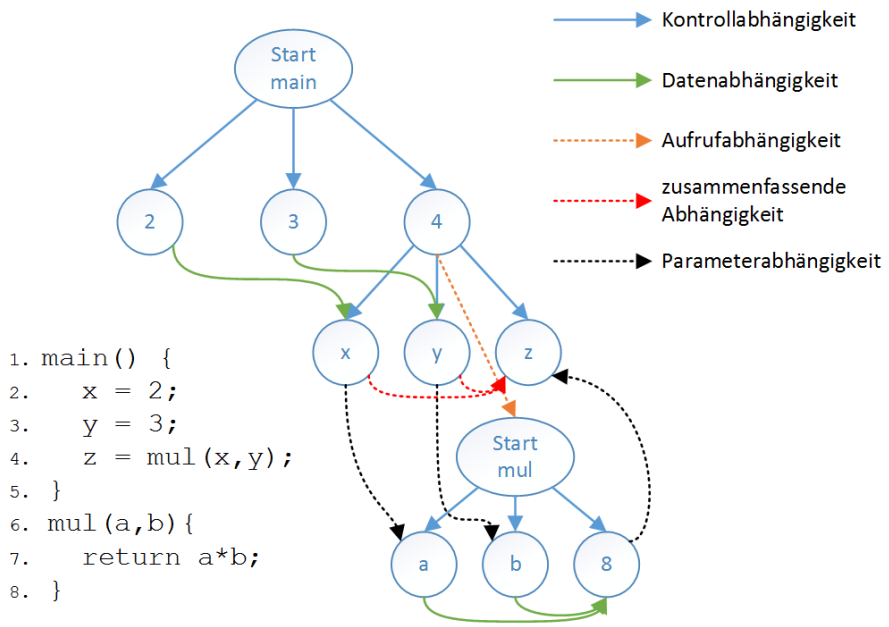


Abbildung 23: Simples Programm und dessen Systemabhängigkeitsgraph

Abbildung 23 zeigt ein simples Programm, das die Werte der Variablen `x` und `y` mit Hilfe der Prozedur `mul` miteinander multipliziert und das Ergebnis in Variable `z` ablegt. Die Initialisierung der Variablen `x` und `y` geschieht in Zeilen 2 und 3, die durch gleichnamige Knoten im SDG repräsentiert werden. Diese haben u.a. eine Kontrollabhängigkeit vom Start-Knoten der `main`-Prozedur, Parameterabhängigkeiten zu den Knoten a und b, die die Eingabeparameter der `mul`-Prozedur repräsentieren, sowie zusammenfassende Abhängigkeiten zum Knoten z, die aus dem Verhalten der `mul`-Prozedur berechnet wurden.

Um SDGs von Java Bytecode zu erstellen, wird u.a. eine *Point-To*-Analyse ausgeführt, um zu bestimmen, auf welches Objekt im Heap eine Referenz verweisen kann, sowie ob zwei Referenzen auf dassel-

be Objekt zeigen können bzw. müssen (Aliasing). Die Repräsentation der Parameterobjekte als Bäume sorgt dabei für die Objektsensitivität der SDGs, womit zwischen Objektfeldern unterschieden werden kann, während Kontextsensitivität durch Berechnung der zusammenfassenden Kanten erreicht wird. Dadurch wird die präzisere Programmanalyse auf Basis der PDGs erreicht. Auch k-CFA (*control flow analysis*)-Algorithmen [64] zur Points-To-Analyse werden unterstützt. Abhängigkeiten über Ausnahmebehandlungen werden ebenfalls im SDG repräsentiert, was jedoch vermehrt zu falsch positiven Ergebnissen der SDG-basierten Analyse aufgrund von impliziten Ausnahmen führen kann (da z.B. jede Java-Operation, die Objekte manipuliert, eine *NullPointerException* werfen kann), die bei der Ausführung nicht auftreten können [15, 35].

PDG-BASIERTE INFORMATIONSFLUSSANALYSE MIT JOANA Innerhalb eines PDGs können mittels *intraprocedural backwards slicing* $BS(v)$ alle Knoten $x \in PDG$ mit $x \to^* v$ berechnet werden, wobei $\to^*$ die Vorgängerrelation bezüglich der Abhängigkeitskanten darstellt. Ist also $x \in BS(v)$, so ist der Knoten v von x erreichbar, und hängt somit davon ab. Sind hingegen alle Knoten $x \in PDG$ gesucht, die von v abhängen, so können diese mittels *intraprocedural forward slicing* $FS(v)$ mit $v \to^* x$ berechnet werden. In [88] wird der Zusammenhang dieser Slicing-Algorithmen und der Definition der transitiven Nichtinterferenz nach Goguen und Meseguer [31] gezeigt, indem das Theorem

$$s \in BS(a) \implies dom(s) \rightsquigarrow dom(a) \tag{6}$$

bewiesen wird. Interferiert folglich die Domäne von a nicht die Domäne von s, so darf es im PDG keinen Pfad von $s \to^* a$ geben.

Durch die entsprechende Annotation von Knoten mit Sicherheitsdomänen, die in einer Interferenzrelation zueinander stehen, kann somit der Informationsfluss zwischen diesen Knoten überprüft werden. Ein Vorteil der PDG-basierten Informationsflussanalyse ist hierbei, dass nicht der gesamte PDG annotiert werden muss: für jeden PDG-Knoten gilt, dass dessen Sicherheitsdomäne von den Domänen ihrer Vorgängerknoten interferiert werden muss. Es muss daher lediglich der Knoten v, der die Eingabe repräsentiert, mit dem Sicherheitslevel $S(v)$ annotiert werden, während der Knoten w, der die Ausgabe repräsentiert, mit dem geforderten Sicherheitslevel $R(w)$ markiert wird. Somit kann die Sicherheitsdomäne $S(x)$ jedes Knotens x, das ein Nachfolger des annotierten Knotens v mit $x \in FS(v)$ ist, automatisch berechnet werden ($S(v) \rightsquigarrow S(x)$). Die geforderte Domäne $R(w)$ muss dabei von der berechneten Sicherheitsdomäne $S(w)$ interferiert werden ($S(w) \rightsquigarrow R(w)$); anderenfalls ist das Programm nicht informationsflusssicher.

Das Programm aus Abbildung 23 kann somit folgendermaßen auf dessen Informationsflussicherheit geprüft werden. Zunächst müssen Sicherheitsdomänen, wie etwa $\mathtt{high}$ und $\mathtt{low}$, sowie ihre Interferenzrelation, wie z.B. $\mathtt{low} \rightsquigarrow \mathtt{high}$, spezifiziert werden. Um festzulegen, dass der Wert von `x` nicht das Ergebnis der Multiplikationsoperation interferieren darf, müssen entsprechend der Knoten `2` als *high* ($S(2) = \mathtt{high}$), und Knoten `z` als *low* ($R(z) = \mathtt{low}$) annotiert werden. Da es einen Pfad von `2` zu `z` gibt, wird die Sicherheitsdomäne von `z` als *high* berechnet ($S(z) = \mathtt{high}$), was jedoch im Widerspruch zur geforderten Sicherheitsdomäne *low* steht ($S(z) \not\rightsquigarrow R(z)$), womit das Programm als unsicher eingestuft wird.

Die Konstruktion der PDGs von monolithischen Java-Anwendungen sowie die Algorithmen zur PDG-basierten Informationsflussanalyse sind im Tool Joana [30] der Karlsruher Universität implementiert.

EINSATZ IN IFLOW Trotz den Nachteilen einer statischen Analyse, deren Ergebnisse zwangsläufig eine Überapproximation der in der Realität möglichen Informationsflüsse darstellen, sowie die vielen Einstellungsmöglichkeiten bei der Analyse (wie etwa die Wahl zwischen einer Mehrzahl von objektsensitiven und k-CFA Points-To-Analysealgorithmen), deren Auswirkung auf die Qualität der Analyseergebnisse je nach Anwendung variiert und schwer vorhersagbar ist, sprechen folgende Argumente für die Anwendung der PDG-basierten Informationsflussanalyse in IFlow:

- automatischer Check der transitive Nichtinterferenz-Eigenschaft nach [31] bzw. [80]
- minimale erforderliche Sicherheitsannotation [36, 49]
- vergleichsweise hohe Präzision durch Berücksichtigung der Kontrollflüsse, des Kontextes, sowie der verschiedenen Objekte und ihrer Felder [35, 89]
- Fokus auf Java-„Sicherheitskernel“ [35], was in IFlow dem zu analysierenden Codeskelett entspricht
- Soundness der Informationsflussanalysealgorithmen [98, 99]

Dazu wird der PDG des automatisch generierten und zu Java-Bytecode kompilierten Codeskeletts entsprechend der modellierten Informationsflusseigenschaften annotiert, und so auf Informationsflussverletzungen geprüft (siehe Unterabschnitt 16.2.2 für Details).

Obwohl die Theorie der PDG-basierten Informationsflussanalyse auch Deklassifikation zulässt [35], wird der Zusammenhang zur intransitiven Nichtinterferenz nicht gezeigt, weshalb hierzu das generierte bzw. kompilierte Codeskelett vor der Analyse bzgl. Deklassifikationseigenschaften entsprechend angepasst werden muss (siehe Unterunterabschnitt 16.2.2.2 und Unterunterabschnitt 16.2.4.3 für Details).

JOANA VS. JIF Neben Joana existieren weitere Tools zur sprachbasierten Informationsflusskontrolle von echten Anwendungen. FlowDroid [2] wurde speziell (und ausschließlich, was gegen dessen Einsatz in IFlow spricht) für Informationsflussanalyse von Android-Anwendungen entwickelt.

Jif ist neben Joana das bekannteste Tool zur Analyse von Java-Code, und stellt eine Erweiterung von Java dar, die auf dem *Decentralized Label Model* aufbaut [69]. Dieses erlaubt es, Vertraulichkeit und Integrität von Daten sowie ihre Deklassifikation durch Sicherheitstypen und spezielle Operationen zu spezifizieren. Informationsflusskontrolle wird sowohl durch einen statischen Check als auch dynamische Sicherheitstypüberprüfung während der Laufzeit umgesetzt.

Jif erfordert jedoch eine Vielzahl von Sicherheitsannotationen des Quellcodes, da sie oft nicht automatisch inferiert werden können. Selbst die Annotation vergleichsweise simpler Anwendungen fällt notorisch komplex aus [36, 75], und erfordert oft zusätzliche Deklassifikationen: eine Beobachtung, die bei der Anwendung von Jif auf die Travel Planner-Fallstudie bestätigt wurde [49]. Dafür müssten die Anwendungsmodelle in IFlow durch zahlreiche zusätzliche Annotationen erweitert werden, was die Modellierung mit MODELFLOW deutlich erschweren würde. Die erforderte explizite Behandlung von impliziten Ausnahmen im Code, das Fehlen der Jif-Laufzeitumgebung für die Android-Plattform, sowie die stagnierte Entwicklung von Jif sprechen ebenfalls gegen dessen Einsatz in IFlow [49].

16.1.2 *Plattformspezifische Informationsquellen und -senken*

INFORMATIONSFLUSSQUELLEN UND -SENKEN In Abschnitt 12.2 zur Modellierung von Informationsflusseigenschaften mit MODELFLOW wurde bereits das Konzept von Informationsflussquellen und -senken eingeführt. Unterabschnitt 16.1.1 beschreibt, wie dieses Konzept in PDG-basierter Informationsflussanalyse zum Tragen kommt: durch die Annotation eines Knotens v mit der Sicherheitsdomäne $S(v)$ wird dieses als eine Quelle von Information festgelegt, während das Markieren eines Knotens w mit der geforderten Sicherheitsdomäne $R(w)$ dieses als eine Informationssenke spezifiziert. Dabei bleibt jedoch die Frage offen, wie solche Knoten bestimmt werden sollen.

Neben Quellen und Senken, die sich auf konkrete Benutzereingaben bzw. Anwendungskomponenten beziehen, müssen auch anwendungsunabhängige, plattformspezifische Quellen und Senken berücksichtigt werden, die von der Anwendung verwendet werden können. Während die Abbildung von anwendungsspezifischen Quellen und Senken auf PDG-Knoten des generierten Codeskeletts in Unterabschnitt 16.2.2 erläutert wird, erfolgt der Zugriff auf plattformspezifische Quellen und Senken über die API der Plattformen, auf denen die finale Anwendung ausgeführt wird. Solche APIs sind üblicherwei-

se sehr umfangreich (die API von Android 4.2 umfasst beispielsweise 110.000 öffentliche Methoden). Das Erkennen von Methoden, die als Informationsquelle oder -senke verwendet werden können (wie etwa für den Zugriff auf das Dateisystem oder Sensoren des Geräts) stellt daher eine Herausforderung dar. Die Aussagekräftigkeit der Informationsflussanalyse hängt jedoch sehr davon ab, ob solche Methoden dabei berücksichtigt worden sind.

ANSATZ ZUM KLASSIFIZIEREN UND KATEGORISIEREN VON QUELLEN UND SENKEN DER ANDROID-API [77] stellt mit *SuSi* einen Ansatz vor, der zum Bestimmen und Kategorisieren solcher Quellen und Senken in der Android API verwendet werden kann. Hierfür wird der Android-Quellcode auf eine Reihe von syntaktischen und semantischen Merkmalen mit Hilfe von maschinellem Lernen analysiert. Es ist naheliegend, jedoch unzureichend, dabei lediglich die durch das Android-Berechtigungssystem geschützte Schnittstellen zu betrachten, da diese nicht alle API-Methoden abdecken, die zur In- oder Exfiltration von Information verwendet werden können (ein Umstand, der auch von Malware ausgenutzt werden kann). Dazu definiert [77] *Quellen* als Aufrufe von API-Methoden, die geteilte Ressourcen (wie etwa die Identifikationsnummer des Geräts) lesen oder schreiben, und einen nicht-konstanten Wert zurückgeben. *Senken* werden als Aufrufe von API-Methoden definiert, die mindestens einen nicht-konstanten Wert aus dem Anwendungscode als Parameter entgegennehmen, und eine geteilte Ressource mit diesem Wert beschreiben.

In dem Ansatz wurde überwachtes Maschinenlernen eingesetzt. Der Klassifikator wurde anhand einer Untermenge von händisch annotierten Beispielen eintrainiert, und anschließend zur Klassifikation und Kategorisierung der Android-API verwendet. Dabei sind Eigenschaften wie Methodenname, Klassenname, Methodenparameter sowie ihre Typen, Datenflüsse von Eingabeparametern zu eingebetteten Methodenaufrufen (sowie von ihren Ausgaben zum Rückgabeparamter), und die geforderte Berechtigung in die Klassifikation eingeflossen. Das Verfahren wurde anhand einer Untermenge der händisch annotierten Methoden evaluiert, und dabei eine Präzision von etwa 90% festgestellt.

EINSATZ IN IFLOW In IFlow werden Listen von anwendungsunabhängigen Quellen und Senken zur Analyse der manuell implementierter Anwendungsmodule eingesetzt. Die Kategorien von Quellen und Senken sind dabei in MODELFLOW durch die Attribute der vordefinierten Klassen *PredefinedSources* und *PredefinedSinks* repräsentiert.

Android API-Dokumentation: `developer.android.com`
Java API-Dokumentation: `docs.oracle.com`

Obwohl SuSi nicht alle Quellen und Senken der Android-Plattform erkennt, und mehrere falsch positiven Ergebnisse liefert, ist die dadurch berechnete und kategorisierte Liste von Methoden vollständiger als die händisch kompilierten Alternativen [77]. Da die Android API den Großteil der Java API implementiert, kann die mit SuSi generierte Liste von Quellen und Senken auch für die Analyse von Java-Code verwendet werden, falls dieser nur die in der Android-API enthaltene Java-API nutzt. Dies kann in IFlow genutzt werden, um nicht nur Android-, sondern auch Webservice-Javacode auf verbotene Zugriffe auf Quellen und Senken zu prüfen. Dabei muss die dem Entwickler verfügbare API nicht gravierend eingeschränkt werden, was die Implementierung flexibler und komfortabler macht.

16.2 CODEANALYSE IN IFLOW

Die folgenden Abschnitte beschreiben den Ansatz zur automatischen Analyse des automatisch generierten Codeskeletts als auch der manuell implementierten Anwendungskomponenten, um die modellierten Informationsflusseigenschaften zu garantieren.

16.2.1 *Aufbau des Codeskeletts*

Dieser Abschnitt beschreibt den Aufbau des Java-Codeskeletts, der aus einem MODELFLOW-Modell einer IFlow-Anwendung generiert wird [47]. Unterabschnitt 15.2.1 stellt den Aufbau des dazugehörigen formalen Modells vor, während Abschnitt 17.1 ihre Refinement-Beziehung diskutiert. Die Elemente des Codeskeletts werden Anhand der *TravelPlanner*-Fallstudie erläutert (siehe Abschnitt 10.5 für die statische und Abschnitt 11.4 für die dynamische Sicht auf die Anwendung).

Während dieser Abschnitt die Grundbausteine des Codeskeletts erläutert, beschreibt und diskutiert Abschnitt 17.2 im Detail die eingesetzten Abstraktionstechniken, sowie die Besonderheiten bei der Abbildung der modellierten Programmabläufe und der Komponentenkommunikation. Unterabschnitt 17.2.2 erläutert zudem die Abstraktion der Anwendungsmodule wie die graphische Benutzeroberfläche, externe Komponenten, und manuelle Methoden.

DATENTYPEN Primitive MODELFLOW-Datentypen werden auf Java-Datentypen abgebildet. *Integer* entspricht einem *int*, *String* einem Java-String, *double* einem *double*, und *Boolean* einem *boolean*.

Aus komplexen MODELFLOW-Datentypen werden gleichnamige Java-Klassen generiert, die von der vordefinierten IFlow-Klasse `IFlowData` erben. Die Attribute dieser Datentypen werden als Attribute solcher Klassen abgebildet.

Siehe Abbildung 9 für das vollständige Klassendiagramm von TravelPlanner

Listing 4 zeigt den relevanten Ausschnitt aus der Java-Klasse des *RequestData*-Datentyps der *Travel-Planner*-Fallstudie, der das Attribut *date* von Typ *String* enthält. Die `copy`-Funktionen implementieren dabei die Kopiersemantik von MEL* Abschnitt 11.1: bei Zuweisungen, Nachrichtenversand, und Übergabe als Parameter wird eine tiefe Kopie des referenzierten Objekts erstellt.

Die Funktion `extractSecurityLevel` gibt einen Integer-Wert zurück, der vom Inhalt einer Instanz von `RequestData` abhängt. Mit `propagateSecurityLevel` kann die Abhängigkeit dieses Inhalts von einem Integer-Wert simuliert werden. Diese Funktionen werden nur für die statische Informationsflussanalyse benötigt; für Details siehe Unterunterabschnitt 17.2.1.1.

Listing 4: Spezifikation des *RequestData*-Datentyps

```
1  public class RequestData implements IFlowData {
2          public String date = new String();
3
4          public RequestData copy() {
5                  RequestData ret = new RequestData();
6                  ret.copy(this);
7                  return ret;
8          }
9
10         private void copy(RequestData from) {
11                 this.date = from.date;
12         }
13
14         public int extractSecurityLevel() {
15                 return IFUtility.extractSecurityLevel(this.date);
16         }
17
18         public void propagateSecurityLevel(int securityLevel) {
19                 this.date = IFUtility.getTaggedType(String.class,
                       securityLevel);
20         }
21         ...
22 }
```

KOMPONENTEN Anwendungskomponenten, vordefinierte und manuelle grafische Oberflächen, sowie externe Apps und Services werden auf gleichnamige Java-Klassen abgebildet. Ihre modellierten Attribute werden auf gleichnamige Java-Attribute dieser Klassen abgebildet.

Siehe Kapitel 11 für Details zur Modellierung der dynamischen Sicht auf eine IFlow-Anwendung

Das Verhalten der Anwendungskomponenten wird aus den modellierten Sequenz- und Aktivitätsdiagrammen abgeleitet.

Für jeden Typ einer Nachricht, die die Komponente empfängt, wird eine eigene Nachrichtenbehandlungssmethode generiert. Der einzige Parameter dieser Nachricht ist eine Instanz des entsprechenden Nach-

richtendatentyps. Im einfachsten Fall bildet diese Methode die modellierten Anweisungen zwischen dem Empfang einer *CallMessage* bzw. *ReplyMessage*, und dem Versand der nächsten *CallMessage* bzw. *ReplyMessage*. Der Versand einer *CallMessage* bzw. *ReplyMessage* wird daher als ein Aufruf der Nachrichtenbehandlungssmethode der Empfängerkomponente abgebildet. Die MEL*-Anweisungen wie Zuweisungen, Methodenaufrufe, arithmetische Operationen, und Variablendeklarationen werden dabei auf entsprechende Java-Anweisungen abgebildet (für Details siehe [65]).

Listing 5 zeigt einen relevanten Ausschnitt aus der Java-Klasse, die die Anwendungskomponente *Airline* aus der *TravelPlanner*-Fallstudie abbildet. Abbildung 24 zeigt den dazugehörigen Ausschnitt aus dem vollständigen Sequenzdiagramm. Der Attribut `flightOffers` bildet das modellierte Komponentenattribut *flightOffers* von *Airline* ab. Die Methode `getFlightOffers` implementiert den Empfang und die Behandlung der *GetFlightOffers*-Nachricht.

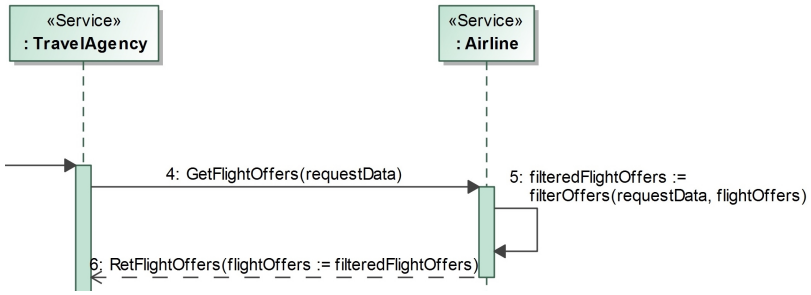


Abbildung 24: Ausschnitt aus dem Sequenzdiagramm der TravelPlanner-Fallstudie

Listing 5: Spezifikation der Anwendungskomponente *Airline*

```
1  public class Airline extends IFlowService {
2
3          public static List<FlightOffer> flightOffers = new
               IFlowArrayList<FlightOffer>();
4
5          public static void getFlightOffers(GetFlightOffers inmsg0
               ) {
6                  RequestData requestData = inmsg0.requestData;
7                  List<FlightOffer> filteredFlightOffers;
8                  filteredFlightOffers = new IFlowArrayList<
                       FlightOffer>(filterOffers(requestData.copy(),
                        new IFlowArrayList<FlightOffer>(flightOffers
                       )));
9                  TravelAgency.retFlightOffers(new RetFlightOffers(
                       filteredFlightOffers));
10         }
11         ...
12 }
```

Der Empfang der Nachricht 4 in Abbildung 24 entspricht dabei den Zeilen 13 und 14 in Listing 5. Zeile 13 spezifiziert die Signatur der Behandlungsmethode für Nachrichten vom Typ *GetFlightOffers*. Zeile 14 weist den Inhalt der Nachricht automatisch der lokalen Variable *requestData* zu. Zeile 16 entspricht dem Aufruf der manuellen Methode *filterOffers* und der Zuweisung ihres Rückgabewerts an die Variable *filteresFlightOffers* in Nachricht 5 des Sequenzdiagramms. An `filterOffers` wird eine Tiefenkopie der lokalen Variable `requestData` übergeben, was der Kopiersemantik von MEL* und des formalen Modells entspricht. Dadurch wird verhindert, dass die zukünftige Implementierung der manuellen Methode den Inhalt des `requestData`-Objekts ändern kann. Abschließend wird in Zeile 17 diese Variable als Inhalt der Antwortnachricht *RetFlightOffers* an *TravelAgency* verschickt, indem diese als Parameter an ihre Nachrichtenbehandlungsfunktion `retFlightOffers` übergeben wird. Siehe Unterunterabschnitt 17.2.1.4 für Details bezüglich Komponentenkommunikation der finalen Anwendung inklusive Implementierung des Webservice-Interfaces sowie Kodierung und Konvertierung der Nachrichten.

Siehe Unterabschnitt 11.2.2 zu Details über automatische Generierung von lokalen Variablen

MEL*-Methoden der Anwendungskomponenten werden zu gleichnamigen Java-Methoden transformiert, wobei die darin enthaltenen MEL*-Anweisungen zu entsprechenden Java-Anweisungen transformiert werden. Unterspezifizierte Anwendungsmodule wie die graphische Benutzeroberfläche und externe Komponenten werden abstrahiert, wobei solche Abstraktionen die Informationsflüsse ihrer konkreten Implementierung überapproximieren. Details sowie Pseudocode dieser Abstraktionen werden in Unterabschnitt 17.2.2 gezeigt und im Detail diskutiert. Unterunterabschnitt 17.2.1.5 erläutert dabei die Überapproximation und Abbildung der in der finalen Anwendung möglichen Programmabläufe. Allgemeine Abstraktionstechniken, die bei der Generierung des Codeskeletts zum Einsatz kommen, werden in Unterunterabschnitt 17.2.1.1 diskutiert.

16.2.2 *Informationsflussanalyse des Codeskeletts*

Informationsflusseigenschaften werden in IFlow anhand des generierten Codeskeletts mittels PDG-basierter Informationsflussanalyse durch Joana garantiert. Anwendungsmodule sowie die Plattformen, auf welchen die verteilte Anwendung ausgeführt werden soll, werden dabei durch Stubs abgebildet, die die Informationsflüsse der realen Implementierung emulieren bzw. überapproximieren und von ihrer Funktionalität abstrahieren. Abschnitt 17.2 argumentiert im Detail, weshalb die Eigenschaften, die so für das monolithische, plattformunabhängige Codeskelett garantiert werden, auch für die finale, verteilte Anwendung gelten, während Abschnitt 17.3 die hierfür getroffene Annahmen an den Anwendungskontext erläutert.

Meldet die Analyse keine Informationsflussverletzungen, so kann der Code der Anwendungskomponenten (*blaue* Kästen in Abbildung 20, (4) und (5)) auf Pakete verteilt werden. Die Stubs der manuellen Module (*rote* Kästen in Abbildung 20, (4) und (5)) werden durch ihre reale Implementierung ersetzt, falls diese nicht auf illegale APIs zugreifen (vgl. Beschreibung ihrer Analyse in Unterabschnitt 16.2.3).

Um die Komponentenkommunikation sowie Deployment auf ein Gerät zu ermöglichen, werden zusätzliche Proxy-Klassen und Konfigurationsdateien automatisch aus dem Anwendungsmodell generiert (*grüne* Kästen in Abbildung 20, vgl. dazu Unterunterabschnitt 17.2.1.4 für Diskussion des Refinements).

16.2.2.1 *Transitive Nichtinterferenz-Eigenschaften*

Bei transitiven Nichtinterferenz-Eigenschaften handelt es sich um Informationsflusseigenschaften, die durch eine Sicherheitspolicy mit einer transitiven Interferenzrelation ausgedrückt werden können. Ihre Modellierung erfolgt (optional) durch eine Sicherheitspolicy, sowie durch explizite Eigenschaften, die die Informationsflüsse zwischen Quellen und -senken bedingungslos erlauben bzw. verbieten (siehe Unterabschnitt 12.2.1 für Details).

Für eine Informationsflussanalyse mit Joana müssen Knoten des generierten PDGs mit Sicherheitsdomänen annotiert werden. Da der PDG aus dem Bytecode der Anwendung generiert wird, muss hierfür der Java-Code des Codeskeletts zu Java-Bytecode kompiliert werden. Die Annotationen bzgl. erlaubter bzw. verbotener Informationsflüsse werden aus der Informationsflusstabelle abgeleitet (vgl. Unterabschnitt 12.2.5), wobei ihre Zeilen und Spalten die MEL-Repräsentation der modellierten bzw. plattformspezifischen Quellen und Senken darstellen.

Als Schnittstelle zu Joana wird das Tool *IFC Console* verwendet, das Annotationen von Java-Klassenattributen und -Methodenparameter automatisch zu Annotation der entsprechenden PDG-Knoten übersetzt. Aus jedem Paar aus modellierter Quelle und Senke, zwischen denen der Informationsfluss verboten ist, wird dabei eine eigene Checkverpflichtung in Form einer Batch-Datei für *IFC Console* generiert. Für den Check werden die Sicherheitsdomänen *high* und *low* und die Interferenzrelation $\mathtt{low} \rightsquigarrow \mathtt{high}$ verwendet. Die PDG-Knoten, die eine Quelle abbilden, werden dabei mit *high* annotiert, während solche, die eine Senke repräsentieren, als *low* markiert werden. Die mit MODELFLOW modellierten Sicherheitsdomänen werden bereits bei der Modelltransformation auf Konsistenz und Widerspruchsfreiheit mit den modellierten Eigenschaften überprüft, und sind für den automatischen Check nicht notwendig.

IFC Console, `pp.ipd.kit.edu/projects/joana/`

Meldet Joana bei allen Checks, die aus einer modellierten Informationsflusseigenschaft abgeleitet wurden, keine Informationsflussverletzung, so gilt diese Eigenschaft für die modellierte IFlow-Anwendung.

Die folgenden Regeln gelten bei der Übersetzung der MEL-Quellen und -Senken zu Java-Attributen und Methodenparametern:

Vergleiche dazu Unterunterabschnitt 15.2.2.2 für die Übersetzung solcher Quellen und Senken zu Speicherstellen und Aktionen des formalen Modells

- Eine Quelle oder Senke, die ein MODELFLOW-Komponentenattribut referenziert, wird auf das entsprechende Attribut der Java-Klasse abgebildet, die diese Komponente im Codeskelett repräsentiert
- Eine Senke, die eine MODELFLOW-Komponente referenziert, wird auf die Eingabeparameter ihrer Nachrichtenbehandlungsmethoden abgebildet, sowie auf Attribute, an die das Ergebnis der *decrypt*-Operation zugewiesen wird. Dies ist notwendig, da der Fluss geheimer Klartextdaten zu einer Komponente erst nach ihrer Entschlüsselung festgestellt werden kann (siehe Unterunterabschnitt 17.2.2.3 für Details)
- Eine Quelle, die eine MODELFLOW-Komponente referenziert, wird auf die Eingabeparameter der Nachrichtenbehandlungsmethoden abgebildet, die diese Komponente aufruft
- Eine vordefinierte, plattformspezifische Senke wird auf die Eingabeparameter der Methoden abgebildet, die auf diese Senke zugreifen (modelliert als Tag *sink* des «`uses`»-Stereotyps, vgl. Unterunterabschnitt 9.2.2.4)
- Eine vordefinierte, plattformspezifische Quelle wird auf die Rückgabeparameter der Methoden abgebildet, die auf diese Quelle zugreifen (modelliert als Tag *source* des «`uses`»-Stereotyps, vgl. Unterunterabschnitt 9.2.2.4)
- Quellen und Senken, die mit den Stereotypen «`otherSinks`» und «`otherSources`» annotiert sind, werden bereits beim Generieren der Informationsflusstabelle auf entsprechende Komponenten und ihre Attribute abgebildet (vgl. Unterabschnitt 12.2.1)
- Eine Quelle, die eine mit *„label“* beschriftete Benutzereingabe repräsentiert, wird auf die Eingabeparameter der Nachrichtenbehandlungsmethode abgebildet, die aus einer mit *„label“* annotierten Sequenzdiagrammnachricht generiert wurde (vgl. Unterabschnitt 11.2.3)
- Eine Quelle oder Senke, die einen anderen Systemnutzer repräsentiert, wird auf das Attribut *sl* der *OtherUser*-Klasse abgebildet, welches dessen internen Zustand repräsentiert (vgl. Unterunterabschnitt 17.2.1.5)

Bei der automatisch generierten Checkverpflichtung wird neben der PDG-Annotation und den dafür benötigten Sicherheitsdomänen auch der Algorithmus zur Points-To-Berechnung angegeben. Standardmäßig werden in IFlow unbeschränkt objektsensitive SDGs generiert, was jedoch je nach Anwendung trotz der genutzten Codeabstraktionstechniken zu aufwendig ausfallen oder zu falsch positiven Ergebnissen führen kann (siehe Erläuterung in Unterabschnitt 16.1.1). Der Entwickler kann in diesen Fällen über die Benutzeroberfläche der *IFC Console* andere Algorithmen wie *3-CFA* angeben. Da die Informationsflussanalyse von Joana *sound* ist [98, 99], können dadurch keine Informationsflussverletzungen unentdeckt bleiben. Kann das Codeskelett aufgrund der Größe oder falsch positiven Ergebnisse nicht analysiert werden, kann der Entwickler das Modell bzw. Codeskelett der Anwendung entsprechend anpassen, oder die Eigenschaft interaktiv verifizieren.

16.2.2.2 *Intransitive Nichtinterferenz-Eigenschaften*

Bei intransitive Nichtinterferenz-Eigenschaften handelt es sich um Informationsflusseigenschaften, die durch eine Sicherheitspolicy mit einer intransitiven Interferenzrelation ausgedrückt werden müssen. Ihre Modellierung erfolgt (optional) durch eine Sicherheitspolicy, sowie durch explizite Eigenschaften, die die Informationsflüsse zwischen Quellen und -senken nach expliziter Deklassifikation über eine Deklassifikationsmethode erlauben (siehe Unterabschnitt 12.2.2 für Details).

PDG-basierte Informationsflusskontrolle kann auch genutzt werden, um solche intransitive Nichtinterferenz-Eigenschaften zu garantieren. Dies wird jedoch nicht wie in [35] beschrieben durch das Annotieren von PDG-Knoten als Deklassifikationsknoten erreicht, da [88] den formalen Zusammenhang zwischen Slicing-Algorithmen und der Nichtinterferenz nur für die transitive Interferenzrelation beweist, und die *IFC Console* diese Funktionalität ohnehin nicht zur Verfügung stellt. Stattdessen werden die Vorzüge des modellgetriebenen Ansatzes ausgenutzt, um den Check der intransitiven Nichtinterferenz-Eigenschaften auf den Check der transitiven Nichtinterferenz zurückzuführen, der wiederum mit Joana ausgeführt werden kann.

Diese Technik kann nur bei Deklassifikationsmethoden eingesetzt werden, die mit MEL* modelliert wurden (vgl. Abschnitt 11.3), und deren Sicherheit als Filter bzw. Anonymisierungsfunktion formal gezeigt wurde (siehe Unterunterabschnitt 15.2.2.2 für Details zur Beweistechnik). Solche Methoden sind in der modellierten Informationsflusseigenschaft mit dem Tag *filter* des «`via`»-Stereotyps markiert.

Die in IFlow modellierten intransitive Nichtinterferenz-Eigenschaften fordern, dass die Informationsflüsse zwischen einer Quelle `q` und Senke `s` nur über eine Filtermethode `decl(in) : ret` erlaubt sind. Eine

sinnvolle Filtermethode besitzt zudem eine Abhängigkeit zwischen ihren formalen Ein- und Ausgabeparameter, also gilt $\mathtt{in} \rightarrow^* \mathtt{ret}$ im PDG dieser Methode. Die Modellierungsrichtlinien für MEL*-Methoden garantieren, dass diese Methode frei von Seiteneffekten ist, d.h., sie greift nur auf ihre Ein- und Ausgabeparameter sowie darin deklarierte lokale Variablen zu (vgl. Abschnitt 11.3). Folglich muss jeder Pfad von $\mathtt{in}$ zu $\mathtt{ret}$ ein Teilpfad jedes Pfades zwischen $\mathtt{q}$ und $\mathtt{s}$ sein; anderenfalls gäbe es einen Fluss von Information zwischen $\mathtt{q}$ zu $\mathtt{s}$, die nicht mit der Deklassifikationsmethode $\mathtt{decl}$ gefiltert bzw. anonymisiert wurde.

Werden nun im SDG des generierten Codeskeletts alle Abhängigkeitskanten zwischen den formalen Parametern $\mathtt{in}$ und $\mathtt{ret}$ (sowie die zusammenfassende Kanten zwischen den dazugehörigen, tatsächlichen Parametern) entfernt, so darf es im resultierten SDG keinen Pfad von $\mathtt{q}$ zu $\mathtt{s}$ mehr geben, was wiederum einer transitiven Nichtinterferenz-Eigenschaft entspricht, die mit Joana überprüft werden kann: $\mathtt{dom}(\mathtt{q}) \not\rightsquigarrow \mathtt{dom}(\mathtt{s})$.

Die beschriebene Modifikation des SDGs wird durch eine entsprechende Anpassung des Codeskeletts vorgenommen, in dem der Rumpf der Deklassifikationmethode $\mathtt{decl}$ durch eine Return-Anweisung ersetzt wird, die ein mit Standardwerten gefülltes Objekt zurückgibt (siehe Tabelle 6 für IFlow-Standardwerte). Dieses hängt nicht von den Eingabeparametern der Methode ab, weswegen der SDG der Anwendung keine Abhängigkeitskanten zwischen den formalen Parametern der Methode (und daher auch keine zusammenfassende Kanten zwischen ihren tatsächlichen Parametern) enthält. Dies ist eine sinnvolle Abstraktion einer bewiesen sicheren Filterfunktion, da ein öffentlicher Beobachter ihrer Ausgabe dadurch nicht auf die Eingabe schließen kann. Da das Codeskelett wie auch die Checkverpflichtung automatisch aus dem abstrakten Anwendungsmodell generiert wird, ist ein eindeutiger Zusammenhang zwischen dem Codeskelett und dessen modifizierter Variante gegeben; das modifizierte Codeskelett wird hierbei lediglich für den Check einer konkreten modellierten intransitive Nichtinterferenz-Eigenschaft mit Joana genutzt. Es ist somit ausgeschlossen, dass zu viele Kanten im SDG entfernt werden, was dazu führen könnte, dass unerwünschte Informationsflüsse nicht mehr gefunden werden können. Wird beim Check keine Informationsflussverletzung gemeldet, so gilt die modellierte intransitive Nichtinterferenz-Eigenschaft für die finale Anwendung. Für die Annotation der PDG-Knoten als Quellen und Senken sowie die Einstellungsparameter der Analyse gelten dieselben Regeln wie in Unterunterabschnitt 16.2.2.1.

Die PDG-basierte Informationsflussanalyse von intransitive Nichtinterferenz-Eigenschaften erfordert die formale Verifikation der relevanten Filtermethoden. Der Check solcher Eigenschaften ist daher erst dann vollautomatisch, wenn die formalen Beweise der Sicherheit dieser Methoden bereits vorliegen. Dies ist beispielsweise dann der Fall, wenn die Spezifikation der bereits verifizierten Filtermethoden in weiteren Fallstudien wiederverwendet wird, oder wenn die Eigenschaft vom Anwendungsnutzer spezifiziert wurde und die Sicherheit der darin referenzierten Filtermethode bereits vom Entwickler erbracht wurde (vgl. Unterabschnitt 16.2.4 für Spezifikation und Check von Benutzereigenschaften).

Intransitive Nichtinterferenz-Eigenschaften, die den Informationsfluss nur nach Verschlüsselung erlauben, können ohne Anpassung des SDGs vollautomatisch gecheckt werden. Kryptographische Operationen werden in IFlow als eigenständige Anwendungsmodule behandelt, die für die codebasierten Informationsflussanalyse abstrahiert werden (für Details siehe Unterunterabschnitt 17.2.2.3). Diese Abstraktion garantiert, dass das Ergebnis der Verschlüsselung nicht von den Klartextdaten abhängt; werden die verschlüsselten Daten mit dem korrekten Schlüssel wieder entschlüsselt, so hängt das Ergebnis wiederum von dem ursprünglichen Klartext ab. Kryptographische Operationen stellen somit eine Sonderform von Filterfunktionen dar, deren Sicherheit bezüglich Informationsfluss bereits von Küsters et al. [56] gezeigt wurde. Auch hier wird die Eigenschaft auf die transitive Nichtinterferenz reduziert, die mit Joana vollautomatisch gezeigt werden kann.

16.2.3 *Check der manuellen Implementierung*

In IFlow ist es möglich, das generierte Codeskelett um händisch implementierten Code der Anwendungsmodule „manuelle Methode“ und „manuelle graphische Oberfläche“ zu erweitern. Bei der Informationsflussanalyse des generierten Codes werden Abstraktionen der zukünftigen Implementierungen solcher Module genutzt, die eine Reihe von Annahmen über diese Implementierung treffen (siehe Details zu Refinement in Unterunterabschnitt 17.2.2.1 und Unterunterabschnitt 17.2.2.5).

16.2.3.1 *Anforderungen an die Implementierung*

ANFORDERUNGEN AN DIE MANUELLE METHODEN Für die Implementierung einer manuellen Methode gilt, dass diese zustandslos sein muss, und nur auf im Modell erlaubte Quellen und Senken zugreifen darf (für Begründung siehe Unterunterabschnitt 17.2.2.1 zu Coderefinement). Konkret bedeutet dies Folgendes:

1. Der Code darf nur auf den Teil der plattformspezifischen API zugreifen, der im MODELFLOW-Modell über den «uses»-Stereotyp erlaubt ist, oder nicht als Quelle bzw. Senke privater Information genutzt werden kann
2. Zugriff auf Java-API, die nicht Teil der Android-API ist, ist verboten
3. Zugriff auf Klassen und Methoden des IFlow-Frameworks ist verboten, da das Framework intern plattformspezifische APIs und geteilte Ressourcen nutzt
4. Deklaration und Nutzung neuer Java-Klassen ist verboten, da so zwischen Methoden(-aufrufen) geteilte, statische Ressourcen implementiert werden können
5. Zugriff auf den internen Zustand der Anwendung (wie etwa den Inhalt der Komponentenattribute) ist verboten
6. Zugriff auf Java-API für Reflexion ist verboten, da auf diese Weise ein Aufruf von verbotenen Methoden vorgenommen werden kann, der von einer statischen Bytecodeanalyse nicht entdeckt wird

ANFORDERUNGEN AN MANUELLE GRAPHISCHE OBERFLÄCHE

Für die Implementierung der manuellen graphischen Benutzeroberfläche gilt, dass diese weder auf plattformspezifische Quellen bzw. Senken noch auf den internen Zustand des Anwendungskerns zugreift. Konkret bedeutet dies Folgendes:

1. Der Code darf nur auf den Teil der plattformspezifischen API zugreifen, der nicht als Quelle bzw. Senke privater Information genutzt werden kann
2. Zugriff auf Java-API, die nicht Teil der Android-API ist, ist verboten
3. Die Interaktion mit der Anwendungskomponente darf nur über Android-Intents erfolgen

Android-Intents sind Nachrichtenobjekte zur Inter- und Intra-App-Kommunikation

4. Da Intents u.a. auch für Android-Systemaufrufe genutzt werden, wodurch Informationslecks entstehen können, wird die direkte Nutzung dieses Android-Mechanismus verboten. Stattdessen muss der Entwickler die IFlow-Bibliothek nutzen, die lediglich die Intent-Kommunikation mit der Anwendungskomponente sowie innerhalb der manuellen Benutzeroberfläche ermöglicht, indem sie die zu versendeten Intents zur Laufzeit prüft und ggf. verwirft.

5. Zugriff auf Java-API für Reflexion ist verboten, da auf diese Weise ein Aufruf von verbotenen Methoden vorgenommen werden kann, der von einer statischen Bytecodeanalyse nicht entdeckt wird

16.2.3.2 *Analyse der manuellen Implementierung*

In IFlow wird die automatische Informationsflussanalyse des generierten Codeskeletts durch sorgfältig gewählte Modellierungsrichtlinien und Abstraktionstechniken gestützt, um dadurch falsch positive Ergebnisse zu vermeiden (vgl. u.a. Unterunterabschnitt 16.2.3.1). Manuelle Anwendungsmodule sollen hingegen komplexeres, plattformspezifisches Verhalten implementieren können, wovon auf Modellebene abstrahiert wird. Um falsch positive Ergebnisse bei der Informationsflussanalyse solcher Module zu vermeiden, müsste der Entwickler dieselbe Sorgfalt an den Tag legen, wodurch der Implementierungsaufwand erheblich steigen würde. Würden anschließend dennoch Informationsflussverletzungen gemeldet werden, so sind diese oft schwer nachzuvollziehen und zu beheben, da die Ergebnisse der Analyse sich auf den automatisch generierten Programmabhängigkeitsgraph beziehen, der sehr umfangreich und komplex ausfallen kann.

Daher wird in IFlow für die Analyse der manuellen Module die *Call Graph*-Analyse genutzt [57]. Die Knoten des Graphs stellen dabei Methodenaufrufe innerhalb der Anwendung dar, während ihre Aufrufhierarchie über gerichtete Kanten repräsentiert wird. Bei der Analyse wird der Graph durchlaufen, und dabei überprüft, ob der Methodenaufruf an der gegebenen Stelle erlaubt ist.

Bei der automatischen Codegenerierung erhält jedes manuelle Modul ein eigenes Java-Paket. Für die *Call Graph*-Analyse des Moduls wird zunächst eine Menge der Methoden M_{mm} berechnet, die in allen solchen Java-Paketen der manuellen Module Mod deklariert werden. Nicht automatisch generierte Klassen aus den Paketen der manuellen Methoden werden dabei verworfen. Anschließend wird dynamisch eine neue `main`-Methode generiert, die diese Methoden aufruft, und als Einstiegspunkt für die Berechnung des Call Graphs gewählt. Bei der Analyse werden nur die Methoden $m \in M_{mm}$ betrachtet, sowie solche, die von m aus aufgerufen werden (im Folgenden M_c). Dabei wird für jedes $c \in M_c$ der Aufrufkontext berücksichtigt, der aus ihrem Aufrufer m, sowie dem aufrufenden Modul $mod \in Mod$ besteht. Für jede Methode c wird anschließend geprüft, ob es die Anforderungen aus Unterunterabschnitt 16.2.3.1 erfüllt.

Modul mod wird hierbei benötigt, um die Klasse der Anforderungen zu bestimmen (vgl. Anforderungen an manuelle Methoden in Unterunterabschnitt 16.2.3.1 und an die manuelle graphische Oberfläche in Punkt 16.2.3.1).

Methode m bestimmt dabei, ob der Aufruf von c in ihrem Kontext zugelassen ist. Dazu wird die im Anwendungsmodell durch den «*uses*»-Stereotyp für m zugelassenen Kategorien von Quellen und Senken mit Hilfe des SuSi-Tools auf eine Menge M_a von Android- und Java-API-Methoden abgebildet (vgl. Unterabschnitt 16.1.2). Handelt es sich bei mod um die manuelle graphische Oberfläche, so gilt $M_a = \emptyset$. Ist $c \in M_a$, so ist der Aufruf erlaubt.

Für den Check der manuellen Implementierung wurde das Tool *MMChecker* entwickelt, das *Soot* [57] nutzt. Soot wird zur Bytecodeanalyse und -instrumentierung von Java- und Android-Anwendungen genutzt, und unterstützt neben Datenfluss- und Points-To-Analyse auch die in IFlow erforderliche Berechnung und Analyse von *Call-Graphen*. Die Analyse des händisch implementierten Codes mit *MMChecker* kann falsch positive Ergebnisse liefern und eine harmlose Implementierung als unsicher klassifizieren. Dies liegt daran, dass die mit SuSi berechnete Liste von kritischen API-Methoden nicht präzise ist, und einige Einträge enthält, die keine Quelle bzw. Senke darstellen. Hierfür verwaltet *MMChecker* eine Whitelist der API-Aufrufen, die von einem Experten nach sorgfältiger Untersuchung des als verboten gemeldeten Aufrufs erweitert werden kann.

16.2.4 *Benutzerzentrische Informationsflusssicherheit*

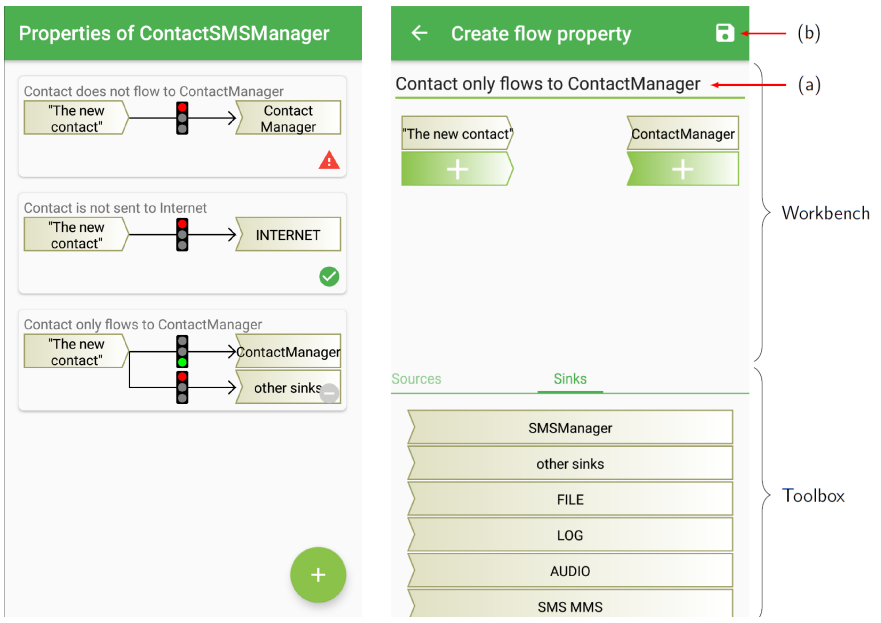
Bisher wurden Informationsflusseigenschaften vorgestellt, die vom Entwickler als Teil des Anwendungsmodells spezifiziert werden müssen. Jedoch kann es auch weitere Sicherheitseigenschaften geben, die für einen Anwendungsnutzer relevant sind, aber vom Modellierer nicht berücksichtigt worden sind. IFlow ermöglicht es dem Nutzer, seine eigenen Informationsflusseigenschaften zu spezifizieren (siehe Unterunterabschnitt 16.2.4.1). Solche Eigenschaften können anschließend in der Tool-unabhängigen Sprache *RIFL* exportiert werden, um die mit IFlow entwickelten Android-Anwendungen überprüfen zu können (siehe Unterunterabschnitt 16.2.4.2). Um die Überprüfung von intransitive Nichtinterferenz-Eigenschaften zu ermöglichen, können die Anwendungen vor dem Analyseschritt entsprechend instrumentiert werden (siehe Unterunterabschnitt 16.2.4.3). Um die sichere Benutzereingabe auf der Android-Plattform zu ermöglichen, wurde eine Reihe von Sicherheitsmechanismen implementiert (siehe Unterunterabschnitt 16.2.4.5).

16.2.4.1 *Spezifikation von Benutzereigenschaften*

Zur Spezifikation der Informationsflusseigenschaften durch den Nutzer wird dieselbe Notation wie in MODELFLOW verwendet. Um jedoch dem Benutzer die Modellierung solcher Eigenschaften mit einem UML-Editor zu ersparen, wurde hierfür ein leichtgewichtiger

Editor *MyFlows* als eine Android-Anwendung entwickelt. Abbildung 25 zeigt die Ansicht der Anwendung für die Fallstudie *ContactSMSManager* (vgl. Kapitel 22). Damit ist es möglich, Informationsflusseigenschaften in derselben Notation wie in MODELFLOW zu spezifizieren.

Die bisher benutzerspezifizierten Informationsflusseigenschaften können in einer Listenansicht eingesehen werden (siehe Abbildung 25a). Sollen neue Eigenschaften spezifiziert werden, so wird hierfür die Editor-Ansicht der Anwendung verwendet (siehe Abbildung 25b). Dort werden im Toolbox-Bereich alle anwendungs- und plattformspezifischen Quellen und Senken der jeweiligen Anwendung dargestellt, die aus ihrem Modell abgeleitet werden. Diese können dem Workbench-Bereich hinzugefügt werden, und miteinander verbunden werden. Dabei wird der Benutzer gefragt, ob der Informationsfluss zwischen der ausgewählten Quelle und Senke erlaubt oder verboten werden soll (analog zu «`allowedFlow`» und «`noFlow`» in MODELFLOW). Wie in MODELFLOW erhält die spezifizierte Eigenschaft eine Bezeichnung im Klartext (Abbildung 25b (a)), und kann anschließend auf dem Gerät gespeichert werden (Abbildung 25b (b)).



(a) Auflistung der Informationsflusseigenschaften

(b) Editor für Informatiosnflusseigenschaften

Abbildung 25: Benutzerdefinierte Informationsflusseigenschaften

16.2.4.2 *RIFL-Export*

Der Nutzer spezifiziert die Eigenschaften nicht für die komplette, verteilte IFlow-Anwendung, wie dies der Fall in MODELFLOW ist, sondern nur für eine App-Anwendungskomponente, die auf seinem Ge-

rät installiert werden soll. Dadurch kann der Nutzer sich vergewissern, dass die von ihm spezifizierte Eigenschaft tatsächlich für den Code gilt, der auf seinem Gerät ausgeführt wird.

Zur Überprüfung solcher Eigenschaften kann ein beliebiges Informationsflussanalysetool verwendet werden, das Android-Apps untersuchen kann wie etwa [2, 67]. Dazu können die Eigenschaften in der toolunabhängigen Sprache *RIFL* (RS^3 Information-Flow Specification Language [25]) zur Spezifikation von Einschränkungen des Informationsflusses in einer Java- bzw. Android-Anwendung exportiert werden.

RS^3 INFORMATION-FLOW SPECIFICATION LANGUAGE Mit RIFL können Quellen, Senken, sowie Sicherheitsdomänen festgelegt werden. Dabei können Quellen und Senken den Sicherheitsdomänen zugeordnet werden, zwischen denen die Informationsflussbeziehungen festgelegt werden. Der Informationsfluss von einer Quelle zu einer Senke ist daher dann erlaubt, wenn der Fluss zwischen den dazugehörigen Sicherheitsdomänen als erlaubt spezifiziert ist.

Die konkrete Form der Quelle und Senke hängt dabei von der verwendeten Sprache der Anwendung ab. RIFL spezifiziert in [25], wie Quellen und Senken für Java-Quellcode und Dalvik-Bytecode angegeben werden sollen. Die Spezifikation der Sicherheitsdomänen und ihrer Interferenzrelation ist dabei sprachunabhängig.

TRANSFORMATION DER BENUTZERSPEZIFIZIERTEN EIGENSCHAFTEN ZU RIFL Die Übersetzung von Quellen und Senken der benutzerspezifizierten Eigenschaften zu RIFL erfolgt analog zur Transformation der Quellen und Senken in MODELFLOW zu Annotationen des Programmabhängigkeitsgraphen, und wird in Unterunterabschnitt 16.2.2.1 erläutert. Dabei werden die Quellen und Senken im Dalvik-Format spezifiziert, da es sich bei der zu analysierenden Anwendung um eine Android-App handelt.

Eine Ausnahme stellen dabei Quellen bzw. Senken dar, die sich außerhalb der betrachteten Anwendung befinden, wie etwa eine andere Anwendungskomponente oder ihre Attribute. Da nur eine konkrete App anstatt der gesamten, verteilten Anwendung auf Informationsflussverletzungen geprüft werden soll, werden solche externen Quellen bzw. Senken auf alle ein- bzw. ausgehenden Nachrichten der betrachteten Anwendung abgebildet. Dadurch werden auch Informationsflüsse erfasst, die nicht nur direkt zwischen der betrachteten Anwendung und der externen Quelle bzw. Senke existieren, sondern auch solche, die über andere Anwendungskomponenten stattfinden. Wird etwa nur die *TravelPlanner*-App der Travel Planner-Fallstudie Kapitel 19 auf illegale Informationsflüsse zur Anwendungskomponente *Airline* geprüft, so werden dadurch auch die Flüsse an die *Tra-*

velAgency-Komponente betrachtet, die ihrerseits die erhaltenen Informationen an die *Airline* weiterleiten könnte.

16.2.4.3 *Codeinstrumentierung zur Überprüfung der modellierten intransitive Nichtinterferenz-Eigenschaften*

Um auch intransitive Nichtinterferenz-Eigenschaften des finalen Codes einer IFlow-Anwendungskomponente überprüfen zu können, wird der in Unterunterabschnitt 16.2.2.2 erläuterter Ansatz verfolgt. Dabei wird der Rumpf der Deklassifikationsmethode `decl` (über die der Informationsfluss von einer Quelle `q` zu einer Senke `s` erlaubt wird) durch eine Return-Anweisung ersetzt, die ein mit Standardwerten gefülltes Objekt zurückgibt. Anschließend kann beispielsweise mit JoDroid [67] überprüft werden, ob es in der so modifizierten Anwendungskomponente ein Informationsfluss von `q` zu `s` vorliegt, was einer transitiven Nichtinterferenz-Eigenschaft entspricht. Ist dies der Fall, so besitzt die originale Anwendungskomponente nicht die ursprüngliche intransitive Nichtinterferenz-Eigenschaft, da die Information, die von `q` zu `s` fließt, nicht notwendigerweise von `decl` gefiltert bzw. anonymisiert wird.

Diese Modifikation der Anwendungskomponente kann jedoch nicht wie in Unterunterabschnitt 16.2.2.2 beschrieben auf Quellcode-Ebene vorgenommen werden, da nun nicht das abstrakte Codeskelett, sondern die finale, lauffähige App analysiert werden soll. Stattdessen wurde ein Tool zur Instrumentierung des Bytecodes der finalen Anwendungskomponente auf Basis von *Soot* [57] entwickelt. Hierbei wird der Bytecode der App in den Zwischencode *Jimple* transformiert, in dem alle Anweisungen der vorgegebenen Deklassifikationsmethode `decl` entfernt werden. Danach wird dem nun leeren Rumpf der Methode eine Jimple-Return-Anweisung hinzugefügt, die ein leeres Objekt vom korrekten Typ erzeugt und zurückgibt, um somit den Informationsfluss von den Eingabeparametern von `decl` zu dessen Ausgabeparameter zu eliminieren. Anschließend wird der resultierende Jimple-Code in das Ursprungsvorformat zurück konvertiert, der nun auf Informationsflussverletzungen hin überprüft werden kann.

Jimple *ist eine vereinfachte Version von Java, die maximal drei Operanden pro Anweisung erlaubt [94]*

Da es sich bei der zu überprüfenden Eigenschaft wieder um eine transitive Nichtinterferenz-Eigenschaft handelt, kann diese ebenfalls im RIFL-Format exportiert werden (vgl. Unterunterabschnitt 16.2.4.2).

Dieser Ansatz darf nur für intransitive Nichtinterferenz-Eigenschaften verwendet werden, die den Informationsfluss durch Deklassifikationsmethoden erlauben, deren Sicherheit formal bewiesen wurde (vgl. Unterunterabschnitt 15.2.2.2).

16.2.4.4 *Codeinstrumentierung der Deklassifikationsmethoden zur Informationsflusskontrolle*

Trotz Anonymisierung oder Filterung der privaten Daten ist der Nutzer der Anwendung unter Umständen nicht damit einverstanden, diese Daten freizugeben. MODELFLOW erlaubt es, die Freigabe der Daten vom Benutzer explizit bestätigen zu lassen, und die Notwendigkeit dieser Bestätigung mit einer Informationsflusseigenschaft auszudrücken (vgl. Abbildung 12.2.2).

Wurde die Möglichkeit, dass der Benutzer mit der Freigabe der Information nicht einverstanden ist, bei der Modellierung der Anwendung nicht berücksichtigt, so kann diese Freigabe durch eine Codeinstrumentierung der finalen Anwendungskomponente verhindert werden. Ist durch den in Unterunterabschnitt 16.2.2.2 beschriebenen Ansatz sichergestellt worden, dass der Informationsfluss von einer Quelle q zu einer Senke s nur über eine konkrete Deklassifikationmethode `decl` vorliegt, so kann durch das Installieren der instrumentierten Anwendungskomponente (bei der `decl` lediglich Standartwerte zurückgibt) jeglicher Informationsfluss von q zu s unterbunden werden.

Dieser Ansatz entspricht einem anwendungsspezifischen Äquivalent des dynamischen Berechtigungsmechanismus, der in Android implementiert ist. Entscheidet sich der Benutzer einer Android-App gegen die Freigabe der durch Zugriffsberechtigungen geschützten, anwendungsunabhängigen Daten (wie etwa die GPS-Position des Nutzers), so erhält die Anwendung stattdessen blinde Daten (wie etwa die GPS-Position „*0, 0*“), die keinen Rückschluss auf die originale Information zulassen. Durch die beschriebene Codeinstrumentierung kann dieser Grad an Sicherheit auch für anwendungsspezifische Informationen erreicht werden. Der vorgestellte Ansatz ist zudem feingranularer als der Berechtigungsmechanismus von Android, da die geheime Information weiterhin der Anwendung zur Verfügung steht, und zu anderen Senken fließen darf.

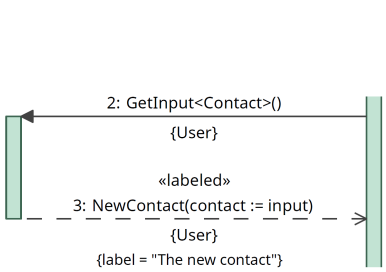
16.2.4.5 *Sicherheit bei der Benutzereingabe*

Benutzereingaben sind oft eine Quelle von geheimen Informationen wie Kreditkarten- oder Kontaktdaten. Daher ist es notwendig sicherzustellen, dass (1) der Modellierer oder der Nutzer bei der Spezifikation von Informationsflusseigenschaften auf solche Quellen verweisen kann, (2) dem Nutzer bewusst ist, welche Eigenschaften für die angeforderte Eingabe gelten, und (3) ein Angreifer nicht in der Lage ist, diese Eingabe zu stehlen.

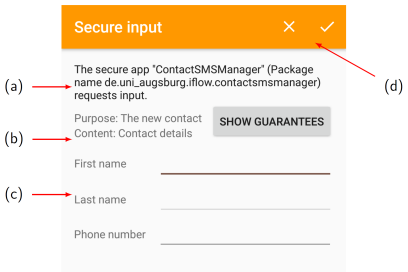
MODELFLOW erlaubt die Modellierung der Benutzereingabe als Quelle geheimer Information. Dies geschieht durch die Beschriftung der Nachrichten der vordefinierten Benutzeroberfläche mit einer Er-

läuterung ihres Inhalts (siehe auch Unterabschnitt 11.2.3). Diese Beschriftung kann bei der Modellierung von Informationsflusseigenschaften verwendet werden (siehe Tabelle 7), und wird dem Benutzer bei der Eingabeaufforderung angezeigt. Dadurch bezieht sich eine solche Eigenschaft nicht auf ein abstraktes Konzept wie ein Attribut einer Anwendungskomponente, dessen Bezeichnung nicht unbedingt dem Inhalt entsprechen muss, sonder auf eine konkrete Benutzereingabe.

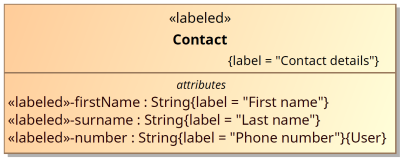
Abbildung 26a zeigt die Beschriftung einer solchen Nachricht der vordefinierten Benutzeroberfläche der *ContactSMSManager*-Fallstudie (siehe Kapitel 22 für die ausführliche Beschreibung der Anwendung), die ihren Inhalt (*„The new contact“*) im Klartext erläutert. Abbildung 26c zeigt dabei die Beschriftung des komplexen Datentyp derselben Fallstudie, der den Inhalt der einzelnen Eingabefelder wie *„First name“* oder *„Phone number“* festlegt.



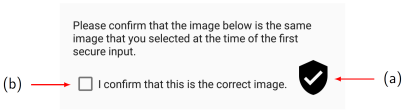
(a) Beschriftete Benutzereingabe



(b) App zur sicheren Benutzereingabe



(c) Beschriftung eines komplexen Datentyps



(d) Bestätigung der Benutzereingabe

Abbildung 26: Sichere Benutzereingabe

Um dem Nutzer einer IFlow-Anwendung anzuzeigen, welche Information von ihm angefordert wird, wurde die anwendungsübergreifende Android-App *SecureInput* zur sicheren Benutzereingabe entwickelt (vgl. Abbildung 26b), die von jeder installierten IFlow-App genutzt wird. Der eindeutige Bezeichner der aufrufenden IFlow-App (Abbildung 26b, a), die angeforderte Eingabe (Abbildung 26b, b), sowie die Beschriftungen der Eingabefelder (Abbildung 26b, c) werden automatisch aus dem Anwendungsmodell generiert. Mit *„Show guarantees“* kann die Informationsflusseigenschaft für diese Eingabe angezeigt werden (vgl. Abbildung 47).

Es gibt eine Reihe von Risiken, die mit der Benutzereingabe auf der Android-Plattform verbunden sind, und einige Angriffsvektoren wie *Phishing* ermöglichen [9]. Um das Stehlen der Benutzereingabe durch

böswillige Apps zu verhindern, wurden die folgenden Sicherheitsmechanismen implementiert:

Das geheime Symbol wird im internen Speicher der Anwendung abgelegt, auf das andere Anwendungen keinen Zugriff haben, vgl. Unterunterabschnitt 17.3.2.2

- Der Nutzer bestimmt während der Installation der Anwendung ein individuelles, geheimes Symbol, das ihm bei jeder Aufforderung zur Benutzereingabe angezeigt wird (vgl. Abbildung 26d). Da die Android-Plattform dem Nutzer keine Möglichkeit bietet, die Identität der angezeigten Anwendung zu erfahren, schützt dieser Mechanismus (inspiriert von *W3C Petname Tool* zur Identifikation vertrauter Webseiten [18]) gegen böswillige Apps, die sich als die legitime App zur sicheren Benutzereingabe ausgeben.

- Die legitime IFlow-App muss explizit bei der App zur sicheren Eingabe registriert werden. Dies kann beispielsweise bei der Installation der IFlow-App nach Bestätigung des Nutzers erfolgen. Dadurch wird sichergestellt, dass eine böswillige App nicht in der Lage ist, die App zur sicheren Eingabe aufzurufen, um den Benutzer zur Eingabe geheimer Information aufzufordern. Damit der Nutzer zwischen den Eingabeaufforderungen gleich benannter IFlow-Apps unterscheiden kann, die evtl. verschiedene Garantien für dieselben Daten bieten, wird zusätzlich zu diesen Garantien auch der eindeutige Bezeichner des Aufrufers angezeigt (vgl. Abbildung 26b, a)

- Um böswillige Apps vom Anzeigen eines transparenten Overlays über die Eingabefelder der App zur sicheren Benutzereingabe und so das Stehlen der geheimen Eingabe abzuhalten, muss der Overlay-Schutz der Android-Plattform genutzt werden (siehe Unterunterabschnitt 17.3.2.6 für Details)

17 REFINEMENT

Dieses Kapitel widmet sich der Frage, wie die formalen Beweise und Ergebnisse der Codeanalysen sich im IFlow-Ansatz auf die finale Anwendung übertragen lassen. Hierzu wird in Abschnitt 17.1 argumentiert, dass das formale Modell sowie das Codeskelett äquivalent bezüglich der in IFlow relevanten Sicherheitseigenschaften sind. Abschnitt 17.2 belegt, dass die finale Anwendung im Vergleich zum Codeskelett keine weiteren Informationsverletzungen aufweist, solange der manuell implementierte Code eine Reihe von festgelegten Kriterien erfüllt. Abschnitt 17.3 beschreibt die eingesetzten Sicherheitsmechanismen, Annahmen über die genutzten Plattformen, sowie Vorgaben für das Deployment und Betrieb der Anwendung, auf die sich die Argumentation der vorhergehender Abschnitte stützt.

17.1 ASM UND DAS CODESKELETT

Damit die Informationsflusseigenschaften, die für ein formales Modell einer IFlow-Anwendung gezeigt wurden, sich auf ihren finalen Code vererben, muss dieser Code eine Verfeinerung des formalen Modells darstellen. Dabei gilt insbesondere, dass der Code keine zusätzlichen Informationsflüsse enthält, die im formalen Modell nicht berücksichtigt wurden und die garantierten Eigenschaften verletzen. Dieser Abschnitt gibt eine Übersicht darüber, weshalb diese Verfeinerungsbeziehung zwischen dem formalen Modell und dem generierten Codeskelett besteht, während Abschnitt 17.2 die Beziehung zwischen dem Codeskelett und der finalen Anwendung im Detail diskutiert.

Bei der Transformation des Anwendungsmodell zum formalen Modell und zum Codeskelett wird sichergestellt, dass diese semantisch äquivalent sind. Eine Regel der ASM entspricht dabei einer Sequenz von funktional äquivalenten Java-Anweisungen, die den modellierten MEL*-Code implementieren. Die Reihenfolge dieser Regeln wird aus den Sequenzdiagrammen abgeleitet und durch das Setzen und Abfragen von Mailboxen und Zustandsvariablen erzwungen, während diese im Programmcode durch die Hintereinanderausführung der Java-Anweisungen sichergestellt wird.

Im Folgenden wird auf die Aspekte der Modelltransformation in IFlow eingegangen, die bezüglich der Verfeinerung zwischen dem formalen Modell und Code betrachtet werden müssen. Dabei wird u.a. argumentiert, dass der Codeskelett alle Speicherstellen des formalen Modells enthält, und dass die Regeln der ASM funktional äquiva-

lent sind zu den entsprechenden Sequenzen von Java-Anweisungen. Details zu den Transformationen des Anwendungsmodells zum formalen Modell können unter Unterabschnitt 15.2.1 eingesehen werden. Details zum Aufbau des Codeskelett sind unter Unterabschnitt 16.2.1 zu finden.

DATENTYPEN Primitive Datentypen von MODELFLOW werden im formalen Modell auf abstrakte Datentypen abgebildet, während im Java-Code sie auf semantisch äquivalente Java-Datentypen gemappt werden.

Komplexe Datentypen sind in MODELFLOW garantiert zyklenfrei (vgl. Abschnitt 10.1). Das Codeskelett ist zudem sharing-frei (verschiedene Referenzen zeigen auf verschiedene Java-Objekte, da bei Zuweisungen und Parameterübergabe eine tiefe Kopie des Java-Objekts erstellt wird), und Null-frei (vgl. Abschnitt 11.1 und Unterunterabschnitt 17.2.1.3). Die Objektstruktur ist daher ein Baum, und alle Komponentenattribute und lokale Variablen sind automatisch vorinitialisiert, so dass die Datentypen des Codeskeletts auf algebraische Datentypen abgebildet werden dürfen.

KOMPONENTEN Eine MODELFLOW-Klasse, die eine Komponente repräsentiert, wird im Codeskelett auf eine Java-Klasse abgebildet, während sie im formalen Modell auf Instanzen der Sorte *Agent* gemappt wird. Im formalen Modell wie im Code werden Service-Komponenten als Singletons betrachtet, während mobile Apps mehrfach instantiiert werden können (vgl. Unterabschnitt 15.2.1 und Unterunterabschnitt 17.2.1.5).

SPEICHERSTELLEN Komponentenattribute und lokale Variablen werden im formalen Modell auf Speicherstellen einer Komponenteninstanz abgebildet. Im Code werden Komponentenattribute als Attribute der jeweiligen Komponentenklasse repräsentiert, während lokale Variablen auf lokale Java-Variablen abegbildet werden. Beim Erstellen neuer Objekte durch das MEL*-Schlüsselwort *create* wird im Code mit `new` ein neues Java-Objekt erzeugt, während im formalen Modell ein neues Element des entsprechenden abstrakten Datentyps erzeugt wird. Der Zugriff auf Variablen und Attribute erfolgt im formalen Modell durch den Lookup der entsprechenden dynamischen Funktion; im Code entspricht dies dem Zugriff auf Variablen und Klassenattribute.

Im formalen Modell erhält jede Komponenteninstanz für jede eingehende Nachricht eine separate Mailbox. Im Code entspricht eine solche Mailbox dem formalen Eingabeparameter der Methode, die diese Nachricht verarbeitet.

Zusätzlich dazu enthält das formale Modell Zustandsvariablen, die die Reihenfolge der ASM-Regeln erzwingen. Die Zustandsvariablen

können im Anwendungsmodell weder explizit gelesen noch geschrieben werden, sondern werden automatisch aus den Sequenzdiagrammen generiert.

Im Code müssen diese Variablen nicht explizit implementiert werden, sondern entsprechen dem Befehlszähler der Java Virtual Machine. Die Reihenfolge der Java-Anweisungen, die aus dem MEL*-Code generiert werden, entsprecht dabei der modellierten Reihenfolge der MEL*-Anweisungen.

AKTIONEN Die modellierten MEL*-Anweisungen werden im formalen Modell zu Aktionen zusammengefasst, die als ASM-Regeln repräsentiert sind. Im Code werden sie zu funktional äquivalenten Java-Anweisungen transformiert, die auf dieselben Speicherstellen zugreifen wie die dazu entsprechenden ASM-Regeln. In der Regel entspricht eine ASM-Regel dem Rumpf einer automatisch generierten Java-Nachrichtenbehandlungsmethode.

Der Check der manuellen Implementierung garantiert, dass die finale Anwendung keine zusätzlichen Informationsflüsse über weitere Speicherstellen enthält, die bei der formalen Verifikation nicht berücksichtigt wurden (vgl. Unterabschnitt 16.2.3).

KOPIERSEMANTIK MEL* und das formale Modell von IFlow haben eine Kopiersemantik, während Java eine Referenzsemantik hat. Daher wird auch im generierten Codeskelett die Kopiersemantik erzwungen, indem bei jeder Zuweisung und Übergabe der Variablen als Methodenparameter eine tiefe Kopie davon erstellt wird.

KOMPONENTENKOMMUNIKATION Im formalen Modell entspricht das Versenden einer Nachricht dem Schreiben dieser Nachricht in die entsprechende Mailbox, wonach die ASM-Regel ausgeführt wird, die diese Nachricht liest und verarbeitet. Im Codeskelett entspricht dies dem Aufruf der zur ASM-Regel funktional äquivalenten Nachrichtenbehandlungsfunktion, die die zu versendende Nachricht als Methodenparameter erhält.

BENUTZERINTERAKTION UND ABLAUFABBRUCH Da der Benutzer der mobilen Anwendung diese schließen bzw. neustarten kann, simulieren die Komponenten, die im formalen Modell und im Code die Benutzeroberflächen repräsentieren, einen solchen Ablaufabbruch, wonach der Ablauf von Anfang an neugestartet wird (siehe Unterunterabschnitt 17.2.1.5 für Details). Zudem simulieren diese Komponenten auch die möglichen Interaktionen des Nutzers mit der graphischen Oberfläche, indem sie die Reihenfolge der modellierten Sequenzdiagramme permutieren.

INFORMATIONSFLUSSEIGENSCHAFTEN DES FORMALEN MODELLS UND DES CODESKELETTS Da in IFlow die modellbasierte Formalisierung der Nichtinterferenz-Eigenschaft von Rushby genutzt wird, die für abstrakte Transitionssysteme definiert wurde, während die finale IFlow-Anwendung ein Java-Programm darstellt, wurde im Rahmen des Projekts bewiesen, dass für ein System, das diese Eigenschaft hat, gilt:

$$s_1 \approx_{\mathrm{sources}(a^*,d)} s_2 \rightarrow \mathrm{run}(s_1, a^*) \approx_d \mathrm{run}(s_2, a^*) \tag{7}$$

$\mathrm{sources}(a^*, d)$ *ist nach Rushby eine Menge von Domänen, die bei einer intransitiven Nichtinterferenz-Relation in der Sequenz von Aktionen* $a*$ *die Domäne* d *interferieren dürfen*

Gleichung 7 sagt aus, dass die Ausführung einer Sequenz von Aktionen a^* in zwei verschiedenen Zuständen s_1 und s_2, die bezüglich ihrer „öffentlichen" Speicherstellen gleich sind, in neuen Zuständen resultiert, die ebenfalls bezüglich dieser Speicherstellen gleich sind. Dies entspricht der sprachbasierten Formalisierung der Nichtinterferenz-Eigenschaft für Programmcode [20, 96], wenn eine Sequenz von Aktionen als ein Programm betrachtet wird. Der Zusammenhang zwischen der PDG-Repräsentation des Java-Codes und der transitiven Nichtinterferenz-Eigenschaft nach Rushby wurde bereits in Unterabschnitt 16.1.1 erläutert.

Dies zeigt, dass die Informationsflusseigenschaften, die für das formale Modell gezeigt wurden, sich auf die Sprachebene übertragen lassen.

17.2 CODESKELETT UND DIE FINALE ANWENDUNG

Das generierte Codeskelett einer IFlow-Anwendung hat zwei Funktionen. Zum Einen wird das Skelett für die Informationsflussanalyse verwendet, und zum Anderen dient es als Basis für den Code der finalen und ausführbaren Anwendung. Daher muss sichergestellt werden, dass das Codeskelett sowohl eine geeignete Struktur besitzt, der die Analyse unterstützt, als auch nach konservativen Erweiterung und Verfeinerung lauffähig wird und die modellierte Funktionalität bietet, ohne neue Informationsflussverletzungen einzuführen. Solche Verfeinerungen sind verwirklicht durch weiteren automatisch generierten Code, IFlow-spezifische Bibliotheken, sowie die vom Entwickler bereitgestellte manuelle Implementierung. In diesem Abschnitt wird im Detail argumentiert, wie das Codeskelett und die finale Anwendung bezüglich ihrer Informationflüsse zusammenhängen.

Die Softwarearchitektur einer IFlow-Anwendung unterteilt sie in *Anwendungskomponenten*, sowie *Anwendungsmodule*, die über fest definierte Schnittstellen eingebunden werden, und im Modell unterspezifiziert sind.

In der Programmierung wird der Programmcode, der anstelle eines anderen Codes steht, als Stub bezeichnet

Das Codeskelett nutzt eine abstrakte "Stub"-Implementierung der Module. Die finale Anwendung besteht aus den Anwendungskomponenten, die anstatt der Stubs die konkrete, plattformspezifische,

und funktionale Implementierung der Module nutzt. Zudem wird bei dem Codeskelett von dem konkreten Anwendungskontext abstrahiert, indem Annahmen darüber getroffen werden, die u.a. durch plattformspezifische Sicherheitsmechanismen erfüllt werden (vgl. Abschnitt 17.3).

Dabei implementieren die abstrakten Modul-Stubs eine geeignete und konservative Überapproximation der Informationsflüsse ihrer konkreten und funktionalen Gegenstücke, wodurch die Informationsflussanalyse der Anwendung möglich wird, während der Code der Anwendungskomponenten möglichst einfach gehalten wird. Dies wird u.a. durch die Abstraktion einer verteilten Anwendung als einen monolithischen Sicherheitskernel erreicht, woraus ein Programmabhängigkeitsgraph mit möglichst wenigen Knoten und Abhängigkeitskanten erstellt werden kann, um falsch positive Ergebnisse bei der Informationsflussanalyse zu vermeiden. "Konservativ" bedeutet dabei, dass die kleinste Menge an möglichen Informationsflüssen angenommen wird, die das unterspezifizierte Modell zulässt. Dies ist beispielsweise sinnvoll bei Black-Box-Modulen wie externe Komponenten, deren Verhalten man oft nicht kontrollieren oder analysieren kann. So könnten die Daten, die ein externer Webservice erhält, u.a. auch an andere Anwendungskomponenten über nicht modellierte Übertragungskanäle weitergeleitet werden, wovon aber bei konservativer Abschätzung abstrahiert wird. "Überapproximation" bedeutet, dass alle Flüsse erfasst werden, die in der konkreten Implementierung des Moduls möglich sind. "Geeignet" ist die Approximation dann, wenn sie von der eingesetzten Informationsflusskontrolle mit möglichst wenigen Fehlwarnungen analysiert werden kann.

„Black Box" ist ein System, dessen innere Struktur nicht bekannt ist

Sind also alle Modul-Stubs *geeignete*, *konservative Überapproximationen* ihrer konkreten Gegenstücke, so weist die finale Anwendung keine weiteren Informationsflüsse auf als solche, die bei der Informationsflussanalyse des Codeskeletts gefunden wurden.

Im Folgenden werden die Techniken vorgestellt, die eine solche geeignete Abstraktion möglich machen, und anschließend wird die Verfeinerung der Module im Einzelnen diskutiert. Da derselbe Code des Anwendungskerns für Analyse und Ausführung verwendet wird, muss hier nicht weiter darauf eingegangen werden.

17.2.1 *Allgemeine Abstraktionstechniken*

17.2.1.1 *Abhängigkeit zwischen Parametern und Zustand des Moduls*

Eine der oft angewandten Techniken für das Abstrahieren von Modulen ist die Simulation des Informationsflusses zwischen den Ein- und Ausgabeparametern (und gegebenenfalls dem persistenten Zustands) eines Moduls. So kann bei einem Modul—wie etwa einer manuellen Methode, die eine Eingabe erwartet und eine Ausgabe liefert—in des-

sen abstrahierten Version eine Abhängigkeit zwischen den Ein- und Ausgabeparametern simuliert werden. Dies stellt eine Überapproximation der Informationsflüsse dar, die in der konkreten Implementierung des Moduls vorkommen können, da diese auch weniger Informationsflüsse zwischen der Ein- und Ausgabe haben kann. Dies ist beispielsweise dann der Fall, wenn die Eingabe des Moduls ein Objekt von einem komplexen Datentyp ist, und die konkrete Implementierung des Moduls nur auf eine Untermenge dessen Attribute zugreift, um die Ausgabe zu generieren.

In den folgenden Abschnitten werden die Eingabe- und Ausgabeparameter eines Moduls als die Mengen $I \subseteq \{in_1..in_n\}$ und $O \subseteq \{out_1..out_m\}$ bezeichnet. Der persistente Zustand des Moduls wird in der folgenden Pseudocode-Notation als sl zusammengefasst. Die Informationsflusssimulation von einer Menge von Variablen $in_1..in_n$ zu einer weiteren Variable out wird als $out = (in_1, .., in_n)$ notiert.

Um im Codeskelett eine solche Informationsflussabhängigkeit zwischen zwei Variablen von beliebigen MODELFLOW-Datentypen zu simulieren, stellt das IFlow-Framework in der `IFlowUtility`-Klasse die statischen Methoden `extractSecurityLevel` sowie `getTaggedType` zur Verfügung (vgl. Listing 21).

Die Methode `extractSecurityLevel` erhält als Parameter Objekte von beliebigen MODELFLOW-Datentypen, und gibt einen Integer zurück, der von allen Attributen des Objekts abhängt. Es ist für komplexe Datentypen nicht ausreichend, die Ausgabe der Methode lediglich von der Referenz auf das Objekt abhängig zu machen: ist ein als geheim annotiertes Datum im Attribut des übergebenen Objekts gespeichert, so gäbe es in dem dazugehörigen Programmabhängigkeitsgraphen keine Abhängigkeit zwischen diesem geheimen Datum und der Ausgabe der Methode. Der Quellcode der `extractSecurityLevel`-Methode für komplexe MODELFLOW-Datentypen ist in Listing 21, Zeilen 3-36 abgebildet. Die Code-Generierung stellt zudem sicher, dass komplexe Modelflow-Datentypen von der vordefinierten `SecurityLevelExtractor`-Klasse erben, damit `extractSecurityLevel` rekursiv auf alle Attribute solcher Datentypen angewandt werden kann (vgl. Listing 4).

Die in IFlow eingesetzte, code-basierte Informationsflussanalyse ist objektsensitiv (siehe [35])

Die Methode `getTaggedType` erhält als Parameter den gewünschten Datentyp sowie den Integer, der von der `extractSecurityLevel`-Methode erstellt worden ist, und gibt eine neue Instanz vom angegebenen Datentyp zurück, die von diesem Integer und somit von allen an die `extractSecurityLevel`-Methode übergebenen Objekte und ihrer Attribute abhängt. Sie wird u.a. dafür verwendet, die Ausgabeparameter der Module zu erstellen, die von allen Eingabeparametern abhängen. Der Quellcode der `getTaggedType`-Methode für komplexe Modelflow-Datentypen ist in Listing 21, Zeilen 37-64 abgebildet.

17.2.1.2 *Benutzersimulation*

IFlow-Anwendungen interagieren mit dem Nutzer, was bei der Informationsflussanalyse berücksichtigt werden muss. Der Nutzer kann Eingaben tätigen, sowie die Entscheidung treffen, welche der möglichen Interaktionsmöglichkeiten als nächste ausgeführt werden soll. Beides ist sowohl über die vordefinierte, als auch über die manuelle Benutzeroberfläche möglich.

Offensichtlich sind diese Entscheidungen eines Benutzers von einer Vielzahl von Faktoren beeinflusst. Manche dieser Faktoren sind bereits Teil des Anwendungsmodells: Beispiele dafür sind Anzeige auf dem Bildschirm bzw. das Ablegen von Daten auf einem vom Benutzer lesbaren Datenträger. Ist der Benutzer im Besitz eines Geräts, auf dem er Administrator-Rechte besitzt, so kann er sogar den internen Zustand der lokalen IFlow-Apps auslesen (siehe auch Anhang D). Solche Informationen, die für den Benutzer zugänglich sind, können seine Entscheidung beeinflussen, wodurch es zu zusätzlichen Informationsflussabhängigkeiten kommt, die selbst im Code der finalen Anwendung nicht repräsentiert sind. Diese Annahme würde jedoch jede feingranulare Informationsflussanalyse verbieten, da alle Informationen auf dem Gerät des Nutzers als gleich geheim bzw. öffentlich angesehen werden müssten. Dies kann auf Wunsch bei der Modellierung berücksichtigt werden (indem u.a. alle IFlow-Apps dieselbe Sicherheitsdomäne erhalten bzw. als Informationsflussquelle oder -senke modelliert werden), ist in MODELFLOW jedoch keine zwingende Voraussetzung.

Daher wird bei der Informationsflussanalyse für den Benutzer angenommen, dass seine Entscheidungen bezüglich der nächsten Interaktion oder der eingegebenen Daten zufällig getroffen werden. Im Folgenden wird die Benutzereingabe in der Pseudocode-Notation als u bezeichnet. Im generierten Code wird hierfür die API der Klasse `UserSimulator` der IFlow-Bibliothek verwendet, die nur vom Codeskelett verwendet wird und intern die Pseudozufallszahlgenerierung von Java nutzt. Dadurch wird bei der statischen Informationsflussanalyse sichergestellt, dass die Benutzereingabe nicht fix ist, und von keinen weiteren Informationen abhängt. Dies ist v.A. wichtig, wenn über die Benutzereingabe verzweigt wird: wäre die Benutzereingabe zur Kompilationszeit fix, so würde die statische Informationsflussanalyse des Bytecodes nur einen der beiden Fälle berücksichtigen. So würde das Programm in Listing 6 als sicher eingestuft werden, wenn die Benutzereingabesimulation für u immer den Wert `false` zurückgeben würde, da in diesem Fall die kritische Zuweisung von geheimen Daten in `high` an die öffentliche Variable `low` niemals ausgeführt werden könnte.

Listing 6: Verzweigung über die Benutzereingabe

```
1 int high;
```

```
2 int low;
3 if(u) low = high;
```

17.2.1.3 *Ausnahmebehandlung*

Die im Code ausgelösten Exceptions können zu neuen, bei der Modellierung nicht berücksichtigten Programmabläufen führen, die ihrerseits potentiell unerwünschte Informationsflüsse ermöglichen (vgl. [37]). Dazu zählen implizite Informationsflüsse, wenn die Ausnahme von einem Exception-Handler weiter oben auf dem Call-Stack abgefangen wird. Bei der Informationsflussanalyse der Programmabhängigkeitsgraphen werden daher Abhängigkeitskanten von der Anweisung, die eine Exception auslösen können, zu dem Exception-Handler eingefügt.

Um die Informationsflussanalyse zu vereinfachen, und zu garantieren, dass sich die finale Anwendung wie das formale Modell verhält, wird bei der Code-Generierung sichergestellt, dass keine der Anweisungen Exceptions auslösen kann. Die MEL*-Sprache ist dabei garantiert Null-frei (vgl. Abschnitt 11.1), während bei den Modulen darauf geachtet wird, dass diese keine Laufzeitfehler auslösen und keine Null-Referenzen zurückgeben. Dazu wird bei der konkreten Implementierung der Module darauf geachtet, dass sie alle potentiell auftretenden Laufzeitfehler intern abfangen, und im Ausnahmefall ein neues Objekt statt einer Null-Referenz zurückgeben. Alle Komponentenattribute werden zudem mit Standardwerten initialisiert. Das Auftreten eines Laufzeitfehlers und die Rückgabe eines neuen Objekts kann nur von den Eingabeparametern des Moduls sowie den plattformspezifischen Quellen abhängen, auf die das Modul zugreifen darf. Diese Informationsflussabhängigkeit zwischen den Ein- und Ausgabeparametern sowie den plattformspezifischen Quellen und Senken muss bei der IF-Simulation jedes einzelnen Moduls sichergestellt werden (siehe Unterabschnitt 17.2.2 für Details). Dadurch wird garantiert, dass der modellierte Programmfluss auch von der finalen Anwendung eingehalten wird, und alle durch Exceptions potentiell auftretenden Informationsflüsse bei der Analyse berücksichtigt werden.

17.2.1.4 *Komponentenkommunikation*

Bei IFlow-Anwendungen handelt es sich um verteilte Systeme, was bei der Informationsflussanalyse berücksichtigt werden muss. Die finale IFlow-Anwendung besteht aus einer oder mehreren Android-Anwendungspaketen (APKs) und kompilierten Play-Web-Service-Paketen, die von dem verwendeten Informationsflussanalysetool nicht als eine Gesamtanwendung analysiert werden kann. Selbst wenn man den Quellcode der Android-Apps und Webservices unverändert zu

einer Codebasis zusammenführt, würde die Informationsflussanalyse zu ungenau ausfallen. Dies liegt daran, wie die eingehenden Nachrichten von den unterliegenden Plattformen behandelt werden. Das Android-System ruft hierfür die `onCreate`-Methode der in *AndroidManifest.xml* spezifizierten Activity auf, in der der App-Entwickler die vom Nachrichteninhalt abhängige Aktion ausführt. Das Play-Framework leitet die empfangene Nachricht an die Nachrichtenbehandlungsmethode weiter, die in der `routes`-Datei des Webservices spezifiziert ist.

AndroidManifest *legt u.a. die Struktur und Schnittstellen einer Android-App fest, und wird in IFlow automatisch generiert*

routes *legt u.a. die Schnittstellen eines Play-Webservices fest, und wird in IFlow automatisch generiert*

In beiden Fällen wird beim Erhalten einer eingehenden Nachricht unabhängig von ihrem Typ oder Inhalt in jeder Komponente dieselbe Java-Methode aufgerufen. Ein vereinfachter Code-Fragment, der diesen Umstand illustriert, ist in Listing 7 abgebildet. `ComponentA` bzw. `ComponentB` repräsentieren dabei jeweils entweder eine Android-App oder einen Webservice. Die generische Methode `handleIncomingMsg` entspricht entweder der Android-Lebenszyklus-Methode `onCreate` oder der internen Play-Framework-Methode, die die Nachricht an eine spezifische Nachrichtenbehandlungsmethode des Webservices weiterleitet. Dies stellt für eine statische Codeanalyse eine Herausforderung dar, da dabei nicht zwischen Nachrichtentypen oder -inhalt unterschieden werden kann. Bei der Informationsflussanalyse resultiert diese Umstand in einer unerwünschten Überapproximation: so würde eine statische Informationsflussanalyse das in Listing 7 abgebildete Programm als unsicher einstufen (wenn `ComponentA.highSource` als geheim und `ComponentB.lowSink` als öffentlich annotiert sind), obwohl die geheimen Daten niemals öffentlicher werden.

Listing 7: Vereinfachter Java-Code der Behandlung von eingehenden Nachrichten bei Android und Play

```
1  class Msg { int type, content; }
2
3  class ComponentA {
4
5          static int highSource;
6
7          public static void sendMsg1(){
8                  Msg msg1 = new Msg();
9                  msg1.type = 1;
10                 msg1.content = highSource;
11                 ComponentB.handleIncomingMsg(msg1);
12         }
13 }
14
15 class ComponentB {
16
17         static int highSink, lowSink;
18
```

```
19          public static void handleIncomingMsg(Msg msg){
20                  if(msg.type == 1) receiveMsg1(msg);
21                  else receiveMsg2(msg);
22          }
23
24          public static void receiveMsg1(Msg msg){
25                  highSink = msg.content;
26          }
27
28          public static void receiveMsg2(Msg msg){
29                  lowSink = msg.content;
30          }
31  }
```

Daher werden im Codeskelett der IFlow-Anwendungen die für den Nachrichtentyp spezifischen Nachrichtenbehandlungsmethoden direkt aufgerufen, ohne eine generische Nachrichtenbehandlungsmethode wie `handleIncomingMsg` in Listing 7 zu verwenden. Ein vereinfachter Code-Fragment, der diesen Umstand illustriert, ist in Listing 8 abgebildet. Die generische Nachrichtenbehandlungsmethode `handleIncomingMsg` ist dennoch Teil des Codeskeletts, da sie in der finalen Anwendung beim Empfang einer Nachricht von der verwendeten Plattform automatisch aufgerufen wird. Die automatische Codegenerierung sorgt dafür, dass bei einer eingehenden Nachricht die korrekte Nachrichtenbehandlungsmethode aufgerufen wird.

Listing 8: Vereinfachter Java-Code der Behandlung von eingehenden Nachrichten im Codeskelett einer IFlow-Anwendung

```
1   class Msg { int type, content; }
2
3   class ComponentA {
4
5           static int highSource;
6
7           public static void sendMsg1(){
8                   Msg msg1 = new Msg();
9                   msg1.type = 1;
10                  msg1.content = highSource;
11                  ComponentB.receiveMsg1(msg1);
12          }
13  }
14
15  class ComponentB {
16
17          static int highSink, lowSink;
18
19          public static void receiveMsg1(Msg msg){
20                  highSink = msg.content;
21          }
22
23          public static void receiveMsg2(Msg msg){
```

```
24                  lowSink = msg.content;
25          }
26
27          public static void handleIncomingMsg(Msg_ msg_){
28                  Msg msg = IFlow.convertIncomingMsg(msg_)
29
30                  if(msg.type == 1) ComponentB.receiveMsg1(msg);
31                  else ComponentB.receiveMsg2(msg);
32          }
33  }
```

Damit das Codeskelett als Teil der finalen Anwendung verwendet werden kann, müssen diese direkten Aufrufe auf den plattformspezifischen Nachrichtenübertragungsmechanismus abgebildet werden. Dies wird durch die Verwendung von Proxy-Klassen in der finalen Anwendung gelöst. Kommunizieren im Anwendungsmodell die Komponenten *ComponentA* und *ComponentB* miteinander, so wird im finalen Quellcode von *ComponentA* (sei es eine Android-App oder ein Webservice) die Klasse, die die Komponente *ComponentB* repräsentiert, durch eine Proxy-Klasse ersetzt, die dasselbe Interface wie das Original implementiert. In Listing 9 ist der Pseudo-Code einer solchen Proxy-Klasse abgebildet, die die Kommunikationsstruktur der finalen IFlow-Anwendung skizziert. Diese Proxy-Klassen gehören zu den Artefakten der finalen Anwendung, und werden automatisch generiert (vgl. Abbildung 2)

Listing 9: Vereinfachter Java-Code einer Proxy-Klasse aus dem finalen Code einer IFlow-Anwendung

```
1   class ComponentB {
2
3           public static void receiveMsg1(Msg msg){
4                   IFlow.sendOutgoingMessage("CompontentB", msg);
5           }
6
7           public static void receiveMsg2(Msg msg){
8                   IFlow.sendOutgoingMessage("CompontentB", msg);
9           }
10
11  }
```

Diese Klasse ist zuständig für die Kodierung des Inhalts der zu versendenden Nachricht als ein JSON-String, die Konvertierung dieser Nachricht in ein plattformspezifisches Format (z.B. ein Android-Intent oder ein HTTPS-Request) und das Versenden der Nachricht an die jeweilige Zielkomponente (vgl. Listing 9, Zeilen 4 und 8).

JSON *steht für* JavaScript Object Notation, *und ist ein Standard-Datenformat zum Datenaustausch zwischen Anwendungen*

Das Empfangen von Nachrichten wird der generischen Nachrichtenbehandlungsmethode übernommen, die von der unterliegenden Plattform aufgerufen wird (vgl. Listing 8, Zeile 27). Dort wird sie in das IFlow-Format konvertiert, und an die spezifische Nachrichtenbehandlungsmethode weitergeleitet (vgl. Listing 8, Zeile 28). Dieses

Machanismus stellt die Nachrichtenreihenfolge und den modellierten Programmfluss sicher, und wird im Folgenden für jeden Komponententyp im Detail erläutert.

APP-ZU-APP-KOMMUNIKATION Die Kommunikation zwischen IFlow-Apps wird mit Hilfe von expliziten, Signatur-geschützten Android-Intents gelöst. Dadurch stellt die Android-Plattform sicher, dass die Nachrichten von keinen anderen Apps auf dem Benutzergerät abgefangen werden können (siehe Unterunterabschnitt 17.3.2.4 für Details).

Die IFlow-Nachrichten werden vor dem Versand mit JSON codiert und als ein Android-Intent verpackt.

Zum Versand der Nachricht wird die Lifecycle-Methode des Android-Frameworks `startActivityForResult` verwendet, die die Empfänger-App aufruft und ihr die Nachricht als Android-Intent übergibt (vgl. Pseudo-Code in Listing 9, Zeilen 4 und 8, bzw. Listing 27 für die konkrete Implementierung in der IFlow-Bibliothek). Der Empfänger des Intents wird aus dem Anwendungsmodell entnommen und entspricht dem Paket- und Klassennamen der modellierten Ziel-App. Damit die Antwort der Ziel-App korrekt behandelt werden kann, wird eine Callback-Methode angegeben, die das von der Ziel-App erhaltene und als JSON kodierte Intent deserialisiert und an die korrekte Nachrichtenbehanldungsmethode der Aufrufer-App weiterleitet. Diese Callback-Methode wird in der Android-Lifecycle-Methode `onActivityResult` der Aufrufer-App beim Erhalten eines Antwort-Intents ausgeführt.

Die App, die ein mit `startActivityForResult` versandtes Intent empfängt, spezifiziert in ihrer *AndroidManifest*-Datei die *Activity*, die dieses eingehende Intent mit der Standard-Funktion des Android-Frameworks `onCreate` bearbeiten soll. In IFlow-Apps ruft `onCreate` die Codeskelett-Methode `handleIncomingMsg()` auf (vgl. Pseudo-Code in Listing 8, Zeile 27), die den Inhalt des Intents deserialisiert und an die spezifische Nachrichtenbehandlungsmethode weiterleitet.

Durch die automatische Generierung der *AndroidManifest.xml*-Datei, der `handleIncomingMsg` sowie der Callback-Methoden zur Bahandlung der eingehenden Intents wird sichergestellt, dass der modellierte Nachrichtenfluss eingehalten wird und sich die finale Anwendung äquivalent zum Codeskelett verhält.

APP-ZU-GUI-KOMMUNIKATION Die Kommunikation zwischen IFlow-Apps und einer manuellen grafischen Oberfläche funktioniert analog zur Kommunikation zwischen IFlow-Apps. Die modellierten IFlow-App und ihre GUI werden als zwei Android-Activities einer Anwendung implementiert. Dabei agiert die GUI-Komponente immer als der Aufrufer der IFlow-App. Der Entwickler der GUI-Komponente nutzt die automatisch generierte Proxy-Klasse, die die auf-

zurufende IFlow-App repräsentiert, um mit dieser App zu kommunizieren.

Die Nachrichtenkodierung sowie Kommunikation über Intents verläuft wie weiter oben beschrieben, da es sich in beiden Fällen um Kommunikation zwischen Android-Activities handelt.

APP-ZU-USER-KOMMUNIKATION Bei der Kommunikation zwischen IFlow-Apps und der vordefinierten User-Komponente unterscheidet man zwischen einer in die App eingebetteten Benutzeroberfläche und einer separaten App für sichere Benutzereingabe mit annotierten Eingabefeldern. Im ersten Fall wird ein Android-Fragment in die Activity der Aufrufer-App geladen, und beim Drücken des Bestätigungsknopfes die eingegebenen Daten an die von der Aufrufer-App spezifizierte Callback-Methode übergeben, die die spezifische Nachrichtenbehandlungsbethode des Aufrufers ausführt (für Details siehe Unterunterabschnitt 17.2.2.4). Im zweiten Fall funktioniert die Inter-App-Kommunikation analog zur Kommunikation zwischen IFlow-Apps.

Ein Android-Fragment *repräsentiert einen Teil der Benutzeroberfläche einer App*

APP-/SERVICE-ZU-SERVICE-KOMMUNIKATION Kommunikation zwischen IFlow-Apps bzw. -Services und anderen IFlow-Services wird mit Hilfe von HTTPS-POST-Anfragen gelöst. Dadurch kann sichergestellt werden, dass die Nachrichten nicht abgehört oder manipuliert werden können (vgl. Unterunterabschnitt 17.3.2.5).

Die IFlow-Nachrichten werden vor dem Versand mit JSON codiert und als eine POST-Nachricht versandt (vgl. Pseudo-Code in Listing 9, Zeilen 4 und 8).

POST ist eine der in HTTP möglichen Anfragemethoden zum Übertragen von Daten

Zum Versand der Nachrichten wird die Apache-Bibliothek für HTTP-Kommunikation verwendet. Das Ziel der POST-Anfrage wird aus der *Config*-Datei ausgelesen, in der die URL des Webservices vom Entwickler angegeben ist. Analog zur Kommunikation zwischen IFlow-Apps wird auch hier eine automatisch generierte Callback-Methode angegeben, die beim Erhalten einer Antwort vom Webservice die spezifische Nachrichtenbehandlungsmethode aufruft.

Config *ist eines der automatisch generierten Artefakte der finalen Anwendung, vgl. Abbildung 2*

Der Play-Webservice, der die POST-Anfrage erhält, spezifiziert in seiner *routes*-Datei die Nachrichtenbehandlungsmethode, die diese Anfrage bearbeiten soll. Darin wird die Nachricht deserialisiert und wie im Modell spezifiziert behandelt.

Durch die automatische Generierung der *routes*-Datei sowie der Callback-Methoden zur Behandlung der eingehenden HTTPS-Antwortnachrichten wird sichergestellt, dass der modellierte Nachrichtenfluss eingehalten wird und sich die finale Anwendung äquivalent zum Codeskelett verhält.

17.2.1.5 *Programmabläufe*

IFlow-Anwendungen werden bei der Informationsflussanalyse als sequentielle Programme betrachtet. Es muss somit sichergestellt werden, dass in dem analysierten sequentiellen Programm alle Abläufe betrachtet werden, die auch in der Realität vorkommen können. Folgende Punkte müssen dabei berücksichtigt werden, die bei der Ausführung der finalen Anwendung eine Rolle spielen:

- Permutation der modellierten Programmabläufe zur Simulation beliebiger Aufrufreihenfolge durch den Nutzer
- Implizite Programmablaufabbrüche
- Neustart einer App-Komponente
- Aufruf der Webservices durch andere Systemnutzer bzw. Angreifer

PERMUTATION DER MODELLIERTEN PROGRAMMABLÄUFE In MODELFLOW können mehrere Programmabläufe modelliert werden (siehe Unterabschnitt 11.2.5). Es muss daher sichergestellt werden, dass jede in der Realität mögliche Reihenfolge dieser Abläufe vom sequentiellen Programm erfasst wird.

Solche Programmabläufe werden von der manuellen grafischen Benutzeroberfläche angestoßen, deren Verhalten bis auf den Versand und das Erhalten von Nachrichten nicht weiter spezifiziert ist. Somit ist die Reihenfolge, in der die Programmabläufe in der finalen Anwendung ausgeführt werden können, von der manuellen Implementierung dieser Benutzeroberfläche sowie von der Benutzereingabe abhängig. Dieser Umstand wird in der Stub-Implementierung der manuellen Benutzeroberfläche durch Abbildung aller möglichen Permutationen der modellierten Programmabläufe simuliert, und wird in Unterunterabschnitt 17.2.2.5 im Detail erläutert.

IMPLIZITE PROGRAMMABLAUFABBRÜCHE Die modellierten Programmabläufe sind in der finalen Anwendung nicht atomar; wird der Benutzer aufgefordert, mit der vordefinierten Benutzeroberfläche zu interagieren, so kann er den Programmablauf abbrechen, indem er den „Back-“ bzw. „Home-Button“ auf seinem mobilen Gerät betätigt. Solche impliziten Abbrüche der Programmabläufe werden durch die Stub-Implementierung der vordefinierten Benutzeroberfläche abgebildet. Hierfür wird abhängig von der simulierten Benutzerabgabe der aktuelle Programmablauf entweder fortgeführt oder zurückgesetzt, was in Unterunterabschnitt 17.2.2.4 im Detail beschrieben wird.

NEUSTART EINER APP-KOMPONENTE Neben einem vom Benutzer initiierten Abbruch und Neustart eines modellierten Programma-

blaufs kann die gesamte App vom Benutzer (oder sogar dem Betriebssystem, falls die Systemressourcen des mobilen Geräts knapp werden komplett geschlossen und zu einem späteren Zeitpunkt wieder geöffnet werden. Eine App-Komponente unterscheidet sich somit von einem Webservice, der im regulären Betrieb weder vom Nutzer noch vom Betriebssystem neugestartet wird.

Im Codeskelett entspricht dies einem impliziten Programmablaufabbruch, wonach ein beliebiger weiterer modellierter Programmablauf stattfinden kann, was bereits in den vorherigen Abschnitten beschrieben wurde. Zusätzlich ist es notwendig, dass der interne Zustand der finalen Anwendung vor und nach dem Neustart einer App gleich bleibt, da jede Komponente im analysierten Codeskelett als ein Singleton mit einem persistenten Zustand abgebildet wird (vgl. Unterunterabschnitt 17.2.1.4). Dies wird für jede App sichergestellt, indem der Inhalt jedes modellierten Komponentenattributs mittels der Android `SharedPreferences`-API bei jedem Schließen und Öffnen der Anwendung im Dateisystem persistent ablegt bzw. daraus wiederherstellt wird. Diese Funktionen werden bei jedem Schließen bzw. Neustart der App mittels der Android `Lifecycle`-API aufgerufen, wodurch garantiert wird, dass der interne Zustand jeder IFlow-App persistent ist.

AUFRUF DER WEBSERVICES DURCH ANDERE SYSTEMNUTZER Im Gegensatz zu IFlow-Apps, die gegen den externen Aufruf durch andere Apps geschützt sind (siehe Unterunterabschnitt 17.3.2.4 für Details), können andere Systemnutzer jederzeit die Schnittstellen eines IFlow-Webservices aufrufen. Dabei kann es sich sowohl um andere legitime Systemnutzer handeln, aber auch um einen Angreifer, der sich nicht an die modellierten Programmabläufe halten muss. Dieser Umstand sowie dessen Folgen für die möglichen Informationsflüsse sind in Unterabschnitt 12.2.4 im Detail diskutiert worden.

Um die nicht explizit modellierten Aufrufe der Webservices im Codeskelett zu simulieren, und gleichzeitig den Informationsfluss zwischen Systemnutzern zu entdecken, ist es naheliegend, solche Benutzer als Instanzen von Klassen abzubilden, die in MODELFLOW mobile Apps repräsentieren.

Die statische Informationsflussanalyse von JOANA ist jedoch nicht in der Lage, dynamisch angelegte Objekte für die Informationsflussanalyse als geheim bzw. öffentlich zu annotieren, weshalb Eigenschaften wie „Es gibt keinen Informationsfluss zwischen verschiedenen Systemnutzern“ mit einem solchen Ansatz nicht garantiert werden können. Zudem können öffentliche Webservice-Schnittstellen auch von potentiellen Angreifern angesprochen werden, die sich nicht an das modellierte Verhalten halten müssen.

Stattdessen wird wie in Unterabschnitt 12.2.4 beschrieben der konkrete, legitime Nutzer von allen anderen Nutzern und Angreifern

unterschieden, die im Codeskelett zu einer Einheit zusammengefasst werden.

Im Folgenden wird diese Einheit *OtherUser* genannt, und wird durch die Codeskelett-Klasse `OtherUser` abgebildet (vgl. Pseudo-Code dieser Klasse in Listing 10). Sie erhält einen gemeinsamen Speicher sl, wodurch simuliert wird wird, dass die Nutzer und Angreifer innerhalb dieser Einheit miteinander kooperieren und untereinander Informationen austauschen können.

Seien ws_1`(var payload)..`ws_n`(var payload)` die Nachrichtenbehandlungsmethoden aller modellierten Webservices, wobei `payload` die übertragenen Nachrichten repräsentieren. Das Codeskelett emuliert eine beliebige Anzahl von solchen Webservice-Aufrufen in einer beliebigen Reihenfolge, indem der Code der `OtherUser`-Klasse die Nachrichtenbehanldungsmethoden dieser Services aufruft (vgl. Zeilen 6-8 in Listing 10). Die Reihenfolge und Anzahl dieser Aufrufe hängt dabei von der emulierten Benutzereingabe u ab (vgl. Zeilen 4-8), während der Nachrichteninhalt `payload` vom internen Speicher sl der `OtherUser`-Einheit sowie der zufälligen Benutzereingabe abgeleitet wird (vgl. Zeile 5). Die Antwort des Webservices wird durch die Nachrichtenbehandlungsmethoden in_1`(var in)..`in_n`(var in)` (vgl. Zeilen 11-13) verarbeitet, wobei die erhaltene Nachricht `in` in den internen Speicher sl einfließt. Dadurch hängt jede von `OtherUser` versandte Nachricht von jeder von den Webservices erhaltenen Antwortnachrichten ab, womit alle in der Realität möglichen Informationsflüsse innerhalb der Gruppe der legitimen Nutzer und Angreifer überapproximiert werden (unter der Annahme, dass keine weiteren Informationsflüsse außerhalb des modellierten Systems wie etwa durch *Social Engineering* stattfinden; siehe Beschreibung des Anwendungskontextes in Abschnitt 17.3).

Listing 10: Pseudocode der IF-Simulation eines beliebigen Webserviceaufrufers

```
1  class OtherUser {
2          var sl;
3          public init(){
4                  while(u){
5                          payload = (sl,u);
6                          if(u == 1) ws_1(payload);
7                          ⋮
8                          if(u == n) ws_n(payload);
9                  }
10         }
11         public in_1(var in){ sl = (sl,in); }
12         ⋮
13         public in_n(var in){ sl = (sl,in); }
14 }
```

Um zu simulieren, dass *OtherUser* zum beliebigen Zeitpunkt mit den modellierten Webservices interagieren kann, werden die Programmabläufe des konkreten, legitimen Nutzers mit den Aktionen von *OtherUser* permutiert. Dies geschieht durch den aufeinanderfolgende Aufrufe der `init()`-Methoden von `OtherUser` sowie der manuellen Benutzeroberfläche `Gui` (siehe Unterunterabschnitt 17.2.2.5 für Spezifikation ihrer Simulation), was durch Pseudocode in Listing 11 veranschaulicht wird. Da jede dieser Methoden eine beliebige Anzahl der Programmabläufe bzw. Webserviceaufrufe in einer beliebigen Reihenfolge implementiert, und mehrere Aufrufe desselben Webservices innerhalb eines modellierten Programmablaufs von den Modellierungsrichtlinien ausgeschlossen sind (siehe Unterabschnitt 11.2.6), werden bei der Informationsflussanalyse alle möglichen Permutationen solcher Abläufe bzw. Aufrufe berücksichtigt. Um in der realen Anwendung Interweaving und Nebenläufigkeit der Webservices auszuschließen, ist die Webservicekommunikation als synchron, und ihre Nachrichtenbehandlungsmethoden als synchronisiert implementiert.

Enthält die Anwendung keine manuelle Oberfläche, so erhält eine Anwendungskomponente eine `init`-Methode, die den einzigen modellierten Programmablauf anstößt

Listing 11: Pseudocode der IF-Simulation aller möglichen Programmabläufe

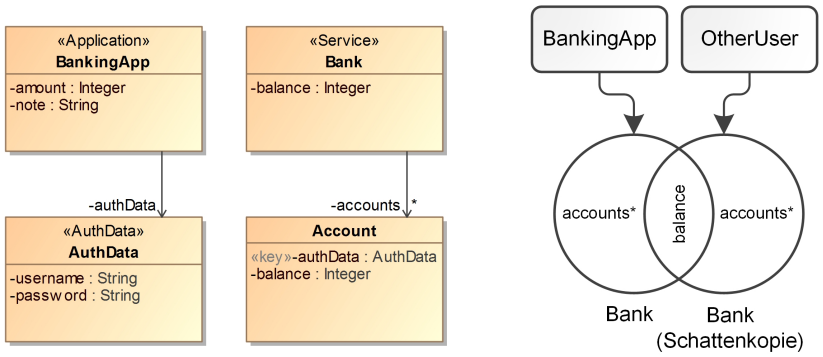
```
1 class Main {
2         public main(){
3                 while(u){
4                         Gui.init();
5                         OtherUser.init();
6                 }
7         }
8 }
```

In Abschnitt 10.4 wurde beschrieben, wie Zugriff auf bestimmte Speicherbereiche eines Webservices (modelliert und implementiert als Einträge einer Map) nur authentifizierten Nutzern erlaubt wird. Auf Implementierungsebene wird also sichergestellt, dass ein Angreifer (bzw. ein weiterer legitimer Nutzer) auf den Map-Eintrag eines legitimen Systemnutzers nicht zugreifen darf (so wie beispielsweise der individuelle Kontostand eines Nutzers anderen Systemnutzern in der Banking-Fallstudie nicht bekannt wird, vgl. Kapitel 20). Auch dieser Umstand muss bei Informationsflusssimulation des *OtherUser* berücksichtigt werden, um *false positives* bei der Analyse des Informationsflusses zwischen Systemnutzern zu vermeiden, da eine statische Codeanalyse nicht zwischen Zugriffen auf benutzerspezifische Map-Einträge unterscheiden kann. Hierzu wird von jeder Klasse, die einen Webservice mit einer solchen Map repräsentiert, eine *Schattenkopie* erstellt. Diese Schattenkopie implementiert dieselben Webserviceschnittstellen und Verhalten wie das Original, greift jedoch bei nicht zugriffgeschützten Webservice-Attributen auf die Attribute des Originals zu. Bei Map-Attributen, deren Einträge zugriffsgeschützt sind, greift die Schattenkopie stattdessen auf eigene Komponente-

nattribute zu. `OtherUser` kommuniziert dabei mit den Schattenkopien der Webservices, so dass die Webserviceaufrufe $\texttt{in}_1\texttt{(var in)}..\texttt{in}_n\texttt{(var in)}$ in Listing 10 sich auf diese Schattenkopien beziehen.

Siehe Kapitel 20 für eine Beschreibung der Fallstudie

Abbildung 27 illustriert diesen Mechanismus anhand der *BankingApp*-Fallstudie. Abbildung 27a zeigt die Anwendungskomponente *Bank*, die eine zugriffgeschützte Map *accounts*, sowie ein nicht zugriffgeschütztes Attribut *balance* enthält. Um die Nichtinterferenz zwischen der Anwendungskomponente *BankingApp* und *OtherUser* zu zeigen, wird in Abbildung 27b eine Schattenkopie von *Bank* erstellt, mit der *OtherUser* interagiert. Diese Schattenkopie enthält eine eigene Kopie der Map *accounts*, teilt sich jedoch das Attribut *balance* mit der Original-*Bank*.



(a) Komponenten der *BankingApp*-Fallstudie (b) Schattenkopie der *Bank*-Komponente

Abbildung 27: Abstraktion zugriffgeschützter Maps für Informationsflusskontrolle

Auf diese Weise wird abgebildet, dass Nutzer in der Gruppe *OtherUser* mit denselben Webservices und ihren Attributen wie der konkrete, legitime Nutzer interagiert (da die Schattenkopien der Webservices dasselbe modellierte Verhalten wie die Originale implementieren), jedoch keinen Zugriff auf die zugriffsgeschützten Speicherbereiche des legitimen Nutzers zugreifen können (da *OtherUser* und der legitime Nutzer dabei auf verschiedene, voneinander unabhängige Speicherstellen zugreifen). Um zu garantieren, dass die Zugriffskontrolle ohne Kenntnis der Authentifikationsdaten des legitimen Nutzers nicht umgangen werden kann (etwa durch Versand des gesamten Map-Inhalts als eine Nachricht), da der resultierende Informationsfluss durch den vorgestellten Analyseansatz nicht entdeckt werden könnte, wird auf Modellierungsebene der Zugriff auf solche Map-Attribute stark eingeschränkt (siehe Unterabschnitt 11.2.6).

17.2.2 *Anwendungsmodule*

Im Folgenden werden die Module im Einzelnen diskutiert. Zunächst wird die Funktionalität des Moduls kurz beschrieben, und dessen Interaktion mit der Umgebung spezifiziert. Danach werden die Annahmen über die konkrete Implementierung des Moduls aufgezählt, und im Anschluss die IF-Simulation des Moduls gezeigt, die diese Annahmen im Codeskelett implementiert. Abschließend wird diskutiert, wie die konkrete Implementierung diese Annahmen erfüllt.

Bei allen Modulen werden nur die relevanten Informationsflüsse zwischen ihren Ein- und Ausgabeparametern ($\{in_1..in_n\} = M \cap C$ und $\{out_1..out_m\} = M \cap C$) sowie plattformspezifischen Quellen Q und Senken S *innerhalb des Moduls* betrachtet. Dabei ist M die Menge aller Speicherbereiche, auf die das Modul zugreift, und C die Menge der Speicherbereiche, auf die die Anwendungskomponenten zugreifen. Informationsflüsse von und zu lokalen Variablen und als nicht kritisch angesehenen plattformspezifischen API-Aufrufen werden nicht speziell betrachtet, da dadurch allein keine unerwünschten Informationsflüsse verursacht werden können. Es wird zudem zwischen zustandsbehafteten Modulen (das Modul kann Daten zwischen Aufrufen zwischenspeichern) und zustandslosen Modulen unterschieden.

In folgenden Abschnitten werden die Informationsflussabhängigkeiten zwischen Methodenparametern bzw. Klassenattributen in Modulabstraktionen und ihren konkreten Implementierungen im Sinne der Vorgängerrelation $\rightarrow^*$ in den Programmabhängigkeitsgraphen ausgedrückt, die solche Abstraktionen und Implementierungen abbilden. So bedeutet etwa $in \rightarrow^* out$ für die formalen Eingabeparameter in und out einer Methode, dass es im PDG der Anwendung zwischen jedem Knoten, der in repräsentiert, und jedem Knoten, der out repräsentiert, einen Pfad gibt. Somit hängt out von in ab, und in beeinflusst out. Handelt es sich dabei um Objekte von einem komplexen Datentyp, so bezieht sich diese Abhängigkeit auch auf alle ihre Attribute.

17.2.2.1 *Manuelle Methoden*

KURZBESCHREIBUNG Manuelle Methoden sind im Kontext des IFlow-Ansatzes solche Methoden, die vom Entwickler von Hand programmiert werden müssen. Damit können unter Anderem Algorithmen oder Zugriffe auf plattformspezifische Quellen und Senken implementiert werden.

Auf Modellebene ist das Modul die Klassenoperation mit dem «*manual*»-Stereotyp. Jede manuelle Methode wird zu einer gleichnamigen Methode dieser Klasse transformiert, die eine statische `execute(...)`-Methode aufruft (vgl. Listing 12, Zeilen 2-4). Diese wird den manuell implementierten Rumpf des Moduls enthalten. Die

Ein- und Ausgabeparameter der Java-Methode entsprechen den modellierten Ein- und Ausgabeparametern der manuellen Methode in MODELFLOW.

INTERAKTION DES MODULS MIT DEN ANWENDUNGSKOMPONENTEN Im Modell werden manuelle Methoden aus Sequenz- oder Aktivitätsdiagrammen aufgerufen. Die Anwendungskomponenten interagieren mit einer manuellen Methode durch das Schreiben ihrer Eingabeparameter `in1..inn`, die im Code den formalen Eingabeparametern der Methode entsprechen, und das Lesen ihres Rückgabewertes `out`, der im Code dem Rückgabewert der Methode entspricht. Dabei ist n die Anzahl der modellierten Parameter der manuellen Methode. Ihrerseits kann die Methode intern mit der unterliegenden Plattform über die im Modell über das *«uses»*-Stereotyp erlaubten, plattformspezifischen Quellen Q_e und Senken S_e kommunizieren.

ANNAHMEN ÜBER DIE KONKRETE IMPLEMENTIERUNG Für die konkrete Implementierung des Moduls wird Folgendes angenommen:

1. Das Modul interagiert mit den Anwendungskomponenten nur über seine formalen Ein- und Ausgabeparameter:

   $M \cap C = \{in_1..in_n\} \cup \{out\}$

2. Das Modul greift nicht auf die im Modell verbotenen plattformspezifischen Quellen und Senken zu:

   $M \cap Q = Q_e$

   $M \cap S = S_e$

3. Alle Eingabeparameter des Moduls beeinflussen die erlaubten, plattformspezifischen Senken:

   $\forall_{q \in Q_e, x \in \{1..n\}} in_x \rightarrow^* q$

4. Der Ausgabeparameter des Moduls ist von allen erlaubten plattformspezifischen Quellen beeinflusst:

   $\forall_{s \in S_e} s \rightarrow^* out$

5. Alle erlaubten plattformspezifischen Quellen beeinflussen alle erlaubten plattformspezifischen Senken:

   $\forall_{q \in Q_e, s \in S_e} q \rightarrow^* s$

6. Alle Eingabeparameter des Moduls beeinflussen dessen Ausgabeparameter:

   $\forall_{x \in \{1..n\}} in_x \rightarrow^* out$

7. Das Modul ist zustandslos

IF-SIMULATION DES MODULS IM MODUL-STUB Der automatisch generierte Stub der Methode (siehe Listing 12) simuliert den Informationsfluss von allen Eingabeparametern zu dem Ausgabeparameter mit Hilfe der dafür in der IFlow-Bibliothek vorgesehenen Funktionen (vgl. Unterunterabschnitt 17.2.1.1), wodurch die Annahme 6 abgebildet wird. Von plattformspezifischen Quellen und Senken wird abstrahiert; stattdessen wird davon ausgegangen, dass es einen IF von allen Eingabeparametern zu allen spezifizierten Senken, sowie von allen spezifizierten Quellen zum Ausgabeparameter gibt (Annahmen 3-5). Dies wird bei der Überprüfung der modellierten Informationsflusseigenschaften berücksichtigt. Annahmen 1, 2, und 7 sind trivial erfüllt.

Listing 12: Pseudocode der IF-Simulation einer manuellen Methode

```
1 class MM {
2         public static var execute(var in_1, .., var in_n){
3                 return out = (in_1,..,in_n);
4         }
5 }
```

Kann man zeigen, dass die getroffenen Annahmen über die manuelle Implementierung des Moduls korrekt sind, so gilt Folgendes:

- Die IF-Simulation dieses Moduls ist eine Überapproximation, da alle relevanten Informationsflüsse, die in der konkreten Implementierung verkommenen können, erfasst werden.
- Die Simulation ist geeignet, da hierfür speziell für die IF-Analyse entwickelte Framework-Methoden genutzt werden (vgl. Unterunterabschnitt 17.2.1.1).
- Die Simulation ist konservativ, da von potentiell möglichen Informationsflüssen außerhalb der entwickelten Anwendung zwischen plattformspezifischen Quellen und Senken abstrahiert wird, damit die Ergebnisse der Informationsflussanalyse aussagekräftig bleiben. Dazu zählt beispielsweise das Szenario, bei dem der Nutzer die Daten, die eine manuelle Methode per SMS versandt hat, in eine Datei auf seinem Gerät abspeichert, auf die eine andere manuelle Methode zugreift.

Als „relevant“ werden Informationsflüsse von und zu Ein- und Ausgabeparametern sowie plattformspezifischen Quellen und Senken bezeichnet

ÜBERPRÜFUNG DER ANNAHMEN Mit Hilfe einer Call-Graph-Analyse wird sichergestellt, dass die konkrete Implementierung nur auf erlaubte plattformspezifische Quellen und Senken, und nie auf den Anwendungskern zugreift (Annahmen 1 und 2, vgl. Unterabschnitt 16.2.3). Zudem wird dadurch auch garantiert, dass das Modul zustandslos ist (Annahme 7). Annahmen 3-6 beschreiben bereits eine Überapproximation aller möglichen relvanten Informationsflüsse in der konkreten Implementierung, und müssen daher nicht überprüft werden.

Java-Exceptions, die innerhalb der `execute`-Methode aufgrund von Programmierfehlern oder ungeeigneten Eingabedaten geworfen werden könnten, werden von einer automatisch generierten Wrapper-Methode abgefangen. Wird eine Exception geworfen, wird an den Anwendungskern ein leeres Objekt zurückgegeben (vgl. Listing 12, Zeilen 3-7 sowie Unterunterabschnitt 17.2.1.3).

Listing 13: Aufruf der manuellen Methode aus dem Anwendungskern

```
1 class IFlowApp extends IFlowApplication {
2 ⋮
3         public var mm(var in_1, .., var in_n){
4                 try { out = MM.execute(in_1, .., in_n); }
5                 catch{ out = new var; }
6                 return out;
7         }
8 ⋮
9 }
```

17.2.2.2 *Vordefinierte Module*

KURZBESCHREIBUNG Modelflow stellt dem Modellierer eine Reihe von vordefinierten Klassen und Methoden zur Verfügung, die beispielsweise zum Auslesen der aktuellen GPS-Position verwendet werden können. Diese sind in Unterabschnitt 9.2.4 aufgelistet. Kryptographische Funktionen werden im nächsten Abschnitt gesondert behandelt.

Auf Modellebene ist das Modul eine Methode einer vordefinierten Klasse. Auf Codeebene ist das Modul eine Methode einer vordefinierten Java-Klasse der IFlow-Bibliothek. Die Ein- und Ausgabeparameter der Methoden dieser Klasse entsprechen den modellierten Ein- und Ausgabeparametern der Operationen der vordefinierten Klassen.

INTERAKTION DES MODULS MIT DEM ANWENDUNGSKERN Im Modell werden die Methoden der vordefinierten Klassen aus Sequenz- oder Aktivitätsdiagrammen aufgerufen. Der Anwendungskern interagiert mit dem Modul durch das Schreiben der Eingabeparameter I der (modellierten) Klassenoperationen, die im Code den Eingabeparametern der jeweiligen Methoden entsprechen, und das Lesen ihrer Rückgabewerte O, die im Code den Rückgabewerten dieser Methoden entsprechen. Die Aufrufe der vordefinierten, statischen Methoden `startGPSTracking` und `stopGPSTracking` innerhalb einer IFlow-App werden in diesem Abschnitt als ein Modul betrachtet, da sie intern einen gemeinsamen Zustand teilen.

Ihrerseits kann jede Methode der vordefinierten Klasse intern mit der unterliegenden Plattform über die im Modell über das *«uses»*-

Stereotyp erlaubten, plattformspezifischen Quellen Q_e und Senken S_e kommunizieren.

ANNAHMEN ÜBER DIE KONKRETE IMPLEMENTIERUNG Für die konkrete Implementierung des Moduls wird Folgendes angenommen:

1. Das Modul interagiert mit dem Anwendungskern nur über dessen Ein- und Ausgabeparameter:

   $M \cap C = I \cup O$

2. Das Modul greift nicht auf die im Modell verbotenen plattformspezifischen Quellen und Senken zu:

   $M \cap Q = Q_e$

   $M \cap S = S_e$

3. Alle Eingabeparameter des Moduls beeinflussen die erlaubten, plattformspezifischen Senken:

   $\forall_{q \in Q_e, in \in I} in \rightarrow^* q$

4. Die Ausgabeparameter des Moduls sind von allen erlaubten plattformspezifischen Quellen beeinflusst:

   $\forall_{s \in S_e} s \rightarrow^* out$

5. Alle erlaubten plattformspezifischen Quellen beeinflussen alle erlaubten plattformspezifischen Senken:

   $\forall_{q \in Q_e, s \in S_e} q \rightarrow^* s$

6. Das Modul ist zustandsbehaftet und hat den persistenten Zustand $sl \in M$.

7. Alle Eingabeparameter des Moduls sowie die erlaubten, plattformspezifischen Quellen beeinflussen den Zustand, und der Zustand beeinflusst die Ausgabeparameter sowie die erlaubten, plattformspezifischen Senken.

   $\forall_{q \in Q_e, s \in S_e, in \in I, out \in O} in \rightarrow^* sl \wedge q \rightarrow^* sl \wedge sl \rightarrow^* out \wedge sl \rightarrow^* s$

IF-SIMULATION DES MODULS IM MODUL-STUB Jede vordefinierte Klasse bzw. Methode ist Teil der IFlow-Bibliothek. Der Pseudocode der IF-Simulation einer vordefinierten Klasse `Pc` mit den vordefinierten Methoden `pm1` und `pm2` ist in Listing 14 abgebildet. Darin wird der Informationsfluss von allen Eingabeparametern $\{in_1..in_n\} = I$ zum persistenten Zustand sl, und davon zu den Ausgabeparametern $\{out_1, out_2\} = O$ mit Hilfe der dafür in der Bibliothek vorgesehenen Funktionen (vgl. Unterunterabschnitt 17.2.1.1) simuliert, wodurch die Annahmen 6-7 erfüllt wird. Von plattformspezifischen Quellen und Senken wird abstrahiert; stattdessen wird davon ausgegangen, dass

es einen IF von allen Eingabeparametern zu allen spezifizierten Senken, sowie von allen spezifizierten Quellen zum Ausgabeparameter gibt (Annahmen 3-5). Dies wird bei der Überprüfung der modellierten Informationsflusseigenschaften berücksichtigt. Annahmen 1 und 2 sind trivial erfüllt.

Listing 14: Pseudocode der IF-Simulation einer zustandsbehafteten vordefinierten Methode

```
1  class Pc {
2          private sl;
3          public var pm1(var in_1, .., var in_m){
4                  sl = (sl,in_1,..,in_m);
5                  return out_1 = (in_1,..,in_m, sl);
6          }
7          public var pm2(var in_m+1, .., var in_n){
8                  sl = (sl,in_m+1,..,in_n);
9                  return out_2 = (in_m+1,..,in_n, sl);
10         }
11 }
```

Kann man zeigen, dass die getroffenen Annahmen über die manuelle Implementierung des Moduls korrekt sind, so gilt Folgendes:

- Die IF-Simulation dieses Moduls ist eine Überapproximation, da alle relevanten Informationsflüsse, die in der konkreten Implementierung verkommenen können, erfasst werden.
- Die Simulation ist geeignet, da hierfür speziell für die IF-Analyse entwickelte Methoden der IFlow-Bibliothek genutzt werden (vgl. Unterunterabschnitt 17.2.1.1).
- Die Simulation ist konservativ, da von potentiell möglichen Informationsflüssen außerhalb der entwickelten Anwendung zwischen plattformspezifischen Quellen und Senken abstrahiert wird, damit die Ergebnisse der Informationsflussanalyse aussagekräftig bleiben.

ÜBERPRÜFUNG DER ANNAHMEN Hier sollen die aufgezählten Annahmen anhand der konkreten Implementierung der `start`- und `stopGPSTracking`-Methoden diskutiert werden, die in Listing 28 abgebildet sind. Die anderen vordefinierten Module sind analog implementiert. Im Gegensatz zu der modellierten `startGPSTracking`-Methode hat die konkrete Implementierung einen Eingabeparameter, der einen Anwendungskontext entgegennimmt. Durch die automatische Codegenerierung wird garantiert, dass beim Aufruf dieser Methode nur der aktuelle Anwendungskontext (also die IFlow-App, die die Methode aufruft) übergeben wird. Die Annahme 1 wird trotz der Übergabe des Anwendungskontextes (und somit einer Referenz auf den Anwendungkern) dennoch erfüllt, da dieser lediglich zur Inter-

aktion mit der Android-Plattform verwendet wird, um die aktuelle GPS-Position abzufragen (siehe Zeile 51).

Im Modell dürfen die Methoden nur auf den Standort des Geräts zugreifen, der nur die Ausgabe des Moduls beeinflusst, was von der konkreten Implementierung ebenfalls erfüllt wird (Annahmen 2-5, siehe Zeilen 51 und 35). Schreiben in das Log und Anzeigen von Benachrichtigungen wird als nicht relevant angesehen, da diese Benachrichtigungen nur vom Gerätenutzer gesehen werden, und man nur mit *root*-Rechten bzw. über das Debug-Interface auf die Logs zugreifen kann (vgl. Unterabschnitt 17.3.2).

Der persistente Zustand ist implementiert als die Singleton-Instanz der `GPSSensor`-Klasse, die die aufgezeichneten Ortsangaben speichert (Annahmen 6 und 7, vgl. Zeile 32).

17.2.2.3 *Kryptographie*

KURZBESCHREIBUNG IFlow bietet dem Modellierer die Möglichkeit, Kryptographieoperationen in seiner IFlow-Anwendung zu nutzen. Aus der Sicht des Informationsflusses sind hierbei die Ver- und Entschlüsselungsoperationen und ihre Abstraktionen besonders interessant, die im Folgenden beschrieben werden.

Auf Modellebene besteht das Modul aus einer Instanz der vordefinierten Klasse *Decryptor*, sowie einer Instanz der vordefinierten Klasse *Decryptor*, die aus der Instanz von *Decryptor* mit Hilfe der Operation *getEncryptor* erstellt wurde (siehe Unterabschnitt 9.2.4 für Details). Im Anwendungscode werden diese durch Instanzen der gleichnamige Klassen aus der IFlow-Bibliothek repräsentiert.

INTERAKTION DES MODULS MIT DEM ANWENDUNGSKERN Im Modell werden Operationen der vordefinierten Klassen *Encryptor* und *Decryptor* aus Sequenz- oder Aktivitätsdiagrammen aufgerufen. Die Anwendungskomponenten interagieren mit dem Modul durch das Schreiben ihres Eingabeparameters m_{in}, der im Code dem formalen Eingabeparameter der Methode *encrypt* entspricht, sowie das Schreiben ihres Eingabeparameters c_{in}, der im Code dem formalen Eingabeparameter der Methode *decrypt* entspricht. Zudem lesen die Anwednungskomponenten den Rückgabewert m_{out} der Methode *decrypt*, sowie den Rückgabewert c_{out} der Methode *encrypt*, die im Code den Rückgabewerten der gleichnamigen Methoden entsprechen.

ANNAHMEN ÜBER DIE KONKRETE IMPLEMENTIERUNG Für die konkrete Implementierung des Moduls wird Folgendes angenommen:

1. Das Modul interagiert mit dem Anwendungskern nur über dessen Ein- und Ausgabeparameter:

   $M \cap C = \{c_{in}, c_{out}, m_{in}, m_{out}\}$

2. Das Modul greift auf keine plattformspezifischen Quellen und Senken zu:

   $M \cap Q = \emptyset$

   $M \cap S = \emptyset$

3. Das Modul ist zustandsbehaftet und hat den persistenten Zustand $sl \in M$.

4. Der Eingabeparameter m_{in} der Operation *encrypt* des Moduls beeinflusst den Zustand sl. Dieser Zustand beeinflusst den Ausgabeparameter m_{out} der Operation *decrypt*, falls ihr Eingabeparameter c_{in} einer der vorhergehenden Ausgaben c_{out} von *encrypt* entspricht. Jedoch beeinflusst m_{in} *nicht* den Rückgabewert c_{out} der Operation *encrypt*.

   $$m_{in} \rightarrow^* sl \wedge m_{in} \not\rightarrow^* c_{out} \wedge (c_{in} = c_{out} \implies sl \rightarrow^* m_{out})$$

IF-SIMULATION DES MODULS IM MODUL-STUB Der vereinfachte Pseudocode der IF-Simulation der Klassen `Decryptor` und `Encryptor`, der die im vorigen Abschnitten formulierten Annahmen erfüllt, ist in Listing 15 abgebildet.

`Decryptor` bildet dabei ein asymmetrisches Schlüsselpaar ab; mit Hilfe der `getEncryptor`-Methode kann die dazugehörige Instanz der `Encryptor`-Klasse erhalten werden, die den öffentlichen Schlüssel aus diesem Schlüsselpaar abbildet. Beide Instanzen teilen dabei zu jedem Zeitpunkt einen gemeinsamen Zustand `sl`, der als eine Map von Klartexten auf Geheimtexte repräsentiert wird.

Das Ergebnis der Verschlüsselung c_{out} wird als ein Zufallswert abgebildet, die *nicht* von der Eingabe abhängt. Die zu verschlüsselnde Information m_{in} wird in der Map `sl` zusammen mit diesem Zufallswert abgelegt. So wird simuliert, dass es keinen Informationsfluss des Klartextes zu einem öffentlicher Beobachter von c_{out} gibt, falls dieser nicht den geheimen Schlüssel besitzt, mit dem er den Geheimtext entschlüsselt.

Bei der Entschlüsselung von c_{in} mit der korrekten Instanz von `Decryptor` wird ein Informationsfluss von dem Zustand `sl` zur entschlüsselten Ausgabe m_{out} simuliert. Damit wird abgebildet, dass es einen Informationsfluss vom ursprünglichen Klartext zum Ergebnis der Entschlüsselung m_{out} gibt, falls die korrekte Instanz von `Decryptor` verwendet wurde.

Listing 15: Pseudocode der IF-Simulation von Crypto-Operationen

```
1 class Decryptor {
2         Map sl;
3
4         public var decrypt(var c_in){
5                 if(sl.containsKey(c_in))
```

```
6                       return m_out = sl.get(c_in);
7               else
8                       return m_out = random();
9       }
10
11      public Encryptor getEncryptor(){
12              return new Encryptor(sl);
13      }
14  }
15
16  class Encryptor {
17      Map sl;
18
19      public Encryptor(Map sl){
20              this.sl = sl;
21      }
22
23      public var encrypt(var m_in){
24              var rand = random();
25              sl.put(m_in, rand);
26              return c_out = rand;
27      }
28  }
```

Kann man zeigen, dass die getroffenen Annahmen über die konkrete Implementierung des Moduls korrekt sind, so gilt Folgendes:

- Die IF-Simulation dieses Moduls erfasst alle relevanten Informationsflüsse der echten Implementierung, und insbesondere den Fluss des zu verschlüsselnden Klartextes zum Ergebnis der Entschlüsselungsoperation. Dabei gibt es keinen Informationsfluss dieses Klartext zum Ergebnis der Verschlüsselung: so kann ein Angreifer, der den Geheimtext erfährt, daraus nicht auf den Klartext schließen, ohne im Besitz des korrekten geheimen Schlüssels zu sein
- Die Simulation ist geeignet, da hierfür speziell für die IF-Analyse entwickelte Methoden der IFlow-Bibliothek genutzt werden (vgl. Unterunterabschnitt 17.2.1.1)
- Die Simulation ist konservativ, da von Angreifern abstrahiert wird, die in der Lage sind, die eingesetzten Verschlüsselungsverfahren zu brechen

GARANTIEN DER KONKRETEN IMPLEMENTIERUNG Die vorgestellte Abstraktion der Kryptographieoperationen nutzt ihre ideelle Funktionalität, wie sie von Küsters et al. [55, 56] vorgeschlagen wurde (siehe auch [33]). Solche ideelle Funktionalität gibt u.a. Privatheitsgarantien selbst in Gegenwart eines polynomisch unbegrenzten Angreifers, der somit aus der Ausgabe der Verschlüsselung nie auf den

„Polynomisch begrenzt“ bezieht sich auf die Ressourcen eines Angreifers zum Brechen der Verschlüsselung

Klartext schließen kann. Besitzt eine Anwendung, die solche Funktionalität nutzt, die Nichtinterferenz-Eigenschaft, und ersetzt man die ideelle Funktionalität durch ihre reale Implementierung (indem man etwa die Operationen aus Listing 15 durch die RSA-Ver- und Entschlüsselungsoperationen ersetzt), so hat die Anwendung die Eigenschaft der kryptographischen Ununterscheidbarkeit für einen polynomisch begrenzten Angreifer (d.h., ein Angreifer kann anhand eines Geheimtextes nicht entscheiden, welche von zwei Klartexten verschlüsselt worden sind, außer es stehen ihm unbegrenzt viele Ressourcen zur Verfügung). Das bedeutet u.a., dass die verschlüsselten Daten für einen solchen Angreifer geheim bleiben.

Die konkrete Implementierung der IFlow-Bibliothek nutzt hierzu RSA-OAEP mit 2048-Bit-Schlüsseln für asymmetrische Verschlüsselung, AES-CBC mit PKCS5-Padding und 256-Bit-Schlüsseln für symmetrische Verschlüsselung, und SHA256 mit RSA und 2048-Bit-Schlüsseln für Signaturen. Hierbei wird das „Bouncy Castle"-Framework für Java genutzt.

Bouncy Castle
`https://www.bouncycastle.org/java.html`

17.2.2.4 *Vordefinierte grafische Benutzeroberfläche*

KURZBESCHREIBUNG IFlow bietet eine vordefinierte graphische Oberfläche an, die im Modell als die «`User`»-Komponente repräsentiert ist. Damit kann der Modellierer eine Reihe von vordefinierten Benutzerinteraktionsmöglichkeiten wie Bestätigung eines Dialogs oder Listenauswahl nutzen, ohne diese händisch implementieren zu müssen.

Auf Modellebene ist das Modul die vordefinierte Klasse mit dem «`User`»-Stereotyp. Im Anwendungscode ist die «`User`»-Komponente als eine Java-Klasse repräsentiert, das die notwendige Schnittstellen für die modellierten Benutzerinteraktion bietet. Diese Klasse ist anwendungsspezifisch, und wird automatisch aus dem Modell generiert.

INTERAKTION DES MODULS MIT DEM ANWENDUNGSKERN Im Modell kann eine IFlow-App im Sequenzdiagramm eine vordefinierte Nachricht an eine «`User`»-Komponente senden. Da eine vordefinierte Nachricht mehrmals versandt werden kann, wird sie im Code bei jedem Vorkommen auf eine eindeutig benannte Nachricht $\texttt{in}_x$ abgebildet. Die «`User`»-Komponente antwortet mit einer eindeutigen Rücknachricht $\texttt{out}_x$. Im Code ist die Interaktion mit der manuellen Benutzeroberfläche durch Aufrufe der Nachrichtenbehandlungsmethoden gelöst (siehe Unterunterabschnitt 17.2.1.4). Die Eingabeparameter des Moduls sind also die Methodenparameter $\texttt{in}_1..\texttt{in}_n$ der Nachrichtenbehandlungsmethoden $\texttt{in}_1\texttt{(var in}_1\texttt{)}..\texttt{in}_n\texttt{(var in}_n\texttt{)}$ der «`User`»-Komponente sowie die Benutzereingabe `u`, während die Ausgabeparameter des Moduls durch die Methodenparameter $\texttt{out}_1..\texttt{out}_n$ der gleichnamigen Nachrichtenbehand-

lungsmethoden `out1(var out1)..outn(var outn)` der aufrufenden IFlow-App abgebildet sind. Dabei ist n die Anzahl der von der *«User»*-Komponente erhaltenen Nachrichten.

Für manche Benutzerinteraktionen wird eine IFlow-App aufgerufen, die für die sichere Benutzereingabe zuständig ist (siehe Unterunterabschnitt 16.2.4.5); für andere wird die Benutzeroberfläche der aufrufenden Anwendung dynamisch angepasst.

ANNAHMEN ÜBER DIE KONKRETE IMPLEMENTIERUNG Das Modul besteht aus (1) dem anwendungsspezifischen, automatisch generiertem Code der *«User»*-Komponente, sowie (2) der darunterliegenden IFlow-Bibliothek, die die plattformspezifische, funktionale Implementierung der Benutzerinteraktion zur Verfügung steht. Für (1) wird im Codeskelett und der finalen Anwendung dieselbe Implementierung genutzt, wodurch dafür keine IF-Simulation benötigt wird. Für (2) wird im Codeskelett eine Stub-Implementierung genutzt, während bei der finalen Anwendung eine konkrete Implementierung eingesetzt wird.

Für die konkrete Implementierung des Moduls wird Folgendes angenommen:

1. Das Modul interagiert mit dem Anwendungskern nur über dessen Ein- und Ausgabeparameter:

   $M \cap C = \{in_1..in_n\} \cup \{out_1..out_n\}$

2. Das Modul greift auf keine plattformspezifische Quellen oder Senken zu:

   Bildschirmausgabe ist selbstverständlich erlaubt

   $M \cap Q = \emptyset$

   $M \cap S = \emptyset$

3. Die Antwortnachricht ist beeinflusst von den in der Nachricht an die *«User»*-Komponente erhaltenen Informationen sowie den Benutzereingaben:

   $\forall_{x \in \{1..n\}} in_x \rightarrow^* out_x \wedge u \rightarrow^* out_x$

4. Der modellierte Kontrollfluss wird eingehalten, d.h., die Benutzerinteraktion ist entweder *erfolgreich*, und der Programmablauf wird mit dem Versenden der Antwortnachricht an die aufrufende App-Komponente fortgeführt, oder sie wird *abgebrochen*, wodurch man zum Anfang des Programmablaufs zurückkehrt (siehe Unterunterabschnitt 17.2.1.5)

5. Das Modul ist zustandslos

IF-SIMULATION DES MODULS IM MODUL-STUB Im Benutzeroberflächenmodul ist jede im Modell vorkommende Benutzeraktion als

eigene und eindeutige Methode der «*User*»-Komponente implementiert, die die vordefinierte Nachricht mit anwendungsspezifischen Daten entgegennimmt (vgl. Listing 16). Der Code der «*User*»-Komponente wird sowohl im Codeskelett als auch der finalen Anwendung unverändert übernommen. Diese Methode hat die folgenden Aufgaben: (1) Instantiierung einer Callback-Methode out_x, die nach erfolgreicher Benutzerinteraktion aufgerufen werden soll und die Antwortnachricht mit den Benutzerdaten an die aufrufende Komponente zurückschickt (Zeile 3), und (2) Anzeigen der Benutzeroberfläche mit der vordefinierten `showUI`-Methode, parametrisiert mit der Callback-Methode sowie den anwendungsspezifischen Daten (Zeile 4).

`showUI` ist stellvertretend für die Methode der IFlow-Bibliothek, die die gewünschte Benutzeroberfläche anzeigt

Listing 16: Pseudocode der «*User*»-Komponente

```
1  class User extends IFlowUser {
2          public in_1(var in){
3                  var callback = (var out) -> out_1(out);
4                  showUI(in_1, callback);
5          }
6          ⋮
7          public in_n(var in){
8                  var callback = (out) -> out_n(out);
9                  showUI(in_n, callback);
10         }
11 }
```

Listing 17 zeigt den Pseudocode der IF-Simulation der `showUI`-Methode. Darin wird die Callback-Methode mit einem Datum parametrisiert, das aus den Eingabeparametern des Moduls in_x sowie der Benutzereingabe u abgeleitet wird (Annahme 3). Der Aufruf der Callback-Methode ist bedingt und hängt ebenfalls von der simulierten Benutzereingabe ab (Annahme 4). Wird die Methode nicht ausgeführt, so kehrt der Programmfluss durch die Lifecycle-Simulation (siehe Unterunterabschnitt 17.2.1.5) wieder zum ersten Schritt des modellierten Programmablaufs ab. Annahmen 1, 2, und 4 sind trivial erfüllt.

Listing 17: Pseudocode der IF-Simulation der `showUI`-Methode

```
1  class IFlowUser {
2          public showUI(var in, var callback){
3                  var out = (in, u);
4                  if(u) callback(out);
5          }
6  }
```

Kann man zeigen, dass die getroffenen Annahmen über die konkrete Implementierung des Moduls korrekt sind, so gilt Folgendes:

- Die IF-Simulation dieses Moduls ist eine Überapproximation, da alle relevanten Informationsflüsse, die in der konkreten Im-

plementierung verkommenen können, erfasst werden: der Inhalt der Antwortnachricht hängt von allen Daten in der Nachricht an die «*User*»-Komponente ab, und der simulierte Kontrollfluss deckt bei jeder Benutzerinteraktion jeden möglichen Programmablauf ab

- Die Simulation ist geeignet, da hierfür speziell für die IF-Analyse entwickelte Methoden der IFlow-Bibliothek genutzt werden. Zudem wird in der Simulation die konkrete Callback-Methode explizit aufgerufen, ohne dass die unterliegende Android-Plattform analysiert werden muss, die bei der finalen Anwendung für den Aufruf der Callback-Methode zuständig ist.
- Die Simulation ist konservativ, da von potentiell möglichen Informationsflüssen abstrahiert wird, zu welchen der Benutzer selbst beiträgt. So wäre es denkbar, dass die vom Benutzer gelieferte Eingabe nicht zufällig ist (beispielsweise dadurch, dass er sich die vorherigen Bildschirmausgaben gemerkt hat, und das Modul somit als zustandsbehaftet angesehen werden müsste), was jedoch die IF-Analyse durch zu viele gemeldete IF-Verletzungen weniger aussagekräftig machen würde.

GARANTIEN DER KONKRETEN IMPLEMENTIERUNG Da der Code der «*User*»-Komponente sowohl im Codeskelett als auch der finalen Anwendung der gleiche ist, muss lediglich die konkrete Implementierung der `showUI(var in, var callback)`-Methoden des IFlow-Frameworks auf die Annahmen hin untersucht werden.

Android-Fragments sind Teile der Benutzeroberfläche einer Activity, die ein eigenes Layout haben

Zur Anzeige einer Benutzeroberfläche kann sowohl ein Android-Fragment verwendet werden, als auch eine externe IFlow-App zur sicheren Benutzereingabe (siehe Unterunterabschnitt 16.2.4.5). In beiden Fällen wird der Inhalt der `in`-Variable lediglich zum Anzeigen auf der Benutzeroberfläche verwendet.

Im Folgenden wird die konkrete Implementierung anhand der Anzeige zur Auswahl eines Listenelements diskutiert (vgl. Listing 22 und Listing 23).

Die Beschriftungen der GUI-Navigationselemente sind statisch und nicht geheim

Zum Anzeigen eines Fragments lädt die konkrete Implementierung die Beschriftungen der GUI-Navigationselemente aus einer XML-Datei (vgl. Listing 23, Zeile 36) sowie die übergebenen anwendungsspezifischen Daten aus der Nachricht an die «*User*»-Komponente (vgl. Listing 23, Zeile 82).

Die Behandlung der Benutzerinteraktionsereignisse implementiert den Aufruf der anwendungsspezifischen thode parametrisiert mit den eingegebenen Benutzerdaten (im Falle einer Auswahlaufforderung wie bei `GetSingleSelection` leiten sich diese Benutzerdaten aus den erhaltenen, anwendungsspezifischen Daten ab, vgl. Listing 23, Zeile 93). Der Benutzerabbruch (beispielsweise durch das Drücken des *Back*-Hardwareknopfs auf dem Gerät) wird als das Versenden

eines Fehler-Codes an den letzten Aufrufer implementiert, der diesen bis an den Erstaufrufer propagiert, wonach der Programmablauf wieder von Anfang an beginnt (vgl. Listing 24). Anschließend wird das Fragment in der Activity der Aufruferkomponente angezeigt (vgl. Listing 22, Zeile 28).

Die offizielle Dokumentation der verwendeten API impliziert keine Leaks der übergebenen Daten an weitere Informationssenken oder Zugriffe auf weitere Quellen

Zum Aufruf der SecureInput-App zur sicheren Eingabe wird ein expliziter Intent an diese App mit den anwendungsspezifischen Daten aus der Nachricht an die «*User*»-Komponente konstruiert, die Callback-Methode wird als Behandlungsmethode des Antwort-Intents registriert, und der Intent versandt. Strenggenommen wird dadurch die Annahme 2 verletzt, da dadurch aber keine Information an eine andere Anwendungskomponente gelangen kann, kann dies vernachlässigt werden.

Folglich werden dadurch die Annahmen an die konkrete Implementierung erfüllt:

1. Die Daten aus der Nachricht an die «*User*»-Komponente werden lediglich zur Anzeige auf dem Bildschirm und Zusammenstellen der Antwortnachricht durch die registrierte Callback-Methode verwendet (Annahme 3)
2. Die Eingaben werden vom Benutzer auf dem Bildschirm getätigt und werden ausschließlich (evtl. über einen Antwort-Intent) an die registrierte Callback-Methode geschickt (Annahme 3)
3. Bei erfolgreicher Benutzereingabe wird der modellierte Programmfluss durch die Callback-Methode fortgeführt, bzw. vollständig abgebrochen, wonach der Programmablauf erneut beginnen kann (Annahme 4)
4. Es wird nicht auf den Anwendungskern oder plattformspezifischen Quellen oder Senken (außer zum Versenden eines expliziten Intents an die App zur sicheren Benutzereingabe) zugegriffen (Annahmen 1 und 2)
5. Das Modul hat keinen persistenten Zustand (Annahme 5)

17.2.2.5 *Manuelle grafische Benutzeroberfläche*

KURZBESCHREIBUNG IFlow bietet dem Modellierer die Möglichkeit, eine eigene graphische Oberfläche für seine IFlow-Anwendung zu implementieren. Dadurch können flexiblere Benutzerinteraktionen ermöglicht werden als mit der vordefinierten Benutzeroberfläche.

Auf Modellebene ist das Modul eine Klasse mit dem «*GUI*»-Stereotyp. Im Anwendungscode wird diese durch eine Klasse repräsentiert, die das das Java-Interface `IGUI` implementiert. Im Codeskelett wird das Interface von einer automatisch generierten Klasse implementiert, das die Informationsflüsse der echten Benutzeroberfläche simuliert. In der finalen Anwendung wird die Klasse durch eine manuelle Implementierung der echten Benutzeroberfläche ersetzt.

INTERAKTION DES MODULS MIT DEM ANWENDUNGSKERN Im Modell kann die *«GUI»*-Komponente im Sequenzdiagramm eine Nachricht out_x an eine IFlow-App senden, und erhält von ihr eine Rücknachricht in_x. Im Code ist die Interaktion mit der manuellen Benutzeroberfläche durch Aufrufe der Nachrichtenbehandlungsmethoden gelöst (siehe Unterunterabschnitt 17.2.1.4).

Im Gegensatz zur vordefinierten Benutzeroberfläche ist die manuelle Oberfläche der Aufrufer der IFlow-App

Die Eingabeparameter des Moduls sind also die Methodenparameter `in1..inn` der gleichnamigen Nachrichtenbehandlungsmethoden `in1(var in1)..inn(var inn)` der *«GUI»*-Komponente sowie die Benutzereingabe `u`, während die Ausgabeparameter des Moduls durch die Methodenparameter `out1..outn` der gleichnamigen Nachrichtenbehandlungsmethoden `out1(var out1)..outn(var outn)` der aufgerufenen IFlow-Apps abgebildet sind. Dabei ist n die Anzahl der von der *«GUI»*-Komponente ausgehenden Nachrichten.

ANNAHMEN ÜBER DIE KONKRETE IMPLEMENTIERUNG Für die konkrete Implementierung des Moduls wird Folgendes angenommen:

1. Das Modul interagiert mit dem Anwendungskern nur über dessen Ein- und Ausgabeparameter:

   $M \cap C = \{in_1..in_n\} \cup \{out_1..out_n\}$

2. Das Modul greift auf keine plattformspezifische Quellen oder Senken zu:

   Bildschirmausgabe ist selbstverständlich erlaubt

   $M \cap Q = \emptyset$

   $M \cap S = \emptyset$

3. Das Modul ist zustandsbehaftet und hat den persistenten Zustand $sl \in M$

4. Alle Eingabeparameter beeinflussen den persistenten Zustand des Moduls

   $\forall_{x \in \{1..n\}} in_x \rightarrow^* sl$

5. Der persistente Zustand des Moduls sowie die Benutzereingabe u beeinflusst alle Ausgabeparameter

   $\forall_{x \in \{1..n\}} sl \rightarrow^* out_x \wedge u \rightarrow^* out_x$

6. Die Auswahl, welche Nachricht als nächste an die IFlow-App versandt werden soll (und somit, welche als Sequenzdiagramm modellierte Funktionalität der Anwendung angestoßen werden soll), wird vom Nutzer getroffen und somit zufällig

IF-SIMULATION DES MODULS IM MODUL-STUB Der Pseudocode der verwendeten IF-Simulation ist in Listing 18 abgebildet. Wie jede

IFlow-Komponenten hat auch das *«GUI»*-Modul eine Nachrichtenbehandlungsmethode für jede modellierte eingehende Nachricht (vgl. Listing 18, Zeilen 11-13). In dem Stub des Moduls wird mit den dafür vorgesehenen Methoden aus der IFlow-Bibliothek (siehe Unterunterabschnitt 17.2.1.1) ein Informationsfluss von den eingehenden Nachrichten zum Klassenattribut `sl` der generierten `GUI`-Klasse simuliert (Annahme 4, vgl. Listing 18, Zeilen 11-13). Im Codeskelett wird dieselbe Instanz der `GUI`-Klasse verwendet, wodurch `sl` den persistenten Zustand des Moduls abbildet (Annahme 3). Zudem implementiert diese Klasse auch eine `init()`-Methode, die im Codeskelett als erste aufgerufen wird (siehe Unterunterabschnitt 17.2.1.5). Diese Methode ist dafür zuständig, in einer (potentiell) endlosen Schleife und in zufälliger Reihenfolge die Nachrichtenbehandlungsmethoden der IFlow-App aufzurufen, um den modellierten Nachrichtenversand zu simulieren (Annahme 6). Diese Nachrichten werden mit dem Klassenattribut `sl` sowie dem Benutzerinput `u` parametrisiert, um den Informationsfluss von allen eingehenden zu der ausgehenden Nachricht zu simulieren (Annahme 5, vgl. Listing 18, Zeilen 6-8). Darüber hinaus greift die Simulation weder auf plattformspezifischen Quellen bzw. Senken noch auf den internen Zustand des Anwendungskerns zu (Annahmen 1 und 2) und bildet somit alle Annahmen an die konkrete Implementierung ab.

Listing 18: Pseudocode der IF-Simulation des *«GUI»*-Moduls

```
1  class GUI implements IGUI {
2          var sl;
3          public init(){
4                  while(u){
5                          out = (sl,u);
6                          if(u == 1) out_1(out);
7                          ⋮
8                          if(u == n) out_n(out);
9                  }
10         }
11         public in_1(var in){ sl = (sl,in); }
12         ⋮
13         public in_n(var in){ sl = (sl,in); }
14 }
```

Kann man zeigen, dass die getroffenen Annahmen über die konkrete Implementierung des Moduls korrekt sind, so gilt Folgendes:

- Die IF-Simulation dieses Moduls ist eine Überapproximation, da alle relevanten Informationsflüsse, die in der konkreten Implementierung verkommenen können, erfasst werden: das Modul speichert alle erhaltenen Informationen ab, der Inhalt der Nachricht an eine IFlow-App hängt von allen als Antwortnachrichten erhaltenen Informationen ab, und der simulierte Kon-

trollfluss deckt jede mögliche Benutzerentscheidung bezüglich der als nächstes auszuführenden Interaktion mit dem Anwendungskern ab

- Die Simulation ist geeignet, da hierfür speziell für die IF-Analyse entwickelte Methoden der IFlow-Bibliothek genutzt werden (vgl. Unterunterabschnitt 17.2.1.1)
- Die Simulation ist konservativ, da von potentiell möglichen Informationsflüssen abstrahiert wird, zu welchen der Benutzer selbst beiträgt. So wäre es denkbar, dass die vom Benutzer gelieferte Eingabe nicht zufällig ist, was jedoch die IF-Analyse durch zu viele gemeldete IF-Verletzungen weniger aussagekräftig machen würde.

ÜBERPRÜFUNG DER ANNAHMEN Mit Hilfe einer Call-Graph-Analyse wird sichergestellt, dass die konkrete Implementierung weder auf plattformspezifische Quellen bzw. Senken noch auf den internen Zustand des Anwendungskerns zugreift (Annahmen 1 und 2, vgl. Unterabschnitt 16.2.3). Somit sind die einzig möglichen relevanten Informationsflüsse innerhalb der konkreten Implementierung des Moduls diejenigen, die bereits bei der IF-Simulation berücksichtigt worden sind (Annahmen 3-6, vgl. Unterunterabschnitt 17.2.2.5) Java-Exceptions, die innerhalb der manuellen Implementierung des Moduls geworfen werden führen entweder zum vorzeitigen Beenden und Neustart der Anwendung, wonach wiederum alle modellierten Benutzerinteraktionen möglich sind, oder werden vom Entwickler abgefangen und behandelt. Beide Fälle sind von der IF-Simulation implizit abgedeckt, da dadurch keine weiteren relevanten Informationsflüsse entstehen können.

17.2.2.6 *Externe Komponenten*

KURZBESCHREIBUNG IFlow bietet dem Modellierer die Möglichkeit, externe Apps und Services in seine Anwendung einzubinden.

Auf Modellebene ist das Modul eine Klasse mit dem *«ExternalApplication»*- oder *«ExternalService»*-Stereotyp. Ein solches Modul wird als zustandsbehaftet angenommen. Im Anwendungscode wird diese Klasse durch eine gleichnamige, automatisch generierte Java-Klasse abgebildet, die sowohl im Codeskelett als auch der finalen Anwendung genutzt wird. Die Klasse erbt von einer der im IFlow-Framework definierten Klassen:

- `ExternalApplicationGSL`, repräsentiert eine zustandsbehaftete Android-App
- `ExternalRestServiceGSL`, repräsentiert einen zustandsbehafteten REST-Webservice

- `ExternalSoapServiceGSL`, repräsentiert einen zustandsbehafteten SOAP-Webservice

Zudem nutzt die generierte Klasse manuelle Methoden zur Konvertierung zwischen den Nachrichtenkodierungen, die von der externen bzw. IFlow-Anwendung unterstützt wird.

Im Folgenden wird das Modul anhand einer externen Android-App diskutiert, die eine Antwortnachricht liefert. Die anderen unterstützten Komponententypen funktionieren analog, und verwenden lediglich ein anderes Kommunikationsprotokoll (REST bzw. SOAP über HTTPS statt Android-Intents).

INTERAKTION DES MODULS MIT DEM ANWENDUNGSKERN Im Modell kann eine IFlow-App und -Service im Sequenzdiagramm eine Nachricht in_x an eine externe IFlow-Komponente senden, und erhält von ihr eine Rücknachricht out_x. Im Code ist die Interaktion mit der externen Komponente durch Aufrufe der Nachrichtenbehandlungsmethoden einer automatisch generierten Schnittstellenklasse gelöst (siehe Unterunterabschnitt 17.2.1.4).

Die Eingabeparameter des Moduls sind also die Methodenparameter $in_1..in_n$ der gleichnamigen Nachrichtenbehandlungsmethoden `in1(var in1)..inn(var inn)` der externen Komponente, während die Ausgabeparameter des Moduls durch die Methodenparameter $out_1..out_n$ der gleichnamigen Nachrichtenbehandlungsmethoden `out1(var out1)..outn(var outn)` der aufrufenden IFlow-App under -Service abgebildet sind. Dabei ist n die Anzahl der im Modell an die externen Komponente gesandten Nachrichten.

ANNAHMEN ÜBER DIE KONKRETE IMPLEMENTIERUNG Für die konkrete Implementierung des Moduls wird Folgendes angenommen:

1. Das Modul interagiert mit dem Anwendungskern nur über dessen Ein- und Ausgabeparameter:

   $M \cap C = \{in_1..in_n\} \cup \{out_1..out_n\}$

2. Das Modul greift auf keine plattformspezifische Quellen oder Senken zu:

   $M \cap Q = \emptyset, M \cap S = \emptyset$

3. Das Modul hat den persistenten Zustand $sl \in M$

4. Alle Eingabeparameter beeinflussen den persistenten Zustand des Moduls

   $\forall_{x \in \{1..n\}} in_x \rightarrow^* sl$

5. Der persistente Zustand des Moduls beeinflusst alle Ausgabeparameter

$\forall_{x \in \{1..n\}} sl \rightarrow^* out_x$

6. Die manuellen Methoden, die für die Konvertierung der Nachrichtenkodierung zuständig sind, greifen auf keine plattformspezifischen Quellen oder Senken zu. Darüber hinaus gelten für sie dieselben Annahmen wie für alle anderen manuellen Methoden (siehe Unterunterabschnitt 17.2.2.1)

IF-SIMULATION DES MODULS IM MODUL-STUB Der Pseudocode der Java-Klasse, die das Modul repräsentiert, ist in Listing 19 abgebildet. Dieser ist für die Kommunikation mit der echten externen Anwendung zuständig, wird automatisch generiert und wird sowohl im Codeskelett als auch der finalen Anwendung unverändert verwendet. Wie jede IFlow-Komponente hat auch diese Klasse eine Nachrichtenbehandlungsmethode für jede modellierte eingehende Nachricht (vgl. Listing 19, Zeilen 2-12).

Diese Methoden haben die folgenden Aufgaben:

1. Konvertierung der IFlow-Nachrichtenkodierung zur nativen Nachrichtenkodierung der externen Anwendung mit Hilfe einer händisch implementierten Methode $mm_{exApp}In_x$ (Zeile 3).

2. Instantiierung einer Callback-Methode, die beim Erhalten einer Antwortnachricht von der externen Komponente aufgerufen werden soll (Zeile 4). Diese konvertiert die Antwortnachricht zur IFlow-Nachrichtenkodierung mit Hilfe einer händisch implementierten Methode $mm_{exApp}Out_x$ und leitet die resultierende Nachricht an die aufrufende IFlow-Komponente weiter.

3. Aufruf der `executeRequest`-Methode zum Versenden der Nachricht an die externe Komponente, parametrisiert mit der konvertierter Nachricht sowie der Callback-Methode (Zeile 5).

`executeRequest` *ist stellvertretend für die Methode der IFlow-Bibliothek, die für die Kommunikation mit einer externen App oder Service zuständig ist*

Listing 19: Pseudocode der Nachrichtenbehandlungsmethode des Moduls

```
1  class ExApp implements ExternalApplicationGSL {
2          public in_1(var in){
3                  var inEx = mm_exApp In_1(in);
4                  var callback = (outEx) -> out_1(mm_exApp Out_1(out
                       ));
5                  executeRequest(inEx, callback);
6          }
7          ⋮
8          public in_n(var in){
9                  var inEx = mm_exApp In_n(in);
10                 var callback = (outEx) -> out_n(mm_exApp Out_n(out
                       ));
```

```
11                  executeRequest(inEx, callback);
12          }
13  }
```

Listing 20 zeigt den Pseudocode der IF-Simulation der `executeRequest`-Methode, die gleichzeitig die Informationsflüsse in der echten externen Komponente abbildet. Darin wird mit den dafür vorgesehenen Methoden aus der IFlow-Bibliothek (siehe Unterunterabschnitt 17.2.1.1) ein Informationsfluss von der eingehenden Nachricht zum Klassenattribut `sl` der `ExternalApplicationGSL`-Klasse simuliert (Annahmen 3 und 4, vgl. Listing 20, Zeilen 2 und 5). Danach wird die übergebene Callback-Methode mit dem `sl`-Attribut parametrisiert und aufgerufen (Annahme 5, vgl. Listing 20, Zeile 6). Die Beschreibung der IF-Simulation der verwendeten manuellen Methoden ist in Unterunterabschnitt 17.2.2.1 zu finden.

Die restlichen Annahmen 1, 2, und 7 sind von der IF-Simulation trivial erfüllt.

Listing 20: Pseudocode der IF-Simulation von `executeRequest`

```
1  class ExternalApplicationGSL {
2          var sl;
3
4          public executeRequest(var inEx, var callback){
5                  sl = (sl, inEx);
6                  callback(sl);
7          }
8  }
```

Kann man zeigen, dass die getroffenen Annahmen über die konkrete Implementierung des Moduls korrekt sind, so gilt Folgendes:

- Die IF-Simulation dieses Moduls ist eine Überapproximation, wobei das Grad der Überapproximation vom Modellierer gewählt wurde: entweder speichert das Modul alle erhaltenen Informationen ab und nutzt diese, um eine Antwort zu versenden, oder nutzt dafür lediglich die zuletzt erhaltene Nachricht von der IFlow-Komponente
- Die Simulation ist geeignet, da hierfür speziell für die IF-Analyse entwickelte Framework-Methoden genutzt werden (vgl. Unterunterabschnitt 17.2.1.1). Zudem wird in der Simulation die konkrete Callback-Methode explizit aufgerufen, ohne dass die unterliegende Plattform analysiert werden muss, die bei der finalen Anwendung für den Aufruf der Callback-Methode zuständig ist.
- Die Simulation ist konservativ, da von potentiell möglichen Informationsflüssen abstrahiert wird, die die externe Komponente zusätzlich haben könnte, wie beispielsweise die Weiterleitung der erhaltenen Informationen an Werbenetzwerke

GARANTIEN DER KONKRETEN IMPLEMENTIERUNG Da der Code der Java-Klasse, die die externe Anwendung repräsentiert sowohl im Codeskelett als auch der finalen Anwendung der gleiche ist, muss lediglich die konkrete Implementierung der `executeRequest(...)`-Methoden des IFlow-Frameworks auf die Annahmen hin untersucht werden.

In Listing 25 ist die Implementierung dieser Methode zur Kommunikation mit einer externen Android-App über Intents abgebildet. Es existieren weitere Versionen dieser Methode, u.a. zur Kommunikation mit Webservices, die jedoch alle analog implementiert sind.

In Zeile 22 wird überprüft, ob auf dem Gerät Anwendungen installiert sind, die die Nachricht empfangen können (siehe Listing 25). Ist dies der Fall, so wird die Callback-Methode bei dem Aufrufer registriert (Zeile 23), und die externe Anwendung aufgerufen (Zeile 25). Anderenfalls wird eine Fehlermeldung ausgegeben, und der modellierte Programmfluss wird nicht fortgesetzt.

Die offizielle Dokumentation der verwendeten API impliziert keine Leaks der übergebenen Daten an weitere Informationssenken oder Zugriffe auf weitere Quellen

Die Annahmen an die konkrete Implementierung der Schnittstelle zur Kommunikation mit der externen Komponente werden wie gefolgt erfüllt bzw. überprüft:

1. Es wird nicht auf den Anwendungskern oder plattformspezifischen Quellen oder Senken (außer zum Versenden eines expliziten Intents an die externe App) zugegriffen (Annahmen 1 und 2)

2. Die Daten aus der Nachricht an die externe Komponente werden lediglich zum Versenden an die externe Komponente verwendet (Annahmen 3-5)

3. Die zur Nachrichtenkonvertierung verwendeten manuellen Methoden werden wie in Unterunterabschnitt 17.2.2.1 beschrieben mit Hilfe einer Call-Graph-Analyse auf die Annahmen hin überprüft (Annahme 6)

Letztlich muss die eigentliche externe Komponente bezüglich der formulierten Annahmen betrachtet werden. Da der Quellcode der externen Komponenten oft nicht vorliegt, muss anhand ihrer Dokumentation (soweit vorhanden) entschieden werden, ob die Annahmen 3-6 erfüllt sind. Im Fall einer externen Android-App garantiert die Android-Plattform, dass die Annahme 2 erfüllt ist (siehe Unterabschnitt 17.3.2, während bei externen Webservices dies dann gilt, wenn die Deployment- und Nutzungsvorgaben eingehalten werden (vgl. Anhang D). Zudem lassen die Berechtigungen der externen Android-App auf die plattformspezifischen Quellen und Senken zurückschließen, auf die die externe App möglicherweise zugreift (Annahme 1, vgl. Unterunterabschnitt 17.3.2.1).

17.3 ANWENDUNGSKONTEXT

Die Komplexität heutiger Computersysteme verbietet es dem Entwickler eines sicherheitskritischen Systems, alle Sicherheitsaspekte selbstständig abzudecken und zu garantieren, ohne dabei die Benutzerfreundlichkeit und die Funktionalität zu vernachlässigen. Da bei der Sicherheitsanalyse immer das Gesamtsystem betrachtet werden muss, ist es notwendig, das sichere Zusammenspiel zwischen dem entwickelten System und dessen Ausführungskontext zu garantieren.

Auch die in IFlow eingesetzte Informationsflussanalyse und -verifikation trifft eine Reihe von Annahmen über den Kontext des Deployments und der Ausführung einer IFlow-Anwendung. Dazu zählt z.B., dass die auf demselben mobilen Gerät installierten Apps miteinander nur über die vom Entwickler vorgegebenen Schnittstellen interagieren können. Diese Annahmen können durch den sorgfältigen Einsatz von verfügbaren Sicherheitsmechanismen, strikte Regeln für das Deployment der Anwendung, sowie Einhaltung der besten Sicherheitspraxis sichergestellt werden. Auch wird den Endnutzern der Anwendung empfohlen, sich an eine Reihe von Vorgaben bezüglich Sicherheitseinstellungen und Nutzung zu halten. Nur dann können die Informationsflusseigenschaften mit dem IFlow-Ansatz garantiert werden.

Im Folgenden wird der Kontext einer IFlow-Anwendung konkretisiert. Abschnitt 17.3.1 beschreibt und rechtfertigt die angenommenen Fähigkeiten der Angreifer, und Abschnitt 17.3.2 detailliert die eingesetzten Sicherheitsmechanismen.

17.3.1 *Angreiferfähigkeiten*

Wenn man von unerwünschten Informationsfluss spricht, geht man üblicherweise von einem "öffentlichen" Beobachter der als geheim spezifizierter Information aus. Dieser wird im Kontext der vorliegenden Arbeit als Angreifer bezeichnet. Es kann mehr als einen explizit modellierten Beobachter von solcher Information geben, zusätzlich zu potentiellen Angreifern, die sich aus dem impliziten Anwendungskontext ergeben. Ein solcher Angreifer kann sowohl passiv bleiben und lediglich die Informationen lesen, die ihm zukommen. Jedoch kann der Angreifer auch aktiv agieren, um die Freigabe solcher Information zu bewirken. Zudem unterscheidet man zwischen dem Angreifer, der als Teil des Systems agiert, und solchem, der von außerhalb auf das System einwirkt.

Um sinnvolle Aussagen über die Sicherheit des Gesamtsystems machen zu können, von dem die mit IFlow entwickelte Anwendung ein Teil ist, sowie geeignete Sicherheitsmaßnahmen sinnvoll auszuwählen, sollen nun die verschiedenen Typen von Angreifern betrachtet werden.

17.3.1.1 *Angreifer als Teil des modellierten Systems*

Bei der Informationsflussanalyse in IFlow geht man von einem oder mehreren öffentlichen Beobachtern im modellierten System aus. Diese leiten sich aus den Senken in den modellierten Informationsflusseigenschaften ab. So kann eine Anwendungskomponente wie eine App oder ein Service einen solchen Beobachter darstellen, aber auch eine plattformspezifische Senke wie das Dateisystem oder das Handynetz. Dadurch stuft man den Benutzer dieser App, Betreiber des Services, oder auch den Empfänger einer SMS-Nachricht (bzw. den Netzbetreiber) als nicht vertrauenswürdig ein. Die Informationsflussanalyse in IFlow stellt sicher, dass ein solcher Beobachter in keinem der möglichen Programmabläufe geheime Informationen in Erfahrung bringen kann.

17.3.1.2 *Dolev-Yao-Angreifer*

Oft wird das Dolev-Yao-Modell verwendet, um Eigenschaften eines sicherheitskritischen Protokolls zu beweisen [22]. Dabei geht man von einem Angreifer aus, der die Nachrichten auf der Leitung zwischen den Kommunikationsteilnehmern abhören, manipulieren, und unterdrücken kann. Da es sich bei den IFlow-Anwendungen um verteilte Systeme handelt, muss auch ein solcher Angreifer in Betracht gezogen werden. Dieser wird nicht explizit in MODELFLOW berücksichtigt, sondern wird implizit bei der Kommunikation zwischen Apps, Services, sowie zwischen Apps und Services angenommen. Diverse Sicherheitsmechanismen auf der Codeebene garantieren, dass ein solcher Angreifer keine geheime Information in Erfahrung bringen kann, und daher bei der Informationsflussanalyse nicht berücksichtigt werden muss. Insbesondere wird in IFlow von einem polynomisch begrenzten Angreifer ausgegangen (siehe Unterunterabschnitt 17.2.2.3).

17.3.1.3 *Böswillige Apps und Services*

IFlow-Anwendungen werden auf Android-Geräten sowie Webservern installiert. Üblicherweise befinden sich auf diesen Geräten auch weitere Apps und Services, die nicht als vertrauenswürdig angesehen werden dürfen. Abgesehen vom unbeabsichtigten Weiterleiten geheimer Information an solche Anwendungen durch das mit IFlow entwickelte System, können solche Apps oder Services auch aktiv versuchen, Nutzerdaten zu stehlen. Dies könnte beispielsweise durch das Auslesen des gemeinsamen Speichers geschehen, aber auch durch Stehlen des Fokus auf der Benutzeroberfläche. Ebenso könnten sie versuchen, sich als eine IFlow-App auszugeben, um das Vertrauen des Nutzers zu missbrauchen, oder mittels Inter-App-Kommunikation echten IFlow-Apps geheime Daten zu entwenden. Auch von solchen Agenten wird in IFlow auf Modellierungsebene abstrahiert; stattdessen sorgen implementierte und vorgegebene Sicherheitsmechanismen auf

Hardware- und Codeebene dafür, dass solche Angriffe verhindert werden.

17.3.1.4 *Unautorisierte Gerätenutzer*

Mobile Geräte unterliegen verstärkt der Gefahr, dass unautorisierte Nutzer Zugriff auf sie bekommen. Verglichen mit Desktops oder Server-Hardware ist es deutlich wahrscheinlicher, dass solche Geräte an öffentlichen Plätzen unbeaufsichtigt gelassen werden, wo sie gestohlen werden können. Eine Reihe von in Android eingebauten Sicherheitsmechanismen kann vom Gerätenutzer eingesetzt werden, um das Risiko von unautorisiertem Zugriff auf geheime Daten zu minimieren.

17.3.1.5 *Autorisierte Gerätenutzer*

Erfahrung zeigt, dass die Endnutzer von sicherheitskritischer Anwendung sich oft versehentlich selbst kompromittieren. Es muss also davon ausgegangen werden, dass der Nutzer ausreichend Sicherheitsbewusstsein an den Tag legt (siehe Anhang D).

17.3.1.6 *Plattform-Entwickler/Gerätehersteller/Behörden*

Wie bei den meisten sicherheitskritischen Systemen muss auch bei IFlow-Anwendungen einer Reihe von Aktoren vertraut werden. So muss u.a. davon ausgegangen werden, dass die genutzten Sicherheitsmechanismen der Zielplattformen fehlerfrei implementiert sind und keine Schwachstellen aufweisen. Ebenso muss den Geräteherstellern vertraut werden, dass sie die Hardware-Sicherheitsmechanismen korrekt implementieren, und keine Hintertüren einbauen. In der Vergangenheit erwiesen sich solche Annahmen oft als irrtümlich; so konnten sicherheitskritische Systeme wie TrustZone von ARM (u.a. für sichere Aufbewahrung von kryptographischen Schlüsseln auf mobilen Geräten) umgangen werden. Jedoch sind Annahmen bzgl. der Sicherheit solcher Geräte und Plattformen notwendig, wenn man diese nutzen möchte.

Zuletzt wird im Rahmen von der vorliegenden Arbeit davon ausgegangen, dass sogenannte “state actors” wie Geheimdienste, Strafverfolgunsbehörden o.Ä. außerhalb des Angreifermodells des Nutzers liegen.

17.3.2 *Plattformzusicherungen*

Die in IFlow genutzten Plattformen bieten eine Reihe von Sicherheitsmechanismen, die im Ansatz genutzt werden, um Informationsflusseigenschaften bezüglich der in Abschnitt 17.3.1 spezifizierten Angreiferfähigkeiten zu garantieren. Dabei handelt es sich um Mechanis-

men, die vom generierten Code gemäß den besten Sicherheitspraxen automatisch eingesetzt werden als auch solche, die der Endnutzer explizit aktivieren kann. Es wird in diesem Abschnitt hauptsächlich auf Zusicherungen der Android-Plattform für IFlow-Apps eingegangen. Auf Annahmen, die in IFlow über die Java-Webservice-Plattform getroffen werden, wird in Anhang D eingegangen.

17.3.2.1 *Android-Berechtigungssystem*

Die meisten aktuellen mobilen Betriebssysteme wie Android, iOS, sowie Windows Phone bieten dem App-Entwickler die Möglichkeit die Zugriffsrechte der App einzuschränken. In Android sind manche der Android-API-Zugriffe wie etwa Abfrage der aktuellen Position über den GPS-Sensor explizit durch solche Berechtigungen geschützt. Möchte der Entwickler sie nutzen, muss er diese Berechtigung bei der Installation der App vom Benutzer anfordern. Der Nutzer muss der Liste der angeforderten Berechtigungen zustimmen, um die Anwendung installieren zu können. Dadurch soll u.a. sichergestellt werden, dass der Benutzer darüber informiert wird, welche Funktionalität die Anwendung implementiert, sowie auf welche seiner privaten Daten die Anwendung voraussichtlich zugreifen wird. Es ist sinnvoll, die Liste der angeforderten Berechtigungen minimal zu halten.

In Android können solche Berechtigungen auch während der Laufzeit vom Benutzer abgefragt werden. Dadurch ist der Benutzer darüber im Klaren, *zu welchem Zeitpunkt* auf seine privaten Daten zugegriffen wird, und hat zudem die Möglichkeit, diesen Zugriff zu verwehren.

Jedoch hat dieses Berechtigungssystem einige gravierende Nachteile. Zum Einen sind die Berechtigungen sehr grobkörnig und nicht vollständig [26]; somit ist es für eine Anwendung beispielsweise möglich, Daten an einen Webserver zu versenden, ohne eine Berechtigung dafür anzufragen (beispielsweise durch das Öffnen eines Browsers mit einer bestimmten URL, die die geheimen Daten des Nutzers kodiert). Auch können durch ein solches Berechtigungssystem keine unerwünschten Informationsflüsse verhindert werden. Hat eine App sowohl Zugriff auf den GPS-Sensor als auch auf das Internet, so kann sie die aktuelle Information des Nutzers jederzeit an ein Werbenetzwerk versenden, selbst wenn die GPS-Funktionalität nur lokal auf dem Gerät gebraucht wird, um die gewünschte Funktionalität zu gewährleisten. Zudem können Apps mit scheinbar harmlosen Berechtigungen miteinander kooperieren, um geheime Daten zu entwenden [38, 83].

Obwohl die mit MODELFLOW modellierten Informationsflusseigenschaften deutlich aussagekräftiger als die Berechtigungen in Android sind, ist das Berechtigungssystem in Android nicht optional, und es ist sehr wohl im Sinne des Entwicklers, nur die notwendigen Berechtigungen anzufragen. Einerseits ist das ein für viele Endnut-

zer bereits vertrautes Sicherheitsmechanismus, bei dem sie in neueren Android-Versionen eine auch während der Laufzeit die Kontrolle über manche ihrer Daten behalten. Anwendungen, die zu viele Berechtigungen anfragen, werden von vielen Benutzern zu Recht kritisch beäugt.

In IFlow wird die Liste der benötigten Berechtigungen automatisch aus den *«uses»*-Annotationen der manuellen Methoden der IFlow-App generiert und in `AndroidManifest.xml` dieser App geschrieben. Dabei werden die Klassifikationen der kritischen API-Aufrufe genutzt [77], und diese auf geeignete Berechtigungen abgebildet. In IFlow werden Berechtigungen auch genutzt, um Inter-App-Kommunikation mit Intents abzusichern (siehe Unterunterabschnitt 17.3.2.4).

17.3.2.2 *Sandboxing*

Das Sicherheitsmodell von Android basiert auf dem unterliegenden Linux-System. In Linux erhält jeder Nutzer eine eigene Identität (*UID*), und kann nicht auf die Daten eines anderen Nutzers zugreifen. Ein Android-Gerät hat üblicherweise nur einen physikalischen Nutzer, daher nutzt Android diesen Sicherheitsmechanismus des Linux-Kernels um die installierten Apps voneinander zu isolieren. Dabei wird einer App bei der Installation eine eindeutige UID zugewiesen, und sie wird in einem eigenen Prozess mit dieser UID ausgeführt. Zudem wird jeder App ein eigenes Verzeichnis auf dem Dateisystem zugewiesen, auf das keine andere App zugreifen kann. Dadurch werden die Android-Apps auf Kernel-Ebene voneinander isoliert, und werden somit in einer eigenen "Sandbox" ausgeführt. Eine Ausnahme stellen Apps dar, die mit demselben öffentlichen Schlüssel signiert sind; solche Apps können sich eine UID teilen, vorausgesetzt, dies ist in ihrem Manifest explizit spezifiziert. Seit Android 4.3 implementiert das Sicherheits-Kernel-Modul *SELinux* das sog. *Mandatory Access Control* (dt.: zwingend erforderliche Zugangskontrolle) in Android [23]. Dies bedeutet u.a., dass Anwendungen nicht eigenständig über die Berechtigungen ihrer Ressourcen verfügen können, und somit beispielsweise ihre Daten aufgrund eines Programmierfehlers öffentlich lesbar machen können. Stattdessen werden Richtlinien über solche Berechtigungen zentral von SELinux verwaltet. Während in Android 4.3 Verletzungen solcher Richtlinien lediglich protokolliert wurden, werden solche Richtlinien in Android 5.0 erzwungen.

Solches System kann garantieren, dass Anwendungen nur über klar definierte Schnittstellen miteinander kommunizieren können, gibt aber wie auch das Berechtigungssystem in Android keine feingranularen Garantien bezüglich der Informationsflüsse zwischen den Anwendungen, dem Betriebssystem, und dem Netzwerk.

Da IFlow-Apps als herkömmliche Android-Apps auf das Gerät installiert werden, nutzen auch sie die Vorteile der Prozessisolation. Obwohl Apps aus einer IFlow-Anwendung mit demselben öffentlichen

Schlüssel signiert sind (siehe Unterunterabschnitt 17.3.2.4), haben sie dennoch eigene UIDs.

Dieses Sicherheitsmechanismus von Android rechtfertigt die Annahme, die bei der Informationsflussanalyse und -verifikation getroffen wird, dass die Apps isoliert voneinander ausgeführt werden. Eine IFlow-App kann also weder auf den Speicher einer anderen App zugreifen, noch hat eine andere (eventuell böswillige) App Zugriff auf ihren Speicher (siehe Abschnitt. Die Kommunikation zwischen den IFlow-Apps ist zudem nur über klar definierte Schnittstellen erlaubt, die in dem MODELFLOW-Modell explizit spezifiziert sind.

Der Sandboxing-Mechanismus von Android stellt sicher, dass die auf dem mobilen Gerät des Nutzer installierte Apps nicht auf die Daten der IFlow-Apps zugreifen können.

17.3.2.3 *APK-Signierung*

APK ist das Paketformat von Android für die Distribution und Installation von Apps, und enthält sowohl ihren Bytecode als auch die Ressourcen wie Konfigurationsdateien und Bilder. Solche APKs müssen vom Entwickler der Anwendung mit einem geheimen Schlüssel signiert werden. Wird die App aktualisiert (beispielsweise durch das manuelle Einspielen einer neuen APK, oder über den Update-Mechanismus eines App-Stores), so muss die neue APK ebenfalls mit demselben Schlüssel signiert sein (*same-origin-policy*). Dadurch wird verhindert, dass dem Nutzer gefälschte Versionen einer App untergeschoben werden können, sei es durch einen Social Engineering-Angriff (dies ist v.A. bei populären Apps ein beliebter Angriffsvektor [95]), oder durch einen kompromittierten App-Store. Solche Signaturen dienen auch zum Absichern von Inter-App-Kommunikation zwischen vertrauenswürdigen Apps vom selben Hersteller.

In IFlow werden alle Apps aus einer IFlow-Anwendung mit demselben geheimen Schlüssel signiert. Damit ist ein Angreifer nicht mehr in der Lage, gefälschte Versionen bereits installierter IFlow-Apps auf dem Gerät des Nutzers zu installieren, oder sich als eine weitere IFlow-App auszugeben, um mit echten IFlow-Apps kommunizieren zu können (siehe Unterunterabschnitt 17.3.2.4).

17.3.2.4 *Inter-App-Kommunikation*

Android stellt mehrere Möglichkeiten zur Verfügung, um Inter-App-Kommunikation zu implementieren. In den meisten Fällen empfiehlt sich die Kommunikation über sog. *Intents*. Intents sind Nachrichtenobjekte, die zwischen Bestandteilen einer Android-App wie *Activities*, *Services*, *Content Providers*, sowie *Broadcast receivers* ausgetauscht werden und Nutzdaten enthalten können. Mit Intents können solche Komponenten gestartet werden, wobei sie innerhalb einer App verschickt werden können, um beispielsweise eine interne Activity

*IFlow-Apps nutzen ausschließlich Android-*Activities*, die ihre Logik und Benutzeroberfläche implementieren*

anzuzeigen, oder auch zwischen verschiedenen Apps ausgetauscht werden können. Der App-Entwickler kann dazu im Manifest angeben, ob eine Komponente Intents von anderen Apps empfangen darf. Man unterscheidet zwischen expliziten und impliziten Intents; erstere rufen eine konkret spezifizierte Komponente auf, während letztere lediglich die Aktion wie etwa "Ort auf Karte anzeigen" angibt, und der Nutzer eine der dafür geeigneten Komponenten auswählen kann, um diese Aktion auszuführen. Dafür muss die Komponente lediglich einen *Intent-Filter* in ihrem Manifest definiert haben, der die Aktion festlegt, auf die die Komponente reagiert. Jede App ist in der Lage, einen Intent-Filter für solche implizite Intents zu registrieren, und somit gegebenenfalls Kommunikation, die für andere Apps gedacht war, abzuhören.

Um die Inter-App-Kommunikation über Intents auf nur vertrauenswürdige Apps einzuschränken, können manuelle Berechtigungen verwendet werden, die die Signatur des Aufrufers überprüfen (*signature-level protection*). Dadurch kann die App, die eine solche Berechtigung verwendet, nur von Apps Intents erhalten, deren APK mit demselben geheimen Schlüssel signiert wurde wie die Empfänger-App. Dies ist u.a. dann sinnvoll, wenn die Empfänger-App auf Nachfrage über ein Intent geheime Benutzerdaten an den Aufrufer liefern soll.

Die Kommunikation zwischen IFlow-Apps auf demselben Gerät muss mit expliziten Intents erfolgen, die mit solchen manuellen Berechtigungen geschützt sind. Dadurch sind andere (potentiell böswillige) Anwendungen auf dem Gerät des Nutzers nicht in der Lage, den IFlow-Apps geheime Daten über die Inter-App-Kommunikationsschnittstellen zu entwenden. Da zudem explizite Intents von Android nur an die spezifizierte Komponente weitergeleitet werden, sind dadurch Angreifer-Apps im Sinne des Dolev-Yao-Modells in IFlow ausgeschlossen.

17.3.2.5 *App-zu-Service- und Service-zu-Service-Kommunikation*

Die Kommunikation über das Internet muss als unsicher angenommen werden, und muss folglich gegen einen Dolev-Yao-Angreifer gesichert werden. Die gängige Lösung ist das *Transport Layer Security* (TLS)-Protokoll, ein Nachfolger des *Secure Sockets Layer* (SSL)-Protokolls. Es implementiert eine Punkt-zu-Punkt Transportverschlüsselung zwischen zwei Teilnehmern und wird häufig bei der Sicherung von Protokollen wie HTTP, SMTP, XMPP und weiteren eingesetzt [21]. Es kann mit einer Vielzahl von Kombinationen aus verschiedenen Kryptoalgorithmen zum Hashen, Vershlüsseln, Authentifizieren, sowie Schlüsselaustausch eingesetzt werden, wobei nur wenige dieser Kombinationen (auch *Cipher Suits* gennant) als empfohlen und sicher gelten. Im ersten Schritt werden Parameter wie unterstützte Protokollversionsnummer, Cipher Suits zwischen den Kommunikationsteilnehmern ausgetauscht. Nachdem die besten solcher Parameter

ausgewählt wurden, werden Zertifikate der Teilnehmer ausgetauscht und auf Gültigkeit überprüft. Schließlich wird ein symmetrischer Sitzungschlüssel ausgetauscht, mit dem die Kommunikation in der aktuellen Sitzung verschlüsselt wird.

Server-Zertifikate werden üblicherweise von Zertifizierungsstellen für die Server-Domäne ausgestellt. Um zu verhindern, dass gefälschte, jedoch gültige Zertifikate von böswillige oder kompromittierte Zertifizierungsstellen akzeptiert werden, wird manchmal das sog. *Certificate Pinning* bzw. *Public Key Pinning* eingesetzt. Hiermit wird das vom Entwickler als gültig angesehene Server-Zertifikat (oder öffentlicher Schlüssel) in der Client-Anwendung gespeichert und beim Aufbau einer Sitzung mit dem vom Server angegebenen Zertifikat verglichen. Stimmen diese nicht überein, wird die Sitzung abgebrochen.

In IFlow wird jede App-zu-Service- und Service-zu-Service-Kommunikation mittels TLS geschützt. Dabei findet auch die Zertifikatüberprüfung statt, wobei auch das Certificate Pinning genutzt werden kann. Letzteres ist optional, da es beim Ändern des Zertifikats dazu kommen kann, dass sich bestehende Clients nicht mehr zum Service verbinden können, ohne ein Update auszuführen. Damit ist die Kommunikation über das Internet gegen einen Dolev-Yao-Angreifer geschützt. Beim Deployment der IFlow-Webservices ist der Server-Betreiber angewiesen, aktuell empfohlene TLS-Konfiguration vorzunehmen (Cipher Suits mit kleinen Schlüssellängen, unsicheren Hashalgorithmen wie MD5 oder SHA-1 oder Verschlüsselungsalgorithmen mit Sicherheitsmängeln wie RC4 sind dabei möglichst zu meiden).

17.3.2.6 *Overlay-Schutz*

Eine vertrauenswürdige Benutzeroberfläche ist ein wichtiges Bestandteil einer Plattform für sicherheitskritische Anwendungen. Eine große Rolle spielt dabei die Fähigkeit des Nutzers, Bedienelemente einer vertrauenswürdigen Anwendung von weniger vertrauenswürdigen unterscheiden zu können. Ist eine Anwendung in der Lage, Teile einer anderen zu überlagern (*overlay*), ohne dass es dem Benutzer auffällt, so werden Angriffe wie *Click-Jacking* (Umleitung einer Benutzerinteraktion an weniger vertrauenswürdige Anwendungen) möglich. Ebenso können mit überlagerten Eingabefeldern geheime Informationen gestohlen werden, oder der Sinn der dargestellten Informationen manipuliert werden.

Android implementiert erst seit Version 6 einen Schutz gegen solche Angriffe. Apps, die andere Anwendungen überlagern möchten, müssen eine neue Berechtigung anfordern, die der Nutzer explizit erlauben muss. Dadurch werden die aufgezählten Angriffe zwar nicht unmöglich, werden aber deutlich erschwert.

In IFlow wurde der sicheren Eingabeaufforderung besondere Achtung geschenkt und Sicherheitsmaßnahmen gegen *Phishing*-Angriffe in Form von zusätzlichen Beschriftungen und Querverweisen auf die garantierten Informationsflusseigenschaften, benutzerdefinierte visuelle Merkmale, sowie explizite Verknüpfung zwischen der IFlow-App und der IFlow-Eingabeaufforderung implementiert. Es ist in Kontext von IFlow empfohlen, Android 6.0 oder höher zu verwenden, damit der Nutzer von dem zusätzlichen Overlay-Schutz gegen bösartige Anwendungen profitieren kann.

17.3.2.7 *Nutzer-Authentifizierung*

Mobile Geräte wie etwa Android-Smartphones bringen eine Reihe von Herausforderungen beim Implementieren der Benutzer-Authentifizierung. Herkömmliche Lösungen wie lange Passwörter sind in diesem Kontext wenig benutzerfreundlich, da der Nutzer sie über eine Bildschirmtastatur eingeben muss, während kurze Passwörter anfällig für Bruteforce-Angriffe sind. Zudem sind bei solchen Geräten weitere Angriffsarten wie etwa Shoulder Surfing (bei dem der Angreifer dem Opfer die geheimen Daten über die Schulter mitliest) wahrscheinlicher, da diese oft an öffentlichen Plätzen genutzt werden.

Android implementiert mehrere Funktionen zur Authentifizierung des Nutzers über die Bildschirmsperre [23]:

- **Entsperrmuster**: Der Nutzer legt auf einer 3x3-Matrix mindestens vier Punkte fest, die für erfolgreiche Authentifizierung miteinander verbinden werden müssen. Die Sicherheit dieses Mechanismus wird als sehr niedrig eingestuft; einerseits ist die Anzahl der möglichen Kombinationen sehr eingeschränkt, weshalb diese Authentifizierungsmethode nicht für das Ableiten eines geheimen Schlüssels (beispielsweise für die Speicherverschlüsselung) benutzt werden darf. Andererseits lässt sich das vorher eingegebene Muster mit bestimmten Lichtverhältnissen und Kameras anhand der Fingerabdrücke und Schmierflecken am Display ableiten (*smudge attack*). Wird das Muster mehr als fünf mal falsch eingegeben, so kann man sich mit den Anmeldedaten eines am Gerät registrierten Google-Kontos authentifizieren. Ist das dafür gewählte Passwort schwach, so kann dieses Mechanismus als Hintertür für das Gerät verwendet werden.

- **PIN/Passwort**: Der Nutzer legt eine numerische PIN oder ein alphanumerisches Passwort fest, das beim Entsperren eingegeben werden muss. Diese Information wird auch verwendet, um daraus einen geheimen Schlüssel zum Verschlüsseln des Gerätespeichers abzuleiten, da diese Methode zur Authentifizierung des Nutzers auf Android als eine der sichersten gilt.. Viele Benutzer wählen üblicherweise recht kurze PINs und Passwörter,

um nicht zu viel Zeit mit dem Entsperren des Geräts zu verbringen. Daher wird zum Ableiten des Schlüssels eine spezielle kryptographische Hashfunktion verwendet (*Key Derivation Function*), die die PIN oder das Passwort auf die notwendige Länge streckt. Seit Android 4.4 wird hierfür die *scrypt*-Funktion genutzt; seit Version 5.0 wird außer der PIN bzw. dem Passwort kann auch ein Geheimnis aus einem sicheren Hardwareelement auf dem Gerät (*Trusted Execution Environment*) verwendet werden, um Offline-Angriffe auf die Speicherverschlüsselung schwieriger zu gestalten.

Um Brute-Force-Angriffe auf dem Gerät (*online*-Angriffe) zu verhindern, bei dem der Angreifer jede mögliche PIN manuell eintippt, wird nach allen fünf falsch eingegebenen PINs eine Wartezeit von 30 Sekunden erzwungen. *Offline*-Angriffe auf ein verschlüsseltes Abbild des Gerätespeichers werden durch die Nutzung eines zusätzlichen Geheimnisses erschwert, das sich nur auf einem sicheren Hardwareelement des Geräts befindet und nicht extrahiert werden kann.

- **Biometrische Merkmale**: Android bietet in den neuesten Versionen eine Reihe von biometrischen Authentifizierungsmechanismen. Darunter zählt zum Einen die Gesichtserkennung: der Nutzer lässt sich von der Frontkamera des Geräts photographieren. Anschließend kann er (gute Lichtverhältnisse vorausgesetzt) sein Gerät mit seinem Gesicht entsperren. Diese Funktionalität ist sein Android 4.0 verfügbar, und gilt als eine der schwächsten Authentifizierungsmethoden, da die Bilderkennung nicht verlässlich funktioniert, und durch ein Photo des Besitzers überlistet werden kann. Ebenso kann die Stimme oder (seit Version 6.0) ein Fingerabdruck des Benutzers zur Authentifizierung verwendet werden. Auch diese Merkmale lassen sich je nach benutzter Hardware relativ einfach fälschen.

- **Vertraulicher Kontext**: Seit Android 5.0 kann auch ein "vertraulicher" Kontext als Authentifizierungsmerkmal verwendet werden. Darunter zählen "vertrauliche Umgebung" (wie etwa der Wohnort des Nutzers), wo das Gerät automatisch entsperrt wird, "vertrauliche Geräte" (wie Smart Watches), die das Gerät entsperren, oder "On-Body Detection", bei dem das Gerät entsperrt bleibt, bis es hingelegt wird. Solche Mechanismen erhöhen die Bedienungsfreundlichkeit auf Kosten der Sicherheit. Die aufgezählten Faktoren können in vielen Fällen leicht gefälscht werden, und deaktivieren effektiv andere Sicherheitsmaßnahmen wie die PIN-Eingabe.

Im Kontext der IFlow-Anwendungen ist es empfohlen, das Gerät mit einer PIN oder einem Passwort zu sperren und zu verschlüsseln.

Zudem sollten ausschließlich Geräte mit einem sicheren Element verwendet werden, sowie die aktuellste Version von Android eingespielt sein. Dadurch können Angriffe durch unautorisierte Benutzer deutlich erschwert werden.

18 VERWANDTE ARBEITEN

VERIFIKATION UND CODEANALYSE ZUR INFORMATIONSFLUSSKONTROLLE Said et al. [8] stellen einen modell-getriebenen Ansatz zur Verifikation der Informationsflusssicherheit verteilter Anwendungen vor. Die Anwendung wird mit *secBIP*, einer Sicherheitserweiterung von *BIP* [5] spezifiziert. Der Ansatz unterstützt das Garantieren der transitiven Nichtinterferenz sowohl für Systemereignisse als auch für Daten, indem zunächst die Sicherheit der verteilten Komponenten, und anschließend ihre Kommunikation und Komposition verifiziert wird. Die Sicherheit der lauffähigen Implementierung des modellierten und verifizierten Systems wird dabei nicht betrachtet.

Sfaxi et al. [86] definieren ein Framework zur Informationsflusskontrolle in verteilten Systemen mit dem Fokus auf automatische Verifikation der Nichtinterferenz-Eigenschaft. Der Ansatz nutzt Jif, um die Informationsflussicherheit des gesamten, verteilten Systems zu zeigen, sowie automatische Analyse der Implementierung der einzelnen Anwendungskomponenten, um sicherzustellen, dass diese die Sicherheitspolicy der Gesamtanwendung nicht verletzen. Automatisch generierte kryptographische Operationen stellen dabei die Sicherheit der übertragenen Komponentenkommunikation sicher.

Zur Informationsflussanalyse von Android-Apps gibt es die Ansätze JoDroid [67] sowie FlowDroid [2]. Beide führen statische Bytecodeanalyse einer einzelnen Android-App aus, wobei JoDroid die PDG-basierte Analyse von JOANA nutzt (vgl. Unterabschnitt 16.1.1), während FlowDroid auf Soot aufbaut, und *taint-tracking* verwendet. Zu vergleichbaren Ansätzen zählen auch [103], [102], und [6], deren Implementierungen jedoch nicht öffentlich verfügbar sind [2].

Bei taint-tracking *wird sensitive Information an ihrer Quelle markiert, und ihr Fluss durch die Anwendung verfolgt*

[56] und [33] schlagen jeweils einen Ansatz zur Analyse von Anwendungen, die Sicherheits-APIs für Krypto-Operationen nutzen. Dabei wird die funktionale Implementierung solcher Operationen durch ihre ideelle Implementierung ausgetauscht, und anschließend die Informationsflussanalyse mit JOANA ausgeführt. Dieser Ansatz wird auch in IFlow verfolgt, siehe Unterunterabschnitt 17.2.2.3.

BENUTZERZENTRISCHE SICHERHEIT [59] stellt einen Ansatz zur Spezifikation und Analyse von Informationsflusseigenschaften einer Android-App durch den Nutzer. Hierzu wurde ein App-Store implementiert, in dem der Nutzer für eine im Store angebotene App plattformspezifische Quellen und Senken festlegen kann. Anschließend prüft der Store, ob zwischen den ausgewählten Quellen und Senken ein Informationsfluss vorliegt. Dabei werden weder Deklassifikation noch anwendungsspezifische Quellen und Senken betrachtet.

[9] beschreibt einen anwendungsunabhängigen Ansatz zur sicheren Eingabe von sensitiven Benutzerdaten auf Android. Darin wird eine Erweiterung von Android vorgeschlagen, die für eine geöffnete App ihre Herkunft (wie etwa *PayPal, Inc* für die PayPal-App) auf dem Display des Nutzers anzeigt. Die in dem Ansatz vorgestellten Sicherheitsmechanismen, die Phishing-Angriffe verhindern sollen, wurden in [27] verbessert und erweitert. Jedoch sind diese Ansätze nicht mit einer Standardversion von Android umsetzbar.

Teil V

FALLSTUDIEN

19 TRAVELPLANNER

19.1 KURZBESCHREIBUNG

TravelPlanner ist eine verteilte Anwendung, die es ihrem Nutzer erlaubt, relevante Flugangebote für seine Reise zu finden. Mit Hilfe der *TravelPlanner*-App auf seinem mobilen Gerät kann der Nutzer den Webservice der Reiseagentur nach verfügbaren und geeigneten Flugangeboten befragen. Nachdem er ein Flugangebot ausgewählt hat, kann er den Flug direkt bei dem Webservice der Fluglinie buchen. Dazu veranlasst er die *CreditCardCenter*-App, die seine Kreditkartendaten veranlasst, diese für die Fluglinie freizugeben, und kann so für den Flug bezahlen. Die Fluglinie benachrichtigt anschließend die Reiseagentur über die erfolgreiche Buchung.

Die Fallstudie wurde mit Hilfe des IFlow-Ansatzes entwickelt. Dazu wurde das Anwendungsmodell mit MODELFLOW erstellt, und daraus der Code und das formale Modell generiert. Mit Hilfe von formaler Verifikation wurde gezeigt, dass der Webservice der Reiseagentur niemals die Kreditkartendaten des Nutzers erfährt. Dieselbe Eigenschaft konnte auch mit automatischer Informationsflussanalyse des Codeskeletts der Anwendung gezeigt werden. Zudem wurde die Eigenschaft formal verifiziert, dass der Webservice der Fluglinie die Kreditkartendaten erst nach expliziter Bestätigung des Nutzers erfährt.

Die *TravelPlanner*-Fallstudie demonstriert die folgenden Aspekte des IFlow-Ansatzes:

- Modellierung verteilter Anwendungen
- Interaktion mit dem Nutzer über die vordefinierte Benutzeroberfläche
- Modellierung, formale Verifikation, und automatischer Check einer transitive Nichtinterferenz-Eigenschaft
  (*Reiseagentur erfährt nicht die Kreditkartendaten*)
- Modellierung und formale Verifikation einer intransitive Nichtinterferenz-Eigenschaft
  (*Fluglinie erhält die Kreditkartendaten nur nach Deklassifikation*)
- Modellierung und formale Verifikation der *when*-Dimension der Deklassifikation
  (*Fluglinie erhält die Kreditkartendaten nur nach Benutzerbestätigung*)

19.2 ANWENDUNGSMODELL, FORMALES MODELL, UND CODE

Die im Rahmen der Fallstudie entstandenen Artefakte sind bereits in vorhergehenden Kapiteln im Detail beschrieben worden.

STATISCHE SICHT Abschnitt 10.5 beschreibt die Klassendiagramme der *TravelPlanner*-Anwendung. Abbildung 9 zeigt dabei die Anwendungskomponenten der Fallstudie, während Abbildung 10 ihre Nachrichtendatentypen abbildet.

DYNAMISCHE SICHT Abschnitt 11.4 beschreibt das Sequenzdiagramm der Fallstudie, der das Abfragen und die Auswahl der Flugangebote, die Deklassifikation der Kreditkartendaten, und die Flugbuchung modelliert.

INFORMATIONSFLUSSMODELLIERUNG Abschnitt 12.1 erläutert die Modellierung der Sicherheitsdomänen und ihrer Relation, sowie die Annotation der Diagramme der Fallstudie, während Unterabschnitt 12.2.1 und Unterabschnitt 12.2.2 ihre Informationsflusseigenschaften zeigt.

FORMALES MODELL UND CODE Unterabschnitt 15.2.1 beschreibt das formale Modell in IFlow anhand der *TravelPlanner*-Fallstudie, und zeigt Ausschnitte aus der Spezifikation der Anwendungskomponenten und der komplexen Datentypen.

Unterabschnitt 16.2.1 erläutert den Aufbau des Codeskeletts der Anwendung; Listing 4 zeigt hierzu die Java-Klasse, die aus einem modellierten komplexen Datentyp generiert wurde, während Abbildung 24 den Code der *Airline*-Komponente zeigt.

20 BANKINGAPP

20.1 KURZBESCHREIBUNG

BankingApp ist eine simple Anwendung, die es ihrem Nutzer ermöglicht, mit Hilfe der mobilen *BankingApp* mit seinem Konto zu interagieren. Die Kontostände aller Nutzer werden dabei vom *Bank*-Webservice verwaltet. Der Nutzer kann Punkte aufladen, diese wieder von seinem Konto abheben, und den aktuellen Kontostand anzeigen lassen.

Die Fallstudie wurde mit dem IFlow-Ansatz entwickelt. Mit Hilfe der automatischen Informationsflussanalyse des Codeskeletts der sicheren Variante der Anwendung (siehe Unterabschnitt 12.2.4) wurde garantiert, dass sich die Nutzer der Anwendung nicht gegenseitig beeinflussen. Dadurch gilt insbesondere, dass weder ein legitimer Nutzer, noch ein Angreifer den Kontostand eines anderen Benutzers erfahren kann. Gleichzeitig wurde für die unsichere Variante der Anwendung gezeigt, dass ein Angreifer den Kontostand eines legitimen Nutzers in Erfahrung bringen kann.

Die *BankingApp*-Fallstudie demonstriert die folgenden Aspekte des IFlow-Ansatzes:

- Modellierung verteilter Anwendungen mit mehreren Abläufen
- Integration einer manuellen Benutzeroberfläche in eine IFlow-Anwendung
- Modellierung und automatischer Check der transitiven Nichtinterferenz zwischen Anwendungsnutzern
  (*BankingApp leakt keine Daten an andere Nutzer des Systems*)
- Betrachtung eines aktiven Angreifers

20.2 ANWENDUNGSMODELL UND CODE

Der Aufbau und das Modell der Anwendung wurden bereits in Unterabschnitt 12.2.4 angesprochen. Die Fallstudie besteht aus der mobilen Anwendung *BankingApp* sowie einem Webservice *Bank*, und bietet ihrem Nutzer die folgenden Funktionalitäten: (1) Aufladen des Online-Kontos, (2) Abheben vom Online-Konto, sowie (3) Abfrage des aktuellen Kontostandes.

STATISCHE SICHT Abbildung 28 zeigt das Klassendiagramm der Fallstudie, das die Anwendungskomponenten *BankingApp* und *Bank*,

sowie das Anwendungsmodul *GUI* spezifiziert, das eine manuelle Benutzeroberfläche abbildet. Zudem ist darin der komplexe Datentyp *AuthData* modelliert, das die Authentifikationsdaten eines Nutzers bestehend aus seinem Benutzernamen und Passwort repräsentiert, sowie der Datentyp *Account*, der den Kontostand eines Nutzers speichert. In Abbildung 29 sind die Nachrichtendatentypen der Fallstudie abgebildet.

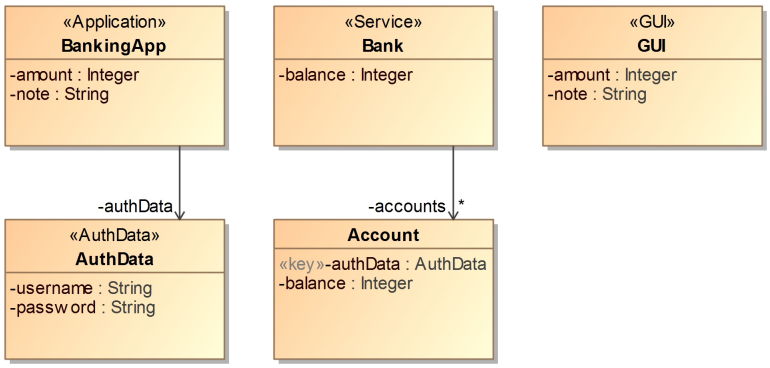


Abbildung 28: Komponenten und Datentypen der Banking-Fallstudie

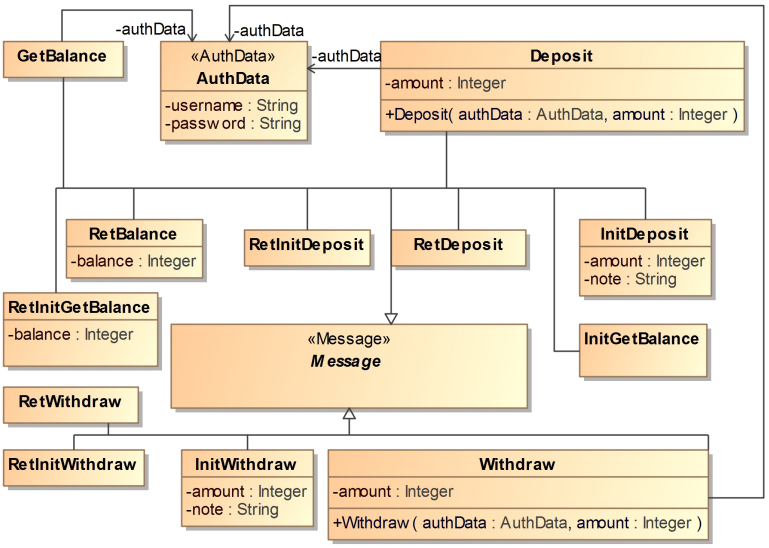


Abbildung 29: Nachrichtendatentypen der Banking-Fallstudie

DYNAMISCHE SICHT Abbildung 30 zeigt das Sequenzdiagramm, das den Programmablauf zum Aufladen des Online-Kontos modelliert. Darin meldet sich die *BankingApp* mit Hilfe der Authentifikationsdaten des Nutzers bei der *Bank* an, und gibt den aufzuladenden Betrag an, den sie von der manuellen Benutzeroberfläche erhalten hat. Abbildung 31 modelliert den Programmablauf zum Abheben von Punkten. Auch darin meldet sich die *BankingApp* mit den Au-

thentifikationsdaten des Nutzers bei der *Bank* an, und gibt den abzuhebenden Betrag an, den sie von der manuellen Benutzeroberfläche erhalten hat. Abbildung 32 und Abbildung 33 zeigen jeweils die sichere und die unsichere Versionen der Programmabläufe zur Kontostandabfrage, die in Unterabschnitt 12.2.4 im Detail diskutiert werden. Die Informationsflusseigenschaft „BankingApp leakt keine Daten an andere Nutzer des Systems“ ist dabei in Abbildung 19 abgebildet.

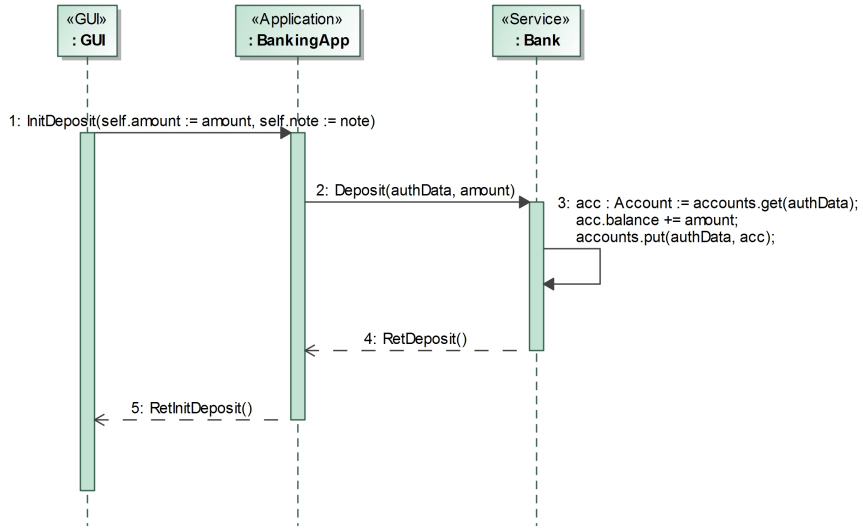


Abbildung 30: Aufladen des Online-Kontos in der Banking-Fallstudie

CODE Der Aufbau des Codeskeletts, der den automatischen Check der Informationsflüsse zwischen verschiedenen Nutzern ermöglicht, wird in Unterunterabschnitt 17.2.1.5 erläutert, während die Abstraktion einer manuellen Benutzeroberfläche in Unterunterabschnitt 17.2.2.5 diskutiert wird.

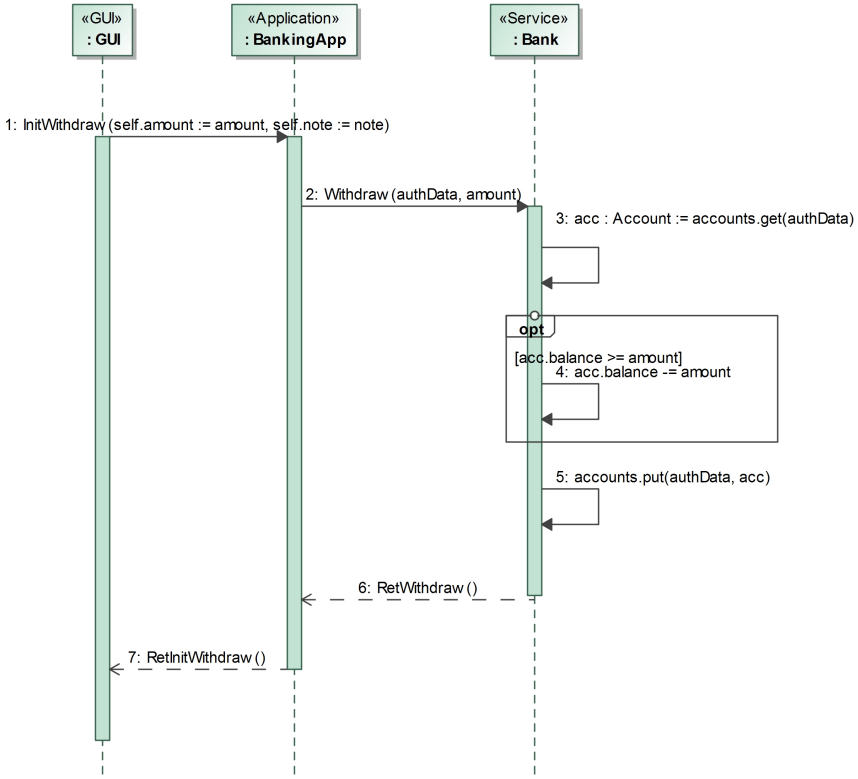


Abbildung 31: Abheben vom Online-Konto in der Banking-Fallstudie

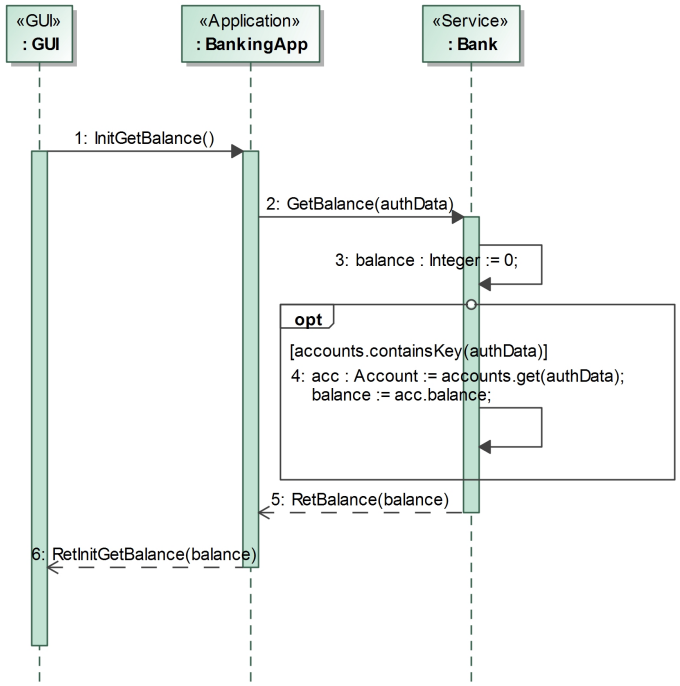


Abbildung 32: Kontostand-Abfrage in der Banking-Fallstudie

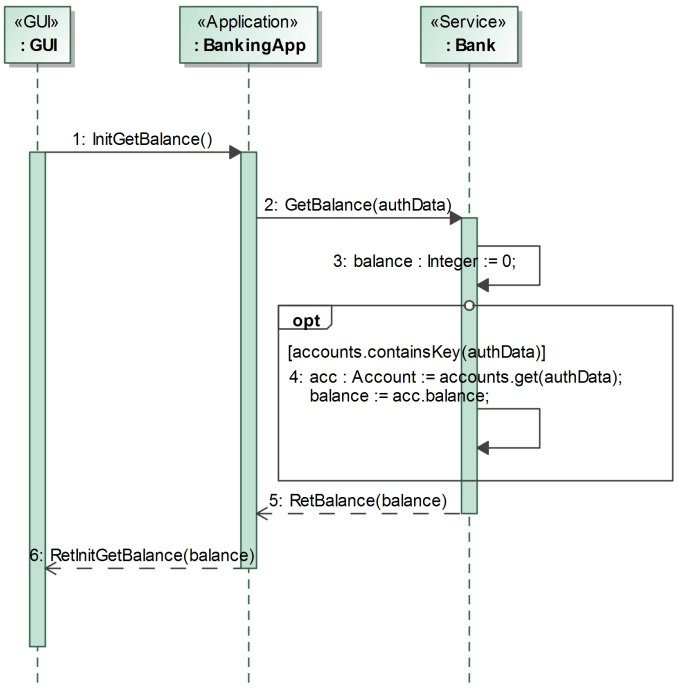


Abbildung 33: Unsichere Kontostand-Abfrage in einer Variante der Banking-Fallstudie

21 DISTANCETRACKER

21.1 KURZBESCHREIBUNG

DistanceTracker ist eine Jogging-App, die mit dem IFlow-Ansatz entwickelt wurde. Der Nutzer kann mit der gleichnamigen, mobilen Anwendung *DistanceTracker* das GPS-Tracking vor seinem Lauftraining starten, und die App beginnt, die aktuelle Position periodisch aufzuzeichnen. Sobald der Benutzer das Tracking anhält, wird aus der Liste der aufgezeichneten GPS-Positionen eine Strecke berechnet, die an den Webservice *TrackerService* gesandt wird (um dort z.B. mit anderen Nutzern verglichen zu werden).

Für die Fallstudie wurde formal bewiesen, dass *TrackerService* nur die gelaufene Strecke des Nutzers erfährt. Dieselbe Eigenschaft konnte auch mit automatischer Informationsflussanalyse gezeigt werden. Zudem wurde bewiesen, dass *TrackerService* aus den von der App erhaltenen Daten nicht die aktuelle oder frühere GPS-Position des Nutzers schließen kann.

Die *DistanceTracker*-Fallstudie demonstriert insbesondere die folgenden Aspekte des IFlow-Ansatzes:

- Nutzung vordefinierter Operationen zur Implementierung plattformspezifischer Funktionalität sowie mathematischer Berechnungen
  (Zugriff auf den GPS-Sensor und Berechnung sphärischer Trigonometrie)
- Modellierung von MEL*-Methoden
- Formale Verifikation und automatischer Check einer intransitiven Nichtinterferenz-Eigenschaft
  (*TrackerService erfährt Information über Position des Nutzers nur nach ihrer Anonymisierung bzw. Filterung*)
- Formale Verifikation einer Deklassifikationseigenschaft bzgl. der der *what*-Dimension der Deklassifikation
  (*TrackerService kann aus den anonymisierten bzw. gefilterten Informationen nicht auf die Position des Nutzers schließen*)

21.2 ANWENDUNGSMODELL UND EIGENSCHAFTEN

STATISCHE SICHT Abbildung 34 zeigt das Klassendiagramm der DistanceTracker-Fallstudie, die ihre Anwendungskomponenten *DistanceTracker* (mobile App) und *TrackerService* (Webservice), sowie ei-

ne Reihe der in der Anwendung benötigten, komplexen Datentypen spezifiziert. Dazu zählt u.a. *Distance*, das die gelaufene Strecke des Nutzers abbildet. Abbildung 35 zeigt die in der Anwendung verwendeten Nachrichtendatentypen.

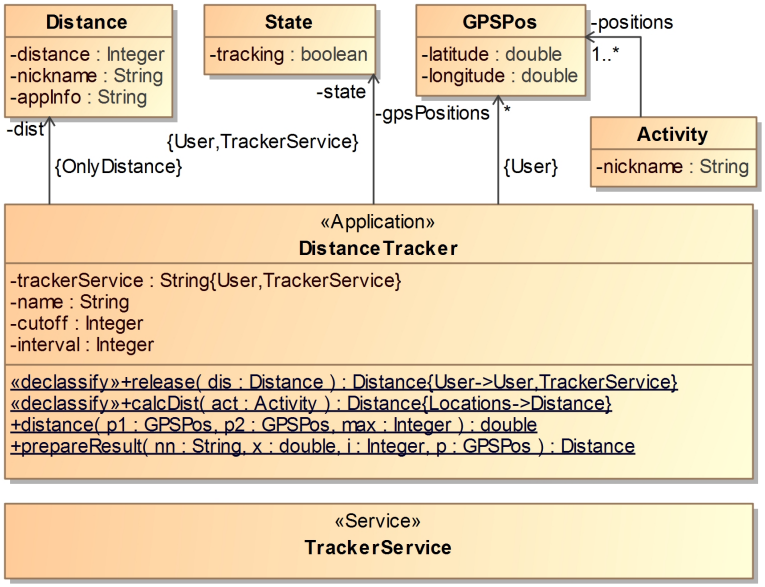


Abbildung 34: Komponenten und Datentypen der DistanceTracker-Fallstudie

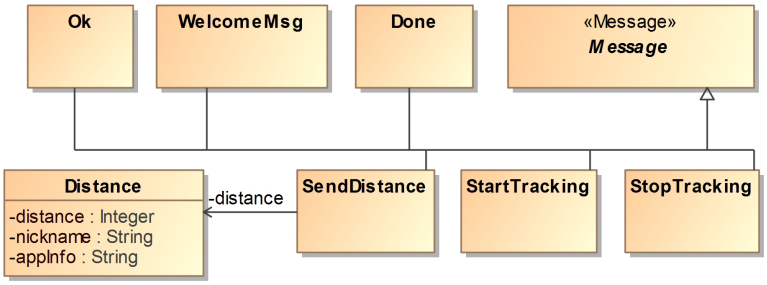


Abbildung 35: Nachrichtendatentypen der DistanceTracker-Fallstudie

INFORMATIONSFLUSS Die Attribute der Anwendungskomponenten sind mit Sicherheitsdomänen annotiert, die zusammen mit ihrer Interferenzrelation in Abbildung 36 spezifiziert sind. Da die Anwendung zwei Deklassifikationsfunktionen *calcDist* (zur Berechnung der gelaufenen Strecke aus einer Liste von GPS-Positionen) und *release* (zur Freigabe dieser Strecke an den Webservice nach expliziter Benutzerbestätigung) benötigt, sind zwei Deklassifikationsdomänen *{Locations->Distance}* sowie *{User->User,TrackerService}* definiert. Die aufgezeichneten GPS-Positionen (gespeichert im Attribut *gpsLocations* von *DistanceTracker*, vgl. Abbildung 34) erhalten dabei die geheimste

Sicherheitsdomäne *{User}*, die mit *calcDist* zur Domäne *{OnlyDistance}* deklassifiziert werden können, um anschließend mit *release* für den Webservice mit der öffentlichsten Domäne *{User,TrackerService}* freigegeben zu werden.

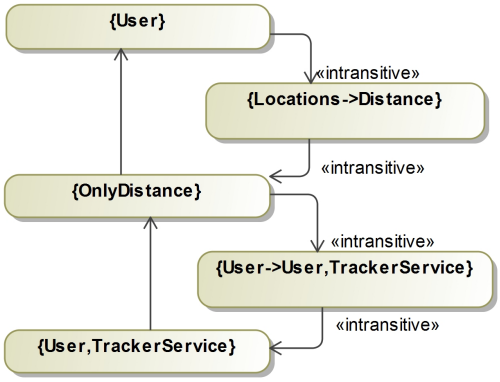


Abbildung 36: Sicherheitsdomänen und Interferenzrelation der DistanceTracker-Fallstudie

Abbildung 37 zeigt die modellierte Informationsflusseigenschaft der Anwendung, dass *TrackerService* die von der GPS-Position abgeleiteten Informationen nur nach expliziter Deklassifikation und Nutzerbestätigung erhält. Abbildung 38 bildet die modellierte Informationsflusseigenschaft ab, die besagt, dass *TrackerService* die GPS-Position des Nutzers nicht erfährt, sondern nur seine gelaufene Distanz, die mit der sicheren Filterfunktion *calcDist* berechnet wurde.

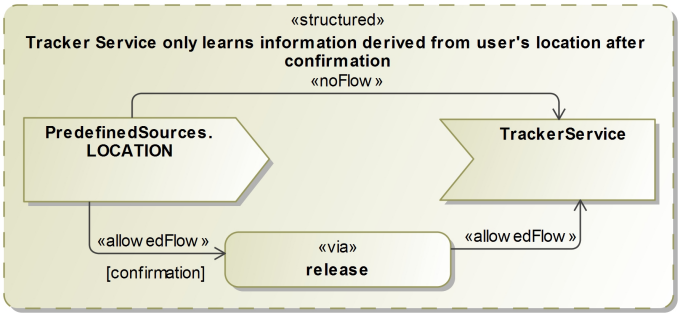


Abbildung 37: Informationsflusseigenschaft der DistanceTracker-Fallstudie

Zur Verifikation der Sicherheit der Filterfunktion *calcDist* muss man das in Abbildung 39 beschriebene Schema instantiieren. Die formalisierte Eigenschaft ist in Abbildung 39 abgebildet. Sie besagt, dass es für jede Ausgabe von *calcDist* über 10,000 mögliche Eingaben gibt, zwischen denen ein öffentlicher Beobachter der Ausgabe nicht unterscheiden kann. Dabei muss jedes Element der Eingabeliste eine gültige GPS-Position darstellen, und alle möglichen Eingabelisten, die zur selben Ausgabe führen, keine gemeinsamen Elemente haben.

Siehe Stenzel et al. für eine detaillierte Diskussion der Sicherheit von calcDist *[93]*

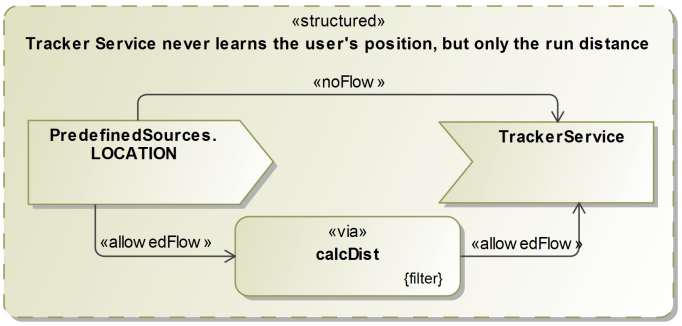


Abbildung 38: Informationsflusseigenschaft der DistanceTracker-Fallstudie

Dadurch wird sichergestellt, dass sich diese Eingaben wirklich unterscheiden, und die Funktion keine geheimen Informationen über die Eingabe leakt.

$\forall$ in, out. **ff**(in) = out $\wedge$ **assumption** (in, out) $\rightarrow$
$\exists$ set. size(set) $>$ 10.000 $\wedge$
($\forall$ a. a $\in$ set $\rightarrow$ **ff**(a) = out $\wedge$ **property(a)** $\wedge$
($\forall$ a, b. a $\in$ set $\wedge$ b $\in$ set $\wedge$ a $\neq$ b $\rightarrow$ **enoughDifferences**(a, b)))

assumption(in, out) $\equiv$ out = true
property(a) $\equiv$ „all GPS positions in the input are valid degrees on earth“
enoughDifferences(a, b) $\equiv$ „the GPS positions of a and b are mutually disjoint“

Abbildung 39: Eigenschaft der Filterfunktion *calcDist*

DYNAMISCHE SICHT Abbildung 40 zeigt das Sequenzdiagramm der Anwendung. Darin wird der Nutzer zum Starten bzw. Anhalten des GPS-Trackings aufgefordert, wofür die vordefinierten Operationen *startGPSTracking* und *stopGPSTracking* verwendet werden. Die Liste der aufgezeichneten GPS-Positionen wird dabei im Attribut *gpsLocations* von *DistanceTracker* gespeichert. Daraus wird mit Hilfe der *calcDist*-Funktion die gelaufene Strecke berechnet, die nach der Benutzerbestätigung und expliziter Deklassifikation durch die *release*-Funktion an den *TrackerService* geschickt wird.

Abbildung 41 zeigt das Aktivitätsdiagramm, das die MEL*-Methode *calcDecl* spezifiziert. Diese erhält eine Liste von GPS-Positionen, die sie in einer *for*-Schleife durchläuft, zwischen je zwei Positionen die Distanz berechnet und zu einer laufenden Summe aufaddiert. *distance* implementiert dabei die Haversine-Formel zur Berechnung der Distanz zwischen zwei Punkten auf der Erde (vgl. Abbildung 43), während *prepareResult* die gelaufene Distanz zurücksetzt, wenn sie zu groß wird.

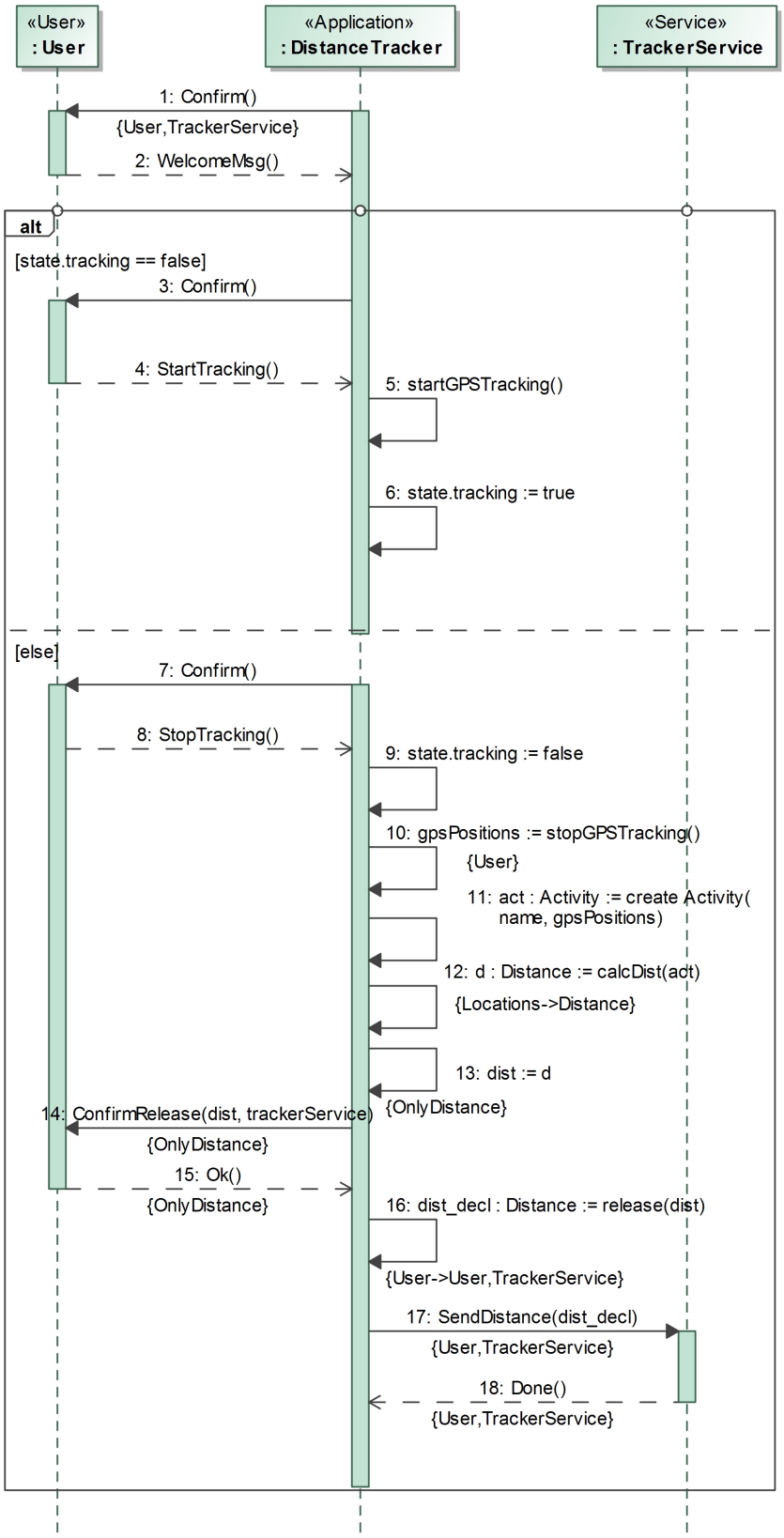


Abbildung 40: Sequenzdiagramm der DistanceTracker-Fallstudie

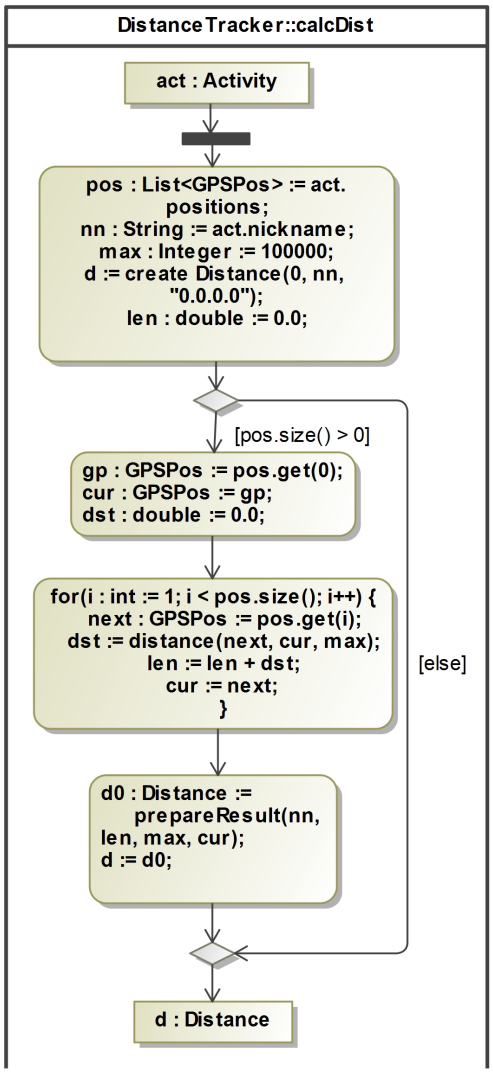


Abbildung 41: Spezifikation der MEL*-Methode *calcDist*

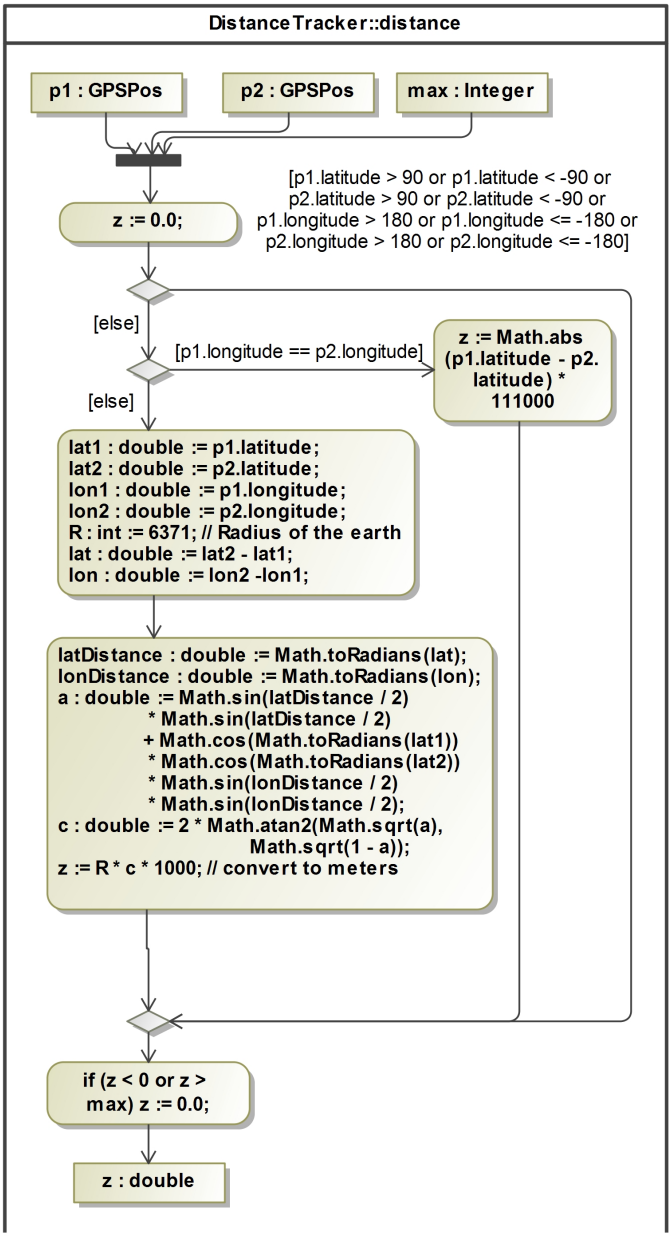


Abbildung 42: Spezifikation der MEL*-Methode *distance*

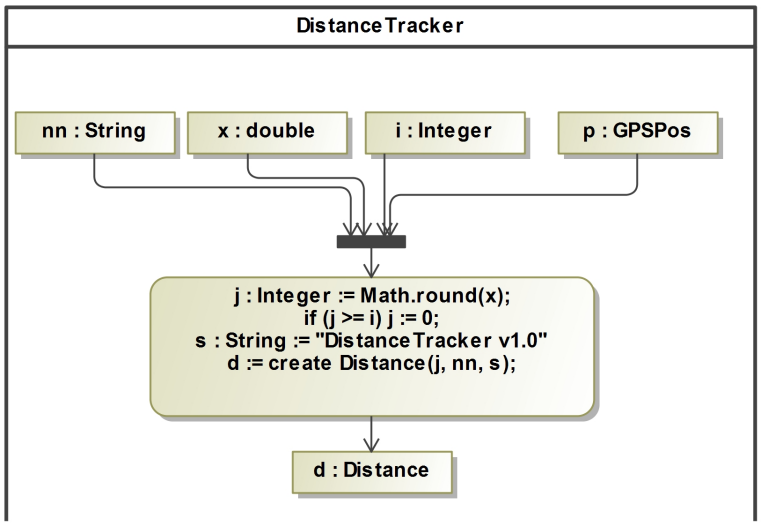


Abbildung 43: Spezifikation der MEL*-Methode *distance*

22 CONTACTSMSMANAGER

22.1 KURZBESCHREIBUNG

ContactSMSManager ist eine IFlow-Anwendung bestehend aus zwei mobilen Apps: *ContactManager* zum Verwalten von Kontaktdaten, sowie *SMSManager* zum Versenden von SMS. Mit *ContactManager* kann der Nutzer neue Kontakte anlegen, sowie bereits vorhandene Kontakte ansehen oder löschen. Zudem kann er den *ContactManager* veranlassen, mit Hilfe der *SMSManager*-App eine SMS an einen der ausgewählten Kontakte zu versenden.

Für die Fallstudie wurde mit Hilfe automatischer Informationsflussanalyse auf Codeebene gezeigt, dass die Anwendung nicht die personalisierten Kontaktdaten des Nutzers leakt.

Die *ContactSMSManager*-Fallstudie demonstriert insbesondere die folgenden Aspekte des IFlow-Ansatzes:

- Modellierung und automatischer Check einer Informationsflusseigenschaft bzgl. der vom Nutzer eingegebenen Daten („Kontaktnamen werden von der Anwendung nicht als Inhalt einer SMS verschickt“)
- Nutzung einer vordefinierten graphischen Oberfläche zur sicheren Benutzereingabe
- Nutzung einer manuellen Benutzeroberfäche
- Modellierung von MEL*-Methoden

22.2 ANWENDUNGSMODELL UND CODE

STATISCHE SICHT Das Klassendiagramm in Abbildung 44 zeigt die Komponenten und komplexen Datentypen der *ContactSMSManager*-Fallstudie. *SMSManager* und *ContactManager* bildet die mobilen Apps der Anwendung zum Verwalten von Kontaktdaten und Versand von SMS ab. Die komplexen Datentypen *Contact* und *SMSBody* repräsentieren dabei jeweils einen Kontakt (bestehend aus dem Namen, Vornamen, und der Telefonnummer), bzw. den Text einer SMS-Nachricht. Die Datentypen erhalten dabei «`label`»-Annotationen wie *„Contact details“*, die ihre Bedeutung dokumentieren, und dem Nutzer bei der Eingabeaufforderung angezeigt werden. Zudem definiert das Diagramm u.a. die Deklassifikationsmethode *removeName*, die die Kontaktdaten vor dem SMS-Versand filtert, sowie die manuelle Me-

thode *executeSendSMS*, die den Versand einer SMS implementieren soll, und daher auf die vordefinierte Senke *SMS_MMS* zugreifen darf.

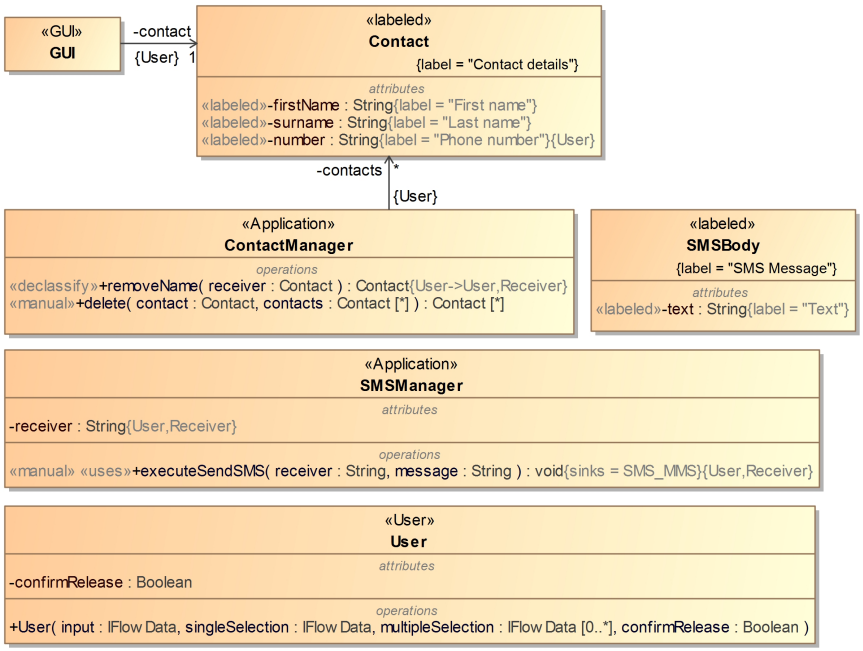


Abbildung 44: Komponenten und Datentypen der ContactSMSManager-Fallstudie

Abbildung 45 zeigt die Nachrichtendatentypen, die für die Inter-App-Kommunikation benötigt werden.

INFORMATIONSFLUSS Abbildung 46 zeigt die Sicherheitsdomänen und ihre Interferenzrelation. Mit der Domäne *{User}* werden die geheimen Kontaktdaten annotiert, die im Attribut *contacts* der *ContactManager*-Anwendungskomponente gespeichert werden (vgl. Abbildung 44). Diese können explizit zur öffentlichen Domäne *{User,Receiver}* mit der Filtermethode *removeName* deklassifiziert werden, um mit der *executeSendSMS*-Methode als SMS versandt werden zu können.

Abbildung 47 zeigt die explizite Informationsflusseigenschaft der Anwendung, die den direkten Informationsfluss der vom Benutzer eingegebenen Kontaktdaten (*„The new contact“*) zur vordefinierten Senke *PredefinedSinks.SMS_MMS* verbietet, diesen aber nach dem Filtern mit der Methode *removeName* erlaubt. Dies ist notwendig, da die Kontaktdaten auch die Telefonnummer enthalten, die zum Versand der SMS benötigt werden.

Abbildung 48 zeigt dabei die Prädikate zur Instantiierung des in Unterunterabschnitt 15.2.2.2 beschriebenen Schemas zur Verifikation der Filterfunktion *removeName*. Sie sagen aus, dass es für jede gefilterte Ausgabe eine Äquivalenzklasse von Eingaben mit derselben Telefonnummer, aber verschiedenen Vor- und Nachnamen gibt. Die

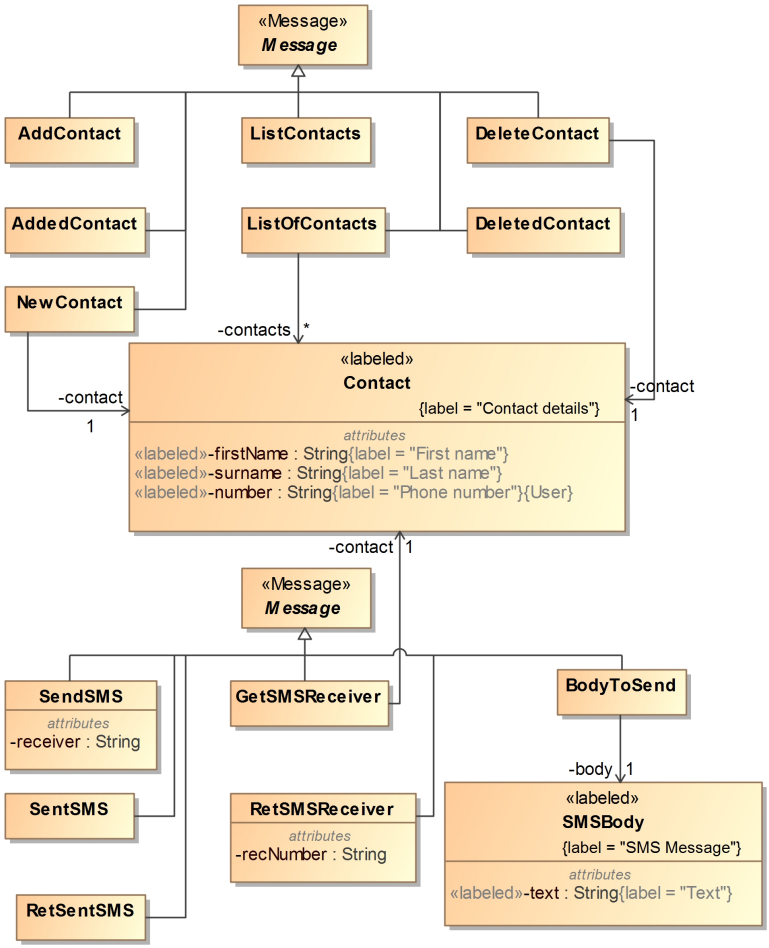


Abbildung 45: Nachrichtendatentypen und Datentypen der ContactSMSManager-Fallstudie

Äquivalenzklasse muss ausreichend groß sein, damit die Filterfunktion als sicher gilt; ein öffentlicher Beobachter kann dann allein anhand der Ausgabe dieser Funktion nicht auf den Vor- und Nachnamen des Besitzers der Telefonnummer schließen.

DYNAMISCHE SICHT Abbildung 49, Abbildung 50, und Abbildung 51 zeigen die Sequenzdiagramme der Fallstudie zum Hinzufügen, Löschen, und Anzeigen der von *ContactManager* verwalteten Kontaktdaten. Dabei interagiert die App mit einer manuellen Benutzeroberfläche (*GUI*), während die Eingabe der Kontaktdaten über die sichere, vordefinierte Benutzeroberfläche erfolgt. Dies geschieht durch die Rückantwort *NewContact* des *User*-Moduls, das mit dem Text *„The new contact“* annotiert ist (vgl. Nachricht 3 in Abbildung 49). Somit bezieht sich die Informationsflusseigenschaft in Abbildung 47 auf den Inhalt dieser Nachricht.

Abbildung 52 zeigt das Sequenzdiagramm zum Versand einer SMS. Darin wird ein Kontakt mit der Funktion *removeName* explizit de-

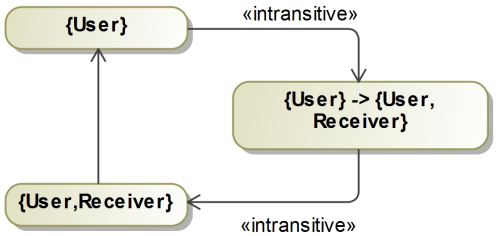


Abbildung 46: Sicherheitsdomänen und Interferenzrelation der ContactSMSManager-Fallstudie

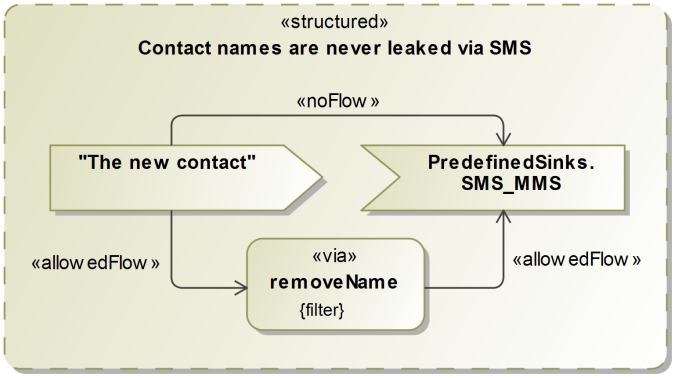


Abbildung 47: Informationsflusseigenschaft der ContactSMSManager-Fallstudie

klassifiziert und gefiltert, indem ihr Inhalt bis auf die Telefonnummer verworfen wird (vgl. Abbildung 53), wonach es an die mobile Anwendungskomponente *SMSManager* weitergeleitet wird. Diese fordert vom Nutzer die Eingabe des Nachrichtentextes, wonach sie diesen mit der manuellen Methode *executeSendSMS* an den Kontakt verschickt.

MANUELLE IMPLEMENTIERUNG Die in der Fallstudie eingesetzten manuellen Anwendungsmodule *GUI* und *executeSendSMS* wurden implementiert, so dass die finale Anwendung auf echter Hardware lauffähig ist. Der Umfang dieser Implementierung stellt etwa 14% des automatisch generierten Java-Codes dar. Die manuelle Implementierung wurde mit *MMChecker* (siehe Unterabschnitt 16.2.3)

224 Zeilen Code für die manuelle Implementierung vs. 1534 Zeilen Code für das automatisch generierte Codeskelett

assumption(in, out) ≡ true
property(a) ≡ a.number = in.number
enoughDifferences(a, b) ≡ a.firstName ≠ b.firstName ∧ a.surname ≠ b.surname

Abbildung 48: Prädikate der Sicherheitseigenschaft der Filterfunktion *removeName*

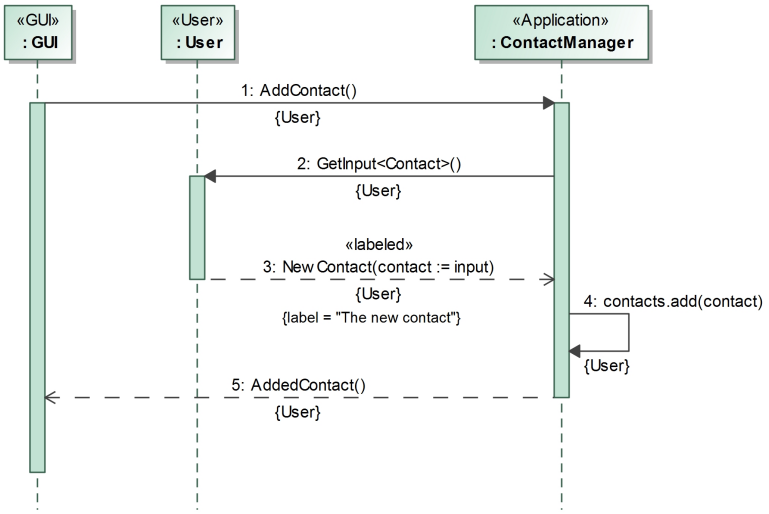


Abbildung 49: Sequenzdiagramm der ContactSMSManager-Fallstudie zum Hinzufügen von Kontakten

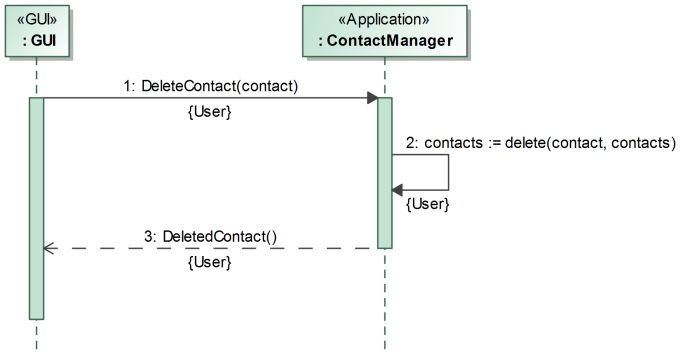


Abbildung 50: Sequenzdiagramm der ContactSMSManager-Fallstudie zum Löschen von Kontakten

auf die im Modell spezifizierten Zugriffseinschränkungen erfolgreich geprüft, und dabei keine Fehler festgestellt.

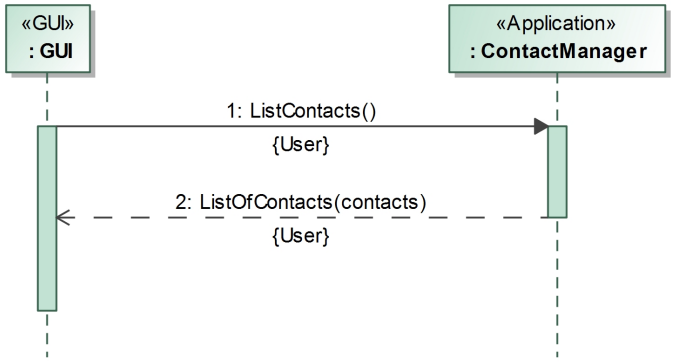


Abbildung 51: Sequenzdiagramm der ContactSMSManager-Fallstudie zum Anzeigen von Kontakten

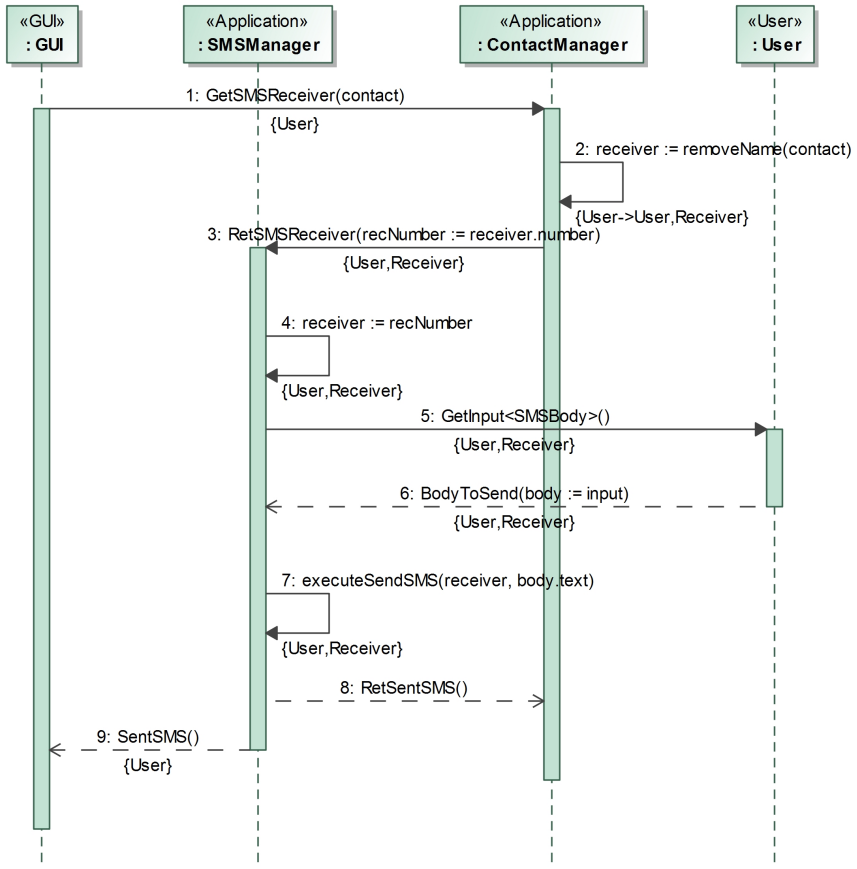


Abbildung 52: Sequenzdiagramm der ContactSMSManager-Fallstudie zum Versand einer SMS

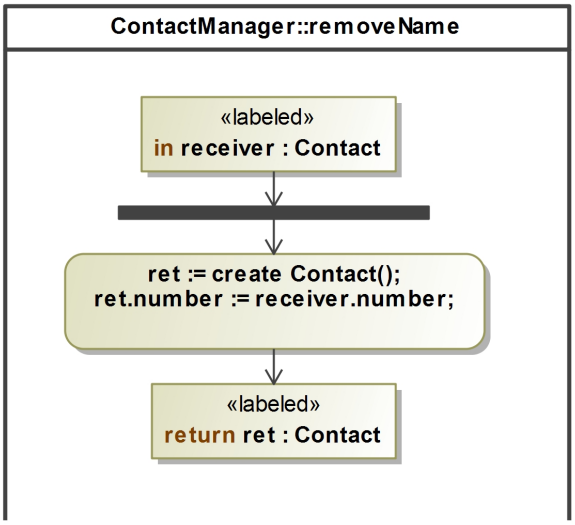


Abbildung 53: Aktivitätsdiagramm der ContactSMSManager-Fallstudie zum Filtern eines Kontakts

23 PRIVATETAXI

23.1 KURZBESCHREIBUNG

PrivateTaxi ist die größte Anwendung, die im Rahmen des IFlow-Projekts modelliert worden ist. Sie ermöglichst es ihren Nutzern, Fahrgemeinschaften zu bilden, indem sie Fahrer und potenzielle Mitfahrer anhand ihrer angegebenen Routen miteinander in Verbindung setzt.

Die Anwendung besteht aus den mobilen Apps *PrivateRider* für Mitfahrer und *PrivateRider* für Fahrer, sowie den zwei Webservices *PrivateTaxiService* und *CalcDistanceService* zur gegenseitigen Zuordnung der Nutzer bzw. Berechnung der Distanz zwischen den Routen zweier Nutzer der Anwendung. Die Fahrer und Mitfahrer können sich bei *PrivateTaxiService* für den Dienst registrierten, und anschließend ihre geplanten Reiserouten angeben. Mit Hilfe der *PrivateRider*-App kann sich der Mitfahrer die verfügbaren Fahrer anzeigen lassen, die eine ähnliche Route haben, und einen dieser Fahrer auswählen. Der Fahrer erfährt dadurch die Route des Mitfahrers, und kann diesen an seinem Startpunkt abholen. Dabei soll keiner der beteiligten Webservices gleichzeitig sowohl die Kontaktdaten als auch die Strecke der Nutzer in Erfahrung bringen können.

Die *PrivateTaxi*-Fallstudie demonstriert insbesondere die folgenden Aspekte der Modellierung mit MODELFLOW:

- Nutzung von Kryptographie, um Informationsflusseigenschaften zu garantieren
  (*PrivateTaxiService* erfährt nicht die Routen seiner Nutzer, da er diese nicht entschlüsseln kann)
- Integration einer externen Anwendung
  (Der Nutzer kann seinen Zielort mit Hilfe einer externen Karten-App angeben)
- Integration einer manuellen Benutzeroberfläche
- Modellierung von MEL*-Methoden
- Nutzung von Filtermethoden, deren Eingabe von einem Angreifer wählbar ist
  (*CalcDistanceService* implementiert eine Filtermethode, deren Eingabe aus den Nachrichten an den Service abgeleitet wird)

23.2 ANWENDUNGSMODELL

STATISCHE SICHT Abbildung 54 zeigt die Komponenten und Datentypen der *PrivateTaxi*-Fallstudie. Zusätzlich zu den mobilen Apps *PrivateRider* (für den Mitfahrer) und *PrivateDriver* (für den Fahrer) definiert das Diagramm die Webservices *PrivateTaxiService* (der die Fahrer und Mitfahrer einander zuordnet) sowie *CalcDistanceService* (der die Distanz zwischen einem Fahrer und einem potenziellen Mitfahrer berechnet). *CalcDistanceService* erhält dabei u.a. eine Deklassifikations-MEL*-Methode *isGoodMatch*, die als Eingabe die Routen zweier Nutzer erhält, und berechnet, ob diese nahe genug aneinander liegen. Mit der externen App *PlacePicker* kann der Nutzer sein Ziel auf einer Karte angeben.

Zudem spezifiziert das Klassendiagramm u.a. vier komplexe, beschriftete Datentypen *RegRiderInfo* und *RegDriverInfo*, die die Registrationsdetails der Fahrer bzw. des Mitfahrers repräsentieren, sowie *RegRiderPass* und *RegDriverPass*, die ihre gewählten Passwörter abbilden. Die Klasse *Match* repräsentiert eine Zuordnung eines Fahrers zu einem Mitfahrer, und speichert ihre Identifikationsdaten sowie öffentliche Schlüssel, *Route* bildet die Route eines Nutzers ab, während die Datentypen *Activity* und *Distance* bereits aus der Fallstudie *DistanceTracker* bekannt sind. Der Fokus der Fallstudie liegt auf automatischer Informationsflussanalyse der Anwendung, weshalb keine Sicherheitsannotationen benötigt werden.

Abbildung 55 zeigt die Nachrichtendatentypen, die für die Kommunikation der Anwendungskomponenten benötigt werden.

DYNAMISCHE SICHT Das Modell der Fallstudie definiert neun Sequenz- und elf Aktivitätsdiagramme, um die folgende Funktionalität abzubilden. Diese sind wegen ihrer Größe im Anhang abgebildet, siehe Anhang C.

- Die Fahrer und Mitfahrer legen auf dem *PrivateTaxiService* jeweils ein Konto an, wobei sie ihre Namen und Passwörter über eine sichere Benutzeroberfläche eingeben (Abbildung 60 und Abbildung 61)
- Die Fahrer und Mitfahrer holen den öffentlichen Schlüssel des *CalcDistanceService* ab, mit dem sie ihre Routen verschießen können (Abbildung 62 und Abbildung 63)
- Die Fahrer und Mitfahrer bestimmen ihre Route, indem sie ihre Zielposition mit Hilfe der externen App *PlacePicker* auswählen, während ihre Startposition ihrer aktuellen GPS-Position entspricht. Sie verschlüsseln ihre Route mit dem öffentlichen Schlüssel der *CalcDistanceService*, und veröffentlichen den resultierenden Geheimtext bei *PrivateTaxiService* (Abbildung 64 und Abbildung 65)

- Der Mitfahrer erkundigt sich bei *PrivateTaxiService*, welche Fahrer sich in seiner Nähe befinden, und in dieselbe Richtung fahren. Dazu vergleicht *PrivateTaxiService* die Route des Mitfahrers mit der Route von jedem angemeldeten Fahrer mit Hilfe des *CalcDistanceService*, und erstellt eine Liste von Vorschlägen (inklusive Namen und öffentliche Schlüssel der vorgeschlagenen Fahrer), die an den Mitfahrer geschickt werden (Abbildung 66). *CalcDistanceService* ist in der Lage, die Routen der Nutzer zu entschlüsseln, und nutzt zur Entscheidung, ob zwei Nutzer einander zugeordnet werden sollen, die Filtermethode *isGoodMatch* (Abbildung 69). Die von *isGoodMatch* genutzten MEL*-Methoden *calcDist*. *distance*, und *prepareResult* sind identisch zu den gleichnamigen Funktionen aus der *DistanceTracker*-Fallstudie (siehe Kapitel 21)
- Der Mitfahrer wählt einen der vorgeschlagenen Fahrer aus, verschlüsselt seine Route mit dem öffentlichen Schlüssel dieses Fahrers, und veröffentlicht das Resultat bei *PrivateTaxiService* (Abbildung 67)
- Der Fahrer erhält von *PrivateTaxiService* die Informationen über den Mitfahrer, der ihn ausgewählt hatte, kann seine Route mit dem eigenen geheimen Schlüssel entschlüsseln, und ihn anschließend von seiner Startposition abholen (Abbildung 68)

Die Hilfs-MEL*-Methoden zum Verwalten von Zuordnungen zwischen Fahrern und Mitfahrern sind in Abschnitt C.3 abgebildet.

INFORMATIONSFLUSS Die Hauptsicherheitseigenschaft der Fallstudie sagt aus, dass *PrivateTaxiService* die Namen der Nutzer erfährt, aber nicht ihre Routen, während *CalcDistanceService* die Routen der Nutzer einsehen kann, jedoch nicht ihre Namen. Dies soll sicherstellen, dass keines der Webservices den Standort ihrer identifizierten Nutzer tracken können.

Abbildung 56 besagt, dass *PrivateTaxiService* über die Route eines Nutzer (d.h., seinen Startort und den mit *PlacePicker* angegebenen Zielort) nur die Ausgabe der Filterfunktion *isGoodMatch* erfährt. Abbildung 57 legt fest, dass *CalcDistanceService* nie die persönlichen Registrierungsdaten wie etwa die Namen der Nutzer erfährt.

23.3 INFORMATIONSFLUSSANALYSE

23.3.1 *Automatische Codeanalyse*

Für eine Reihe von manuell reduzierten Versionen des Codeskeletts bestehend aus einer Untermenge der modellierten Abläufe konnte mit Hilfe der automatischen Codeanalyse gezeigt werden, dass *PrivateTaxiService* die Route des Nutzers nicht erfährt. In einer von solchen

Versionen veröffentlicht der Mitfahrer seine mit dem öffentlichen Schlüssel des *CalcDistanceService* verschlüsselte Start- und Zielposition bei dem *PrivateTaxiService*, aus der eine Auswahl von passenden Fahrern mit Hilfe von *CalcDistanceService* berechnet wird. Dies demonstriert die erfolgreiche Anwendung der Abstraktionen von kryptographischen Operationen (siehe Unterunterabschnitt 17.2.2.3 für Details): der Empfänger des Geheimtextes (*PrivateTaxiService*) wird nicht vom Klartext (geheime Route des Nutzers) beeinflusst, da er den für die Entschlüsselung benötigten geheimen Schlüssel nicht kennt.

Aufgrund des Umfangs des resultierenden Programmabhängigkeitsgraphen, der aus dem generierten Codeskelett erstellt wurde (~18,000 Knoten und ~50,000 Kanten; vgl. PDG von *ContactSMSManager* mit ~1,200 Knoten und ~2,200 Kanten) konnte die Anwendung trotz der eingesetzten Abstraktionen nicht vollständig mit JOANA geprüft werden, ohne dass falsch positive Ergebnisse gemeldet wurden.

Die Größe des Codeskeletts der PrivateTaxi-Anwendung zusammen mit der Abstraktion der IFlow-Bibliothek beträgt etwa 16,000 Zeilen Code

23.3.2 *Deklassifikationsmethode* isGoodMatch

Die Fallstudie nutzt die Operation *isGoodMatch(riderRoute : Route, driverRoute : Route) : Boolean* der Komponente *CalcDistanceService*, die zwei Routen als Eingabeparameter entgegennimmt, und einen booleschen Wert zurückgibt. Sie ist im Klassendiagramm mit dem Stereotyp «`declassify`» als eine Deklassifikationsmethode markiert. Die Methode *isGoodMatch* ist mit MEL* spezifiziert; sie berechnet den Abstand zwischen den Start- und Zielpositionen zweier Routen (jeweils kodiert als geographische Länge und Breite, die durch den vordefinierten MODELFLOW-Datentyp *GPSLocation* repräsentiert werden), und gibt **true** zurück, falls beide Abstände 1000m unterschreiten.

Das in Unterunterabschnitt 15.2.2.2 vorgestellte Framework erlaubt es, die Eigenschaft zu formalisieren und zu beweisen, dass ein öffentlicher Beobachter, der die Ausgabe der Methode lesen kann, nicht ohne Weiteres auf die Eingabe schließen kann. Im Folgenden wird diese Formalisierung vorgestellt und diskutiert.

DEKLASSIFIKATIONSEIGENSCHAFT Es liegt nahe, dieselbe Eigenschaft für die Methode *isGoodMatch* zu wählen, die bereits in Kapitel 21 für die Methode *calcDist* vorgestellt wurde: sie besagt, dass ein öffentlicher Beobachter nicht auf die Eingabe der Methode (Liste von Positionen) schließen kann, wenn er nur ihre Ausgabe (Distanz zwischen diesen Positionen) erfährt. Auch *isGoodMatch* erhält eine Liste von Positionen (und zwar genau vier), und gibt ein Ergebnis zurück, das von den Distanzen zwischen den Punkten der übergebenen Routen abgeleitet wird. Ihre Implementierung nutzt dabei *calcDist* der Fallstudie *DistanceTracker* (vgl. Kapitel 21); es liegt also nahe, dass die Eigenschaft trivial erfüllt ist.

Im Gegensatz zur Distance Tracker-Fallstudie ist der öffentliche Beobachter in der Private Taxi-Fallstudie (repräsentiert z.B. durch die Komponente *PrivateTaxiService*) jedoch zusätzlich in der Lage, die Eingabe für die Methode *isGoodMatch* ***selbst zu wählen***. So kann *PrivateTaxiService* beispielsweise die Route eines legitimen Nutzers als das erste Eingabeparameter *riderRoute* nutzen, welche ihr nur in verschlüsselter Form vorliegt. Um die Route dieses Nutzers im Klartext zu erfahren, kann der Angreifer versuchen, die Komponente *CalcDistService* wiederholt aufzurufen, und dabei den zweiten Parameter so zu wählen, so dass ihre Methode *isGoodMatch* **true** zurückgibt. Die Sicherheit einer Deklassifikationsfunktion hängt somit nicht nur von ihrer Spezifikation ab, sondern auch vom Kontext, in dem sie ausgeführt wird.

Jeder ist in der Lage, eine beliebige Route mit dem öffentlichen Schlüssel von CalcDistService *zu verschlüsseln.*

Die Methode *isGoodMatch* kann somit mit einer Passwortcheck-Methode verglichen werden, das in der Literatur ein Standard-Beispiel für eine Deklassifikationfunktion darstellt (vgl. Abschnitt 4.1 und [93]), da der durch sie mögliche Informationsfluss vernachlässigbar, und sie für die Funktionalität vieler Anwendung notwendig ist. Ähnlich wie bei der Überprüfung einer vom öffentlichen Beobachter eingegebenen Zeichenfolge mit einem geheimen, in der Datenbank gespeicherten Passwort, wird bei *isGoodMatch* die geheime Benutzerroute *riderRoute* mit einer öffentlichen Eingabe *driverRoute* des Angreifers verglichen. Gibt *isGoodMatch* den Wert *falsch* zurück, so erfährt der öffentliche Beobachter nichts über die Position des Nutzers außer den Bereichen auf der Erde innerhalb eines 1000m Radius um die Punkte *driverRoute.loc* und *driverRoute.dest*, in denen der Benutzer sich **nicht** befindet bzw. befinden wird, so wie bei dem Passwort-Checker der öffentliche Beobachter nichts über das geheime Passwort eines Nutzers erfährt außer dem, was es **nicht** ist [93].

Abbildung 58 zeigt die formalisierte Eigenschaft, dass es mindestens 10.000 Eingaben für die Methode *isGoodMatch* gibt, die sich nur bei einem von beiden Eingabeparametern unterscheiden, so dass die Ausgabe der Methode **false** ist. Das bedeutet, dass es für die Anwendungskomponente *PrivateTaxiService* aufwendig ist, die Position eines konkreten Nutzers (*in.riderRoute* in Abbildung 58) in Erfahrung zu bringen, selbst wenn sie diese methodisch mit selbst gewählten Positionen (*in.driverRoute* in Abbildung 58) vergleichen lässt.

Um auszudrücken, dass die intuitive Eigenschaft auch dann erfüllt ist, wenn der öffentliche Beobachter die Eingaben *riderRoute* und *driverRoute* vertauscht, müssen sie auch in den Prädikaten **property** und **enoughDifferences** vertauscht werden.

Jedoch ist ortsbezogene Privatsphäre ein komplexes Gebiet und Gegenstand aktueller Forschung (vgl. u.a. [73, 97]). Es ist möglich, dass durch abgefangene Metadaten, die Historie des Nutzerverhaltens, die spezifischen geographischen Gegebenheiten, oder Triangula-

tion [43] die Position des Nutzers weiter bzw. schneller eingeschränkt werden kann. Dies würde komplexere und anwendungsspezifischere Analyse- und Verifikationsmethoden erfordern, was über die Ziele des IFlow-Ansatzes hinausgeht. Ähnliche Einschränkungen gelten jedoch auch für das Standardbeispiel der Deklassifikation durch eine Passwortcheck-Methode: der Wertebereich der Benutzerpasswörter ist in der Realität stark beschränkt, und kann durch zusätzliche Informationen weiter verkleinert werden.

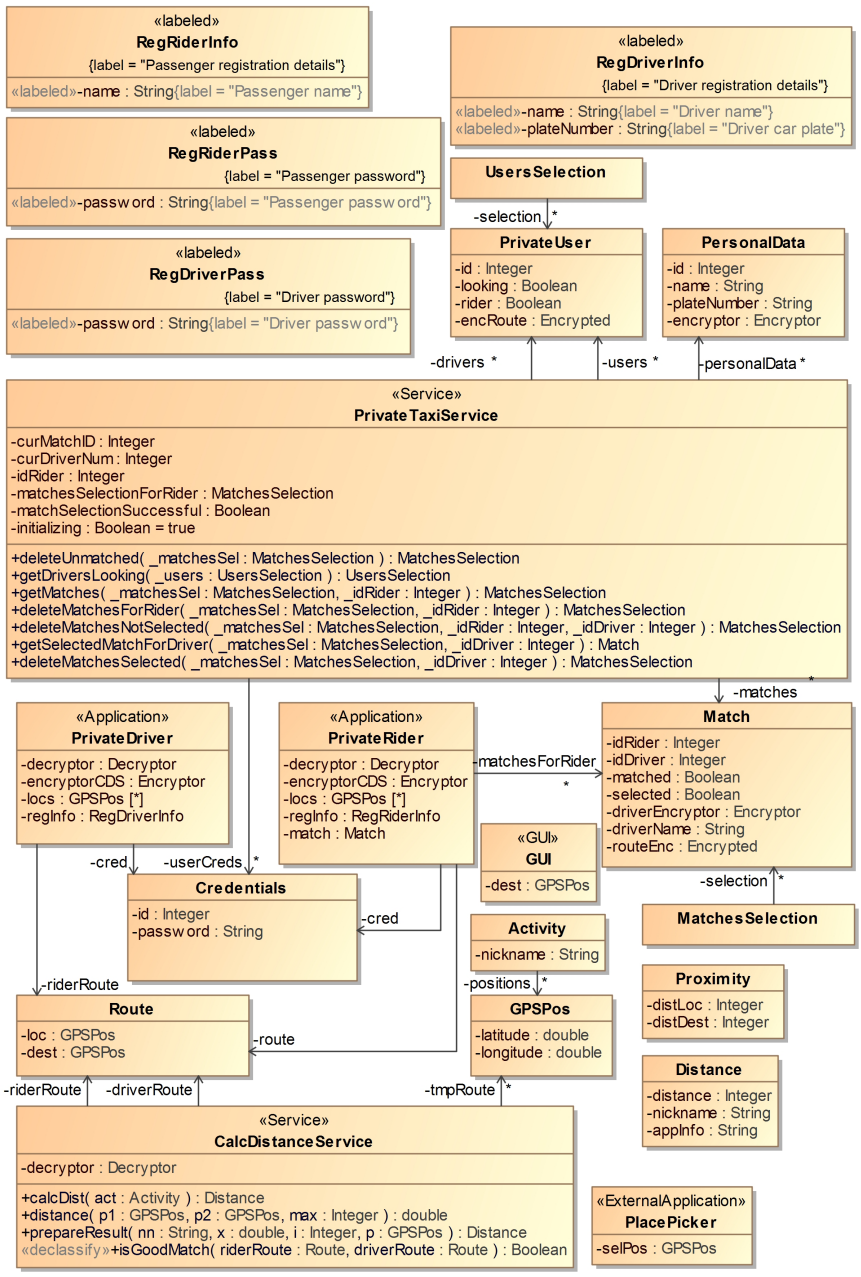


Abbildung 54: Komponenten und Datentypen der PrivateTaxi-Fallstudie

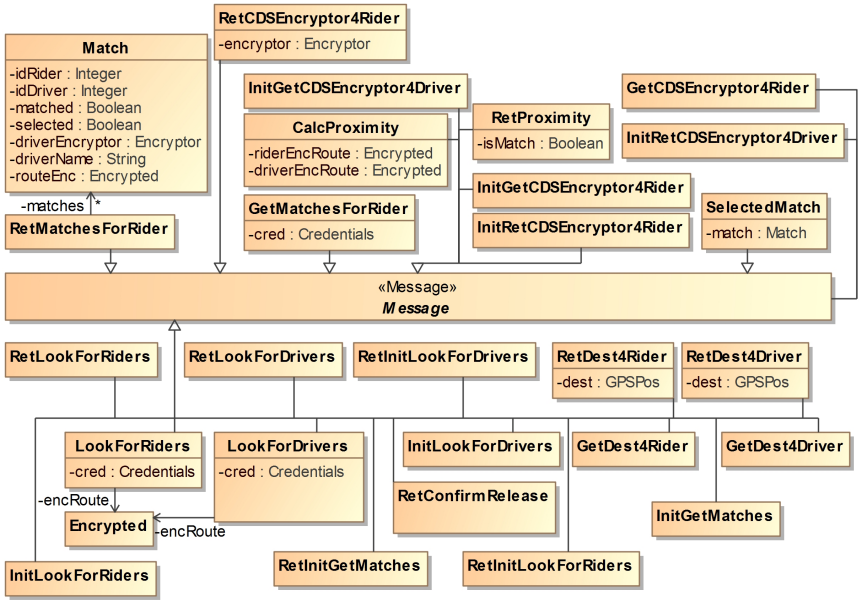


Abbildung 55: Nachrichtendatentypen und Datentypen der PrivateTaxi-Fallstudie

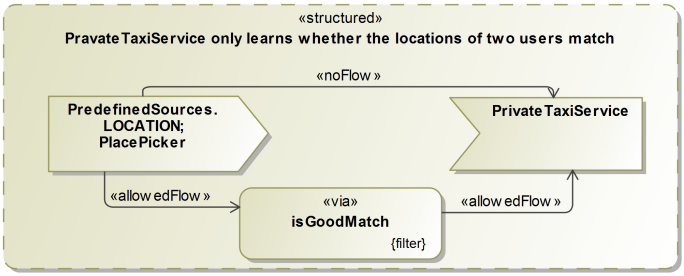


Abbildung 56: Informationsflusseigenschaft der PrivateTaxi-Fallstudie: PrivateTaxiService erfährt über die Ortsdaten der Nutzer nur, ob die Nutzer einander zugeordnet werden können

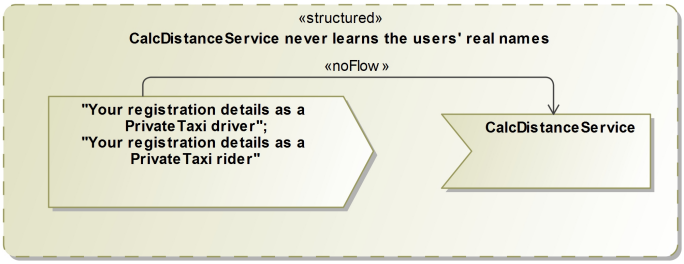


Abbildung 57: Informationsflusseigenschaft der PrivateTaxi-Fallstudie: CalcDistanceService erfährt nicht die Namen der Nutzer

$\forall$ in, out. **ff**(in) = out $\wedge$ **assumption** (in, out) $\rightarrow$
$\exists$ set. size(set) $>$ 10.000 $\wedge$
($\forall$ a. a $\in$ set $\rightarrow$ **ff**(a) = out $\wedge$ **property(a)** $\wedge$
($\forall$ a, b. a $\in$ set $\wedge$ b $\in$ set $\wedge$ a $\neq$ b $\rightarrow$ **enoughDifferences**(a, b)))

assumption(in, out) $\equiv$ out = false
property(a) $\equiv$ a.riderRoute = in.riderRoute $\wedge$
„all GPS positions in the input are valid degrees on earth“
enoughDifferences(a, b) $\equiv$ a.driverRoute.loc $\neq$ b.driverRoute.loc $\wedge$
a.driverRoute.dest $\neq$ b.driverRoute.dest

Abbildung 58: Eigenschaft der *isGoodMatch*-Methode

Teil VI

KONKLUSION

24 ZUSAMMENFASSUNG

Es ist nicht einfach, die Vertraulichkeit von sensitiven Nutzerdaten in komplexen, verteilten Anwendungen zu garantieren. Wie die vorliegende Arbeit zeigt, hilft es, die Betrachtung solcher Eigenschaften in die Systementwicklungsmethodik zu integrieren.

Hierzu wurde eine neue Modellierungssprache MODELFLOW entworfen, mit der die Struktur, das Verhalten, und die Informationsflusseigenschaften eines sicherheitskritischen Systems bestehend aus mobilen Apps und Webservices graphisch spezifiziert werden können. Besonderer Wert wurde dabei auf Relevanz und Verständlichkeit solcher Eigenschaften, sowie die sichere und benutzerfreundliche Interaktion mit dem Endnutzer gelegt, denn nur so kann die korrekte Verwendung des Systems gewährleistet werden [79, 100].

Die formale Semantik von MODELFLOW ist durch das formale Modell gegeben, das aus dem Anwendungsmodell generiert wird, und ein formales Framework zur Informationsflusskontrolle instantiiert. Die für das formale Modell durch interaktive Verifikation garantierten Eigenschaften vererben sich dabei auf den lauffähigen Code der Anwendung.

Ein wichtiger Aspekt des vorgestellten Ansatzes ist die Möglichkeit, Techniken zur vollautomatischen Informationsflusskontrolle einzusetzen. So muss der Entwickler keine Kenntnisse in formalen Methoden und interaktiver Verifikation vorweisen, um eine Vielzahl der unterstützten Informationsflusseigenschaften garantieren zu können. Der aus dem Modell generierte Code lässt sich mit einem aktuellen Informationsflussanalysetool vollautomatisch auf die spezifizierten Informationsflusseigenschaften hin analysieren, wofür eine Reihe von Codeabstraktionstechniken entwickelt werden musste. Dieser Code kann händisch erweitert werden, ohne dass die garantierten Eigenschaften verletzt werden, und als Android Apps und Java-Webservices auf echter Hardware installiert werden.

Das Ergebnis dieser Arbeit ist der modellgetriebene Ansatz *IFlow* zur Entwicklung informationsflusssicherer Systeme. Der Ansatz ist toolunterstützt, und wurde anhand mehrerer Fallstudien evaluiert. Durch dessen Einsatz lassen sich gängige Datenlecks wie etwa die Freigabe persönlicher Daten an nicht autorisierte Nutzer vermeiden, die in der Vergangenheit bei einer Vielzahl von Internet-Dienstleistern wie Uber [74], Target [14], und Google [10] festgestellt wurden.

25 AUSBLICK

Der vorgestellte Ansatz ist keineswegs die Lösung, sondern ein weiterer Schritt im Kampf um die Privatsphäre im Internet. Um mit den sich ständig weiterentwickelnden Paradigmen und Mechanismen der nativen Plattformen, Entwicklungssprachen und -tools für mobile Geräte und Webservices Schritt zu halten, muss auch die Entwicklungsmethodik ausgebaut und aktualisiert werden. Des weiteren ist die Entwicklung einer Infrastruktur für die Verwaltung und Verbreitung sicherer Anwendungen notwendig [59]. Auch müssen bereits vorhandene Formalismen und Tools zur Informationsflusskontrolle weiter verbessert werden, um ihre aktuellen Einschränkungen (wie etwa bzgl. der Komplexität und Flexibilität der zu analysierenden Systeme [34]) zu beseitigen. Zuletzt müssen neben der Vertraulichkeit und den Umständen der Freigabe geheimer Information auch die potenziellen Folgen solcher Freigabe für den Endnutzer betrachtet und intuitiv vermittelt werden können. Dies erscheint notwendig, um der apathischen Einstellung vieler Nutzer gegenüber der Verletzung ihrer Privatsphäre entgegenzuwirken.

In der Zukunft werden das kommende „Internet der Dinge“ (*Internet of Things, IoT*), das die Zahl an verbundenen Geräten bis 2020 auf 30 Milliarden ansteigen lassen soll [70], sowie neuartige Maschinenlernalgorithmen [87] und dezentrale Konsensmechanismen [60] neue Herausforderungen mit sich bringen. Solche Systeme benötigen neue Lösungsansätze, um die Vertraulichkeit von Benutzerdaten zu garantieren. So sind aktuell verbundene IoT-Geräte oft auf zentrale Server angewiesen, die die von den Geräten gesammelte Nutzerdaten zentral und unverschlüsselt verwalten. Maschinenlernalgorithmen, die z.B. Gesundheitsprobleme analysieren sollen, benötigen ihrerseits große Mengen an Rohdaten, die sensitive Krankheitsinformationen von Patienten enthalten und geschützt werden müssen [16]. Bei den meisten der aktuellen Konsensalgorithmen (bis auf wenige Ausnahmen wie [54, 82]), die die Implementierung sicherer Anwendungen ohne einer zentralen Autorität ermöglichen, sind alle verwalteten Nutzerdaten für alle Systemteilnehmer einsehbar.

Zuletzt soll erwähnt werden, dass die Privatsphäre der Nutzer im Internet oft nicht aufgrund von Fehlimplementierungen, sondern aus wirtschaftlichen Gründen verletzt wird. So sind heute viele Künstler, Software-Entwickler, und Inhaltsanbieter auf Einnahmen durch Werbenetzwerke angewiesen, an die sie die privaten Daten ihrer Nutzer verkaufen, um diesen zielgerichtete Werbung anzeigen zu können. So sind heute viele Künstler, Software-Entwickler, und Inhalts-

anbieter auf Einnahmen durch Werbenetzwerke angewiesen, an die sie die privaten Daten ihrer Nutzer verkaufen, um diesen zielgerichtete Werbung anzeigen zu können. Um die Privatsphäre der Nutzer zu schützen, müssen neben Werkzeugen, die die Entwickler bei der Implementierung informationsflusssicherer Anwendungen unterstützen, alternative Einnahmequellen und Plattformen geschaffen werden, die dieses Problem lösen.

Teil VII

APPENDIX

A IFLOW FRAMEWORK

A.1 INFORMATIONSFLUSSSIMULATION ZWISCHEN VARIABLEN

Listing 21: Ausschnitt aus dem Quellcode der `IFUtility`-Klasse zur Simulation einer Informationsflussabhängigkeit zwischen beliebigen Variablen im generierten Codeskelett

```
1  public final class IFUtility {
2          ⋮
3          /**
4           * Extracts the highest security level present among the
                given objects by deeply inspecting every single
5           * object.
6           *
7           * <p>
8           * Note that this method cannot be used for the following
                objects:
9           * <ul>
10          * <li>Primitive array types. In this case use the
                dedicated methods for primitive array types</li>
11          * <li>Collections. In this case use the dedicated method
                for collections.</li>
12          * </ul>
13          *
14          * <p>
15          * Note that in order to extract the security level from
                arbitrary arrays of type {@code Object} it
16          * must be passed as the only argument casted to {@code
                Object[]}. It is possible to pass it together
17          * with other objects, but then the method will return
                the wrong result in certain situations.
18          *
19          * @param objects The objects to extract the highest
                security level from.
20          *
21          * @return An integer value tagged with the highest
                present security level among the objects extracted
22          * by deeply accessing all the given objects.
23          */
24         public static int extractSecurityLevel(Object...objects)
               {
25                 int securityLevel = 0;
26
27                 for (Object object : objects) {
```

```
28                      if (object instanceof
                            SecurityLevelExtractor) {
29                          securityLevel += ((
                                SecurityLevelExtractor)
                                object).extractSecurityLevel
                                ();
30                      } else {
31                          securityLevel += (object != null)
                                ? 1 : 0;
32                      }
33              }
34
35              return securityLevel;
36      }
37      /**
38       * Returns an instance of the desired return type tagged
             with the given security level.
39       *
40       * <p>
41       * For generic types use {@link #getTaggedType(
             TypeReference, int)}.
42       *
43       * @param type The desired return type.
44       * @param securityLevel An integer value tagged with the
             desired security level for the type instance
45       * to return.
46       *
47       * @return An instance of the desired return type tagged
             with the given security level.
48       */
49      public static <T> T getTaggedType(Class<T> type, int
            securityLevel) {
50              T taggedType = null;
51              try {
52                      taggedType = IFUtility.initType(type);
53
54                      if (securityLevel > 0) {
55                          taggedType = IFUtility.initType(
                                type);
56                      }
57              } catch (InstantiationException e) {
58                      // Do nothing since JOANA does not care
                            in case exception handling is turned
                            off.
59              } catch (IllegalAccessException e) {
60                      // Do nothing since JOANA does not care
                            in case exception handling is turned
                            off.
61              }
62
63              return taggedType;
64      }
```

```
65          ⋮
66  }
```

A.2 VORDEFINIERTE BENUTZEROBERFLÄCHE

Listing 22: Quellcode von `showGetSingleSelectionFragment` zum Anzeigen einer Listenauswahl

```
1  class IFlowUser {
2  ⋮
3          /**
4           * Shows a selection fragment in the main layout of the
                given IFlow application, i.e. all
5           * the contents in the main layout are replaced with the
                selection fragment.
6           *
7           * @param iFlowApplication The IFlow application in which
                 to show the selection fragment.
8           * @param titleIDName identifier name of title text
9           * @param items List of items to display as the
                selectable options in the fragment.
10          * @param singleSelectionListener The selection listener
                which is called when the user selects an
11          * item out of the list of items.
12          * @param addToBackStack Flag indicating if the returned
                fragment should be put on the application
13          * task's back stack or not.
14          *
15          * @return The IFlow selection fragment which is created
                and displayed.
16          */
17          protected <T extends IFlowData>
                GetSingleSelectionFragment<T>
                showGetSingleSelectionFragment(IFlowApplication
                iFlowApplication,
18                              String titleIDName, List<T> items,
                                    SingleSelectionListener<T>
                                    singleSelectionListener, boolean
                                    addToBackStack) {
19                  GetSingleSelectionFragment<T> fragment =
                        GetSingleSelectionFragment.newInstance(
                        iFlowApplication, titleIDName,
20                                      items, singleSelectionListener);
21                  FragmentTransaction transaction =
                        iFlowApplication.getFragmentManager().
                        beginTransaction();
22                  transaction.replace(R.id.a_main_container,
                        fragment);
23
24                  if (addToBackStack) {
```

```
25                          transaction.addToBackStack(null);
26                  }
27
28                  transaction.commit();
29
30                  return fragment;
31          }
32  ⋮
33  }
```

Listing 23: Quellcode von `GetSingleSelectionFragment` zum Anzeigen einer Listenauswahl

```
1  /**
2   * Fragment for receiving a user selected item out of a list of
        items.
3   *
4   *
5   * @param <T> The type of IFlow data.
6   */
7  public class GetSingleSelectionFragment<T extends IFlowData>
       extends ListFragment {
8
9          private String title;
10
11         /**
12          * The selection listener.
13          */
14         private SingleSelectionListener<T>
               singleSelectionListener;
15
16         /**
17          * Static factory method for creating a new IFlow single
                selection fragment.
18          *
19          * <p>
20          * It is best practice in Android to avoid a non-default
                constructor for fragments and instead
21          * use a static factory method.
22          *
23          * @param iFlowApplication The IFlow application to
                display the selection fragment in.
24          * @param items List of items to display as selection
                options for the user.
25          * @param singleSelectionListener The listener to call
                when the user selects an item.
26          *
27          * @return The newly created IFlow single selection
                fragment configured with the given
28          * parameters.
29          */
```

```
30          public static <T extends IFlowData>
                GetSingleSelectionFragment<T> newInstance(
                IFlowApplication iFlowApplication,
31                          String titleIDName, List<T> items,
                                SingleSelectionListener<T>
                                singleSelectionListener) {
32                  GetSingleSelectionFragment<T> fragment = new
                        GetSingleSelectionFragment<T>();
33                  fragment.setDataItems(items, iFlowApplication);
34                  fragment.setSingleSelectionListener(
                        singleSelectionListener);
35
36                  int titleID = iFlowApplication.getResources().
                        getIdentifier(titleIDName, "string",
                        iFlowApplication.getPackageName());
37                  fragment.setTitle(iFlowApplication.getString(
                        titleID));
38
39                  return fragment;
40          }
41
42          /**
43           * Sets fragment title
44           * @param title Title of the fragment
45           */
46          private void setTitle(String title) {
47                  this.title = title;
48          }
49
50          /**
51           * Adds title header to the fragment
52           */
53      public void onActivityCreated(Bundle savedInstanceState) {
54          super.onActivityCreated(savedInstanceState);
55
56          TextView ret = new TextView(getActivity());
57
58          ret.setText(title);
59          ret.setTextSize(TypedValue.COMPLEX_UNIT_SP,18);
60          ret.setGravity(Gravity.CENTER);
61
62          this.getListView().addHeaderView(ret);
63      }
64
65          /**
66           * Set the selection listener.
67           *
68           * @param singleSelectionListener The selection listener
                    to set.
69           */
70          public void setSingleSelectionListener(
                SingleSelectionListener<T> singleSelectionListener) {
```

```
71                  this.singleSelectionListener =
                        singleSelectionListener;
72          }
73
74          /**
75           * Sets the list of data items to present for selection.
76           *
77           * @param items The list of data items.
78           * @param iFlowApplication The IFlow application to
                 display the data items in.
79           */
80          public void setDataItems(List<T> items, IFlowApplication
                iFlowApplication) {
81                  IFlowArrayAdapter arrayAdapter = new
                        IFlowArrayAdapter(iFlowApplication, items);
82                  this.setListAdapter(arrayAdapter);
83          }
84
85          @Override
86          public void onListItemClick(ListView l, View v, int
                position, long id) {
87                  super.onListItemClick(l, v, position, id);
88
89                  @SuppressWarnings("unchecked")
90                  T selectedItem = (T) l.getItemAtPosition(position
                        );
91
92                  if (this.singleSelectionListener != null) {
93                          this.singleSelectionListener.
                                onSingleSelection(selectedItem);
94                  }
95          }
96  }
```

Listing 24: Quellcode von `IFlowApplication` zum Propagieren eines Benutzerabbruchs

```
1   /**
2    * Represents an IFlow application.
3    *
4    */
5   public abstract class IFlowApplication extends Activity {
6   ⋮
7           @Override
8           public void onActivityResult(int requestCode, int
                resultCode, Intent data) {
9           ⋮
10                  if (data != null && dataProvider.getBooleanExtra(
                        CANCEL_NOTICE, false)) {
11                          this.userInputCancelled();
12                          return;
```

```
13                  }
14          ⋮
15          }
16
17          /**
18           * Callback that is used to handle an input that should
                 have been done by the user
19           * but that was cancelled.
20           */
21          public void userInputCancelled() {
22                  if (this.getCallingActivity() != null) {
23                          IntentBuilder cancelNotice =
                                IntentBuilder.create().putExtra(
                                CANCEL_NOTICE, true);
24                          this.setResult(RESULT_CANCELED,
                                IntentUtility.buildIntent(
                                cancelNotice));
25                          this.finish();
26                  } else {
27                          this.init();
28                  }
29          }
30
31  ⋮
32  }
```

A.3 KOMMUNIKATION ZWISCHEN IFLOW-KOMPONENTEN

Listing 25: Quellcode der `executeRequestForResult`-Methode zur Kommunikation mit einer externen Android-App

```
1   public abstract class ExternalApplicationGSL {
2   ⋮
3           /**
4            * Starts an external application with the given intent
                 in order to retrieve a result.
5            *
6            * <p>
7            * If there is an application available to handle the
                 given intent, is used to start the external
8            * application. Otherwise the user is informed there
                 exists no application capable of handling the
9            * configured intent.
10           *
11           * <p>
12           * Note that the usage of this method only makes sense
                 for intent protocols returning a result.
13           *
```

```
14          * @param externalApplication The external application to
                start in order to obtain a result.
15          * @param iFlowApplication The IFlow application to start
                the external application from.
16          * @param intentBuilder The configured intent builder.
17          * @param resultFunction The result function that
                contains the program flow to resume after having
18          * received the desired result.
19          */
20         protected static final void executeRequestForResult(
               ExternalApplication externalApplication,
               IFlowApplication iFlowApplication,
21                         IntentBuilder intentBuilder,
                               ResultFunction resultFunction) {
22                 if (IntentUtility.isIntentResolvable(
                       iFlowApplication, intentBuilder)) {
23                         iFlowApplication.setResultFunction(
                               resultFunction);
24
25                         iFlowApplication.startActivityForResult(
                               IntentUtility.buildIntent(
                               intentBuilder), 0);
26                 } else {
27                         Toast.makeText(iFlowApplication, R.string
                               .external_app_unavailable_error_msg,
28                                       Toast.LENGTH_SHORT).show
                                           ();
29                 }
30         }
31 ⋮
32 }
```

Listing 26: Quellcode der `isIntentResolvable`-Methode zur Überprüfung eines Android-Intents auf Gültigkeit

```
1  public final class IntentUtility {
2  ⋮
3          /**
4           * Returns if the intent configured by the given builder
                is resolvable for the given IFlow application.
5           *
6           * @param iFlowApplication The IFlow application from
                which to resolve the built intent.
7           * @param intentBuilder The configured intent builder.
8           *
9           * @return <code>true</code> if the intent configured by
                the builder is resolvable, otherwise
10          * <code>false</code>.
11          */
12         public static boolean isIntentResolvable(IFlowApplication
                iFlowApplication, IntentBuilder intentBuilder) {
```

```
13                 PackageManager packageManager = iFlowApplication.
                       getPackageManager();
14                 List<ResolveInfo> list = packageManager.
                       queryIntentActivities(intentBuilder.intent,
15                                 PackageManager.MATCH_DEFAULT_ONLY
                                       );
16
17                 return list.size() > 0;
18         }
19 ⋮
20 }
```

Listing 27: Quellcode der Nachrichtenversandmethode zur Kommunikation zwischen IFlow-Apps

```
1  public abstract class IFlowApplicationProxy extends
       IFlowApplication {
2          /**
3           * Opens another IFlow application in order to retrieve a
                result message.
4           *
5           * @param iFlowApplication The originating IFlow
                application from which to invoke the other IFlow
6           * application.
7           * @param packageName The package name of the IFlow
                application to open.
8           * @param className The fully qualified class name of the
                IFlow application to open, i.e. including
9           * the package name.
10          * @param requestCode The request code to use for
                starting the other IFlow application.
11          * @param key The data key with which the given message
                is stored in the intent invoking the other
12          * IFlow application.
13          * @param message The IFlow message to send to the other
                IFlow application.
14          * @param resultFunction The result function that will be
                called when the invoked IFlow application
15          * finishes and returns its result.
16          */
17         protected <T extends IFlowMessage> void
               startActivityForResult(IFlowApplication
               iFlowApplication, String packageName,
18                             String className, int requestCode, String
                                    key, T message,
                                RequiredResultFunction resultFunction
                                ) {
19                 String jsonData = new Gson().toJson(message);
20                 Intent intent = new Intent();
21                 intent.setClassName(packageName, className);
22                 intent.putExtra(key, jsonData);
```

```
23                  iFlowApplication.setResultFunction(resultFunction
                        );
24                  iFlowApplication.startActivityForResult(intent,
                        requestCode);
25          }
26  }
```

A.4 VORDEFINIERTE METHODEN

Listing 28: Quellcode der `startGPSTracking`- und `stopGPSTracking`-Methoden zum Aufzeichnen und Auslesen der aufgezeichneten GPS-Positionen

```
1   public class IFlowUtil {
2   ⋮
3           /**
4            * Starts GPSTracking for the given context
5            * @param context the context to track in
6            */
7           public static void startGPSTracking(Context context) {
8                   GPSSensor sensor = GPSSensor.getInstance();
9                   sensor.clearData();
10                  sensor.setContext(context);
11                  sensor.startSensor();
12          }
13
14          /**
15           * Stops GPS tracking and returns all tracked positions
16           * @return a list of all tracked Positions
17           */
18          public static List<GPSPos> stopGPSTracking() {
19                  GPSSensor sensor = GPSSensor.getInstance();
20                  sensor.stopSensor();
21                  ArrayList<Location> locations = sensor.
                        getLocations();
22                  ArrayList<GPSPos> pos = new ArrayList<GPSPos>();
23                  for (Integer i = 0; (i < locations.size()); i++)
                        {
24                      Location location = locations.get(i);
25                      pos.add(new GPSPos(location.getLatitude()
                            , location.getLongitude()));
26                  }
27                  return pos;
28          }
29  ⋮
30  }
31
32  public class GPSSensor implements LocationListener{
33          ⋮
```

```
34          @Override
35          public void onLocationChanged(Location location) {
36                  if(location!=null){
37                          hasLocation = true;
38                          locations.add(location);
39                  }
40          }
41
42          public ArrayList<Location> getLocations() {
43                  return locations;
44          }
45
46          /**
47           * Starts the Sensor
48           */
49          public void startSensor(){
50                  hasLocation = false;
51                  locationManager = (LocationManager) mContext.
                        getSystemService(Context.LOCATION_SERVICE);
52                  long curTime = System.currentTimeMillis();
53                  for(String provider : locationManager.
                        getAllProviders()){
54                          if(locationManager.isProviderEnabled(
                                provider)){
55                                  locationManager.
                                        requestLocationUpdates(
                                        provider, 5*1000, 0, this);
56                                  Location lastknown =
                                        locationManager.
                                        getLastKnownLocation(provider
                                        );
57                                  if(lastknown != null){
58                                          long diff = curTime -
                                                lastknown.getTime();
59                                          if(diff <ACCEPT_DIFF_MS){
60                                                  onLocationChanged
                                                        (lastknown);
61                                          }
62                                  }
63                          }
64                  }
65          }
66
67          /**
68           * Stops the Sensor
69           */
70          public void stopSensor(){
71                  while(!hasLocation){
72                          try {
73                                  Thread.sleep(500);
74                          } catch (InterruptedException e) {}
75                  }
```

```
76                  if(locationManager != null){
77                          locationManager.removeUpdates(this);
78                  }
79          }
80  }
```

B

KONKRETE SYNTAX VON MEL*

Init = CallMessage | CallMessage*

ToSelfMessage = Expr | Stm*
CallMessage = Identifier (ExprList)
ReplyMessage = Identifier (ExprList)

Action = Expr | Stm*
Guard = Expr | else

Stm = Expr;
ExprList = ε | Expr[, Expr]*

Expr = Locvardecl | Assignment | CreateExpr | MethodCall
| BinaryExpr | UnaryExpr | LiteralExpr | FieldAccess
| Name | (Expr) | ForExpr
LocvarDecl = Identifier : Type | Identifier : Type := Expr
Assignment = FieldAccess := Expr | Name := Expr
CreateExpr = create Identifier (ExprList)
MethodCall = Identifier (ExprList) | Expr.Identifier (ExprList)
BinaryExpr = Expr Binop Expr
UnaryExpr = Unop Expr | Expr Unop
LiteralExpr = true | false | NumberLiteral | StringLiteral
FieldAccess = Expr.Identifier
ForExpr = for(Expr; Expr; Expr) Stm*

Name = Identifier | Name.Identifier | self
Identifier = Gültiger Java Identifier
Type = IdentType | Boolean | Number | String
Binop = == | != | < | > | <= | >= | + | - | * | / | % | and | or
Unop = + | - | not | ++ | –
NumberLiteral = Gültiges Java Integer oder Float Literal
StringLiteral = Gültiges Java String Literal
IdentType = Identifier

Abbildung 59: Konkrete Syntax von MEL*

C

MODELL DER PRIVATETAXI-FALLSTUDIE

C.1 SEQUENZDIAGRAMME

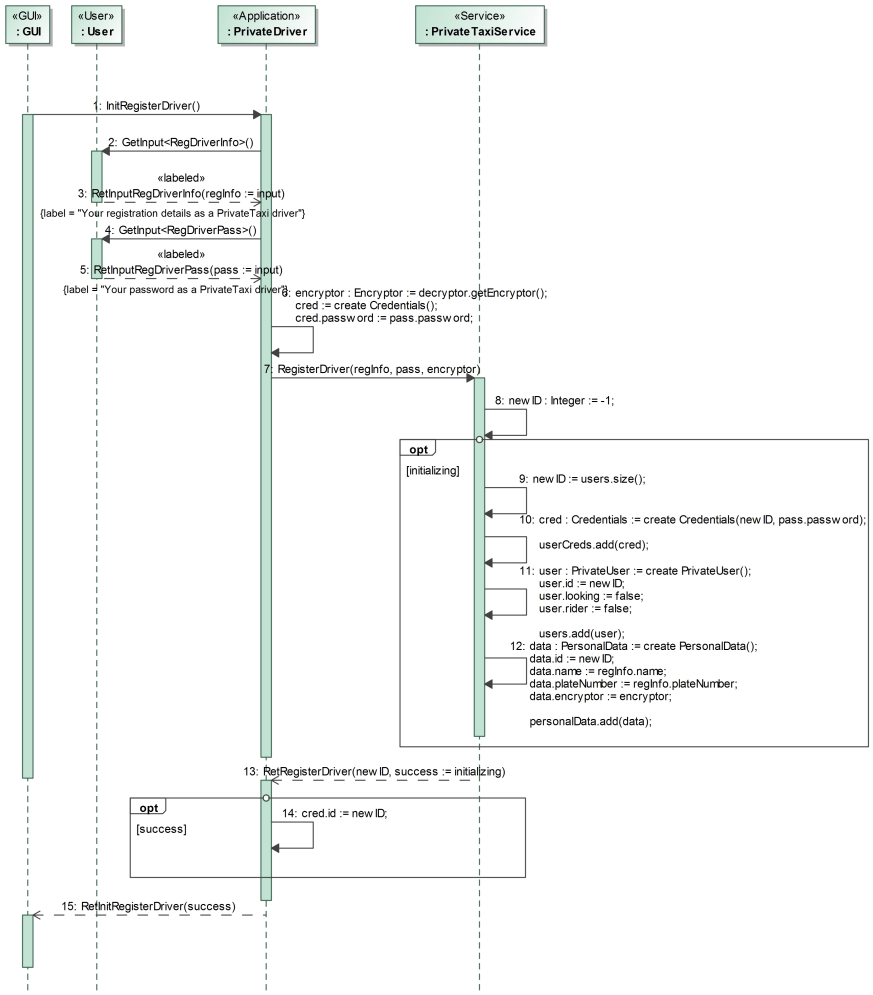


Abbildung 60: Sequenzdiagramm der PrivateTaxi-Fallstudie zur Anmeldung eines Fahrers

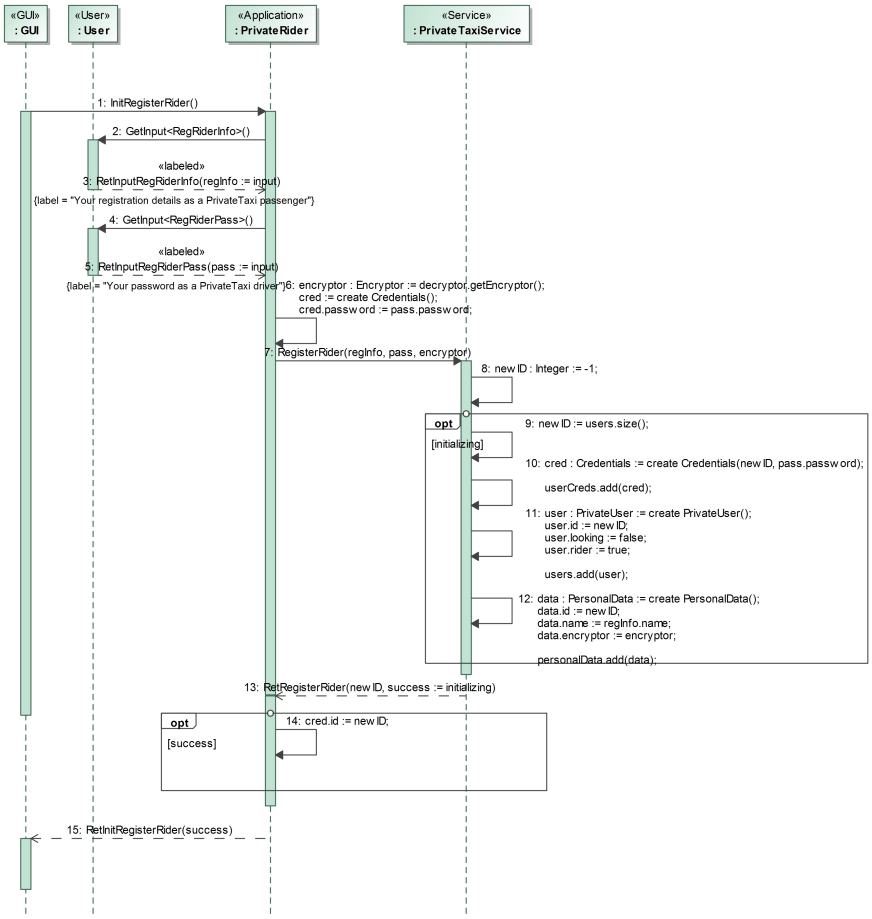


Abbildung 61: Sequenzdiagramm der PrivateTaxi-Fallstudie zur Anmeldung eines Mitfahrers

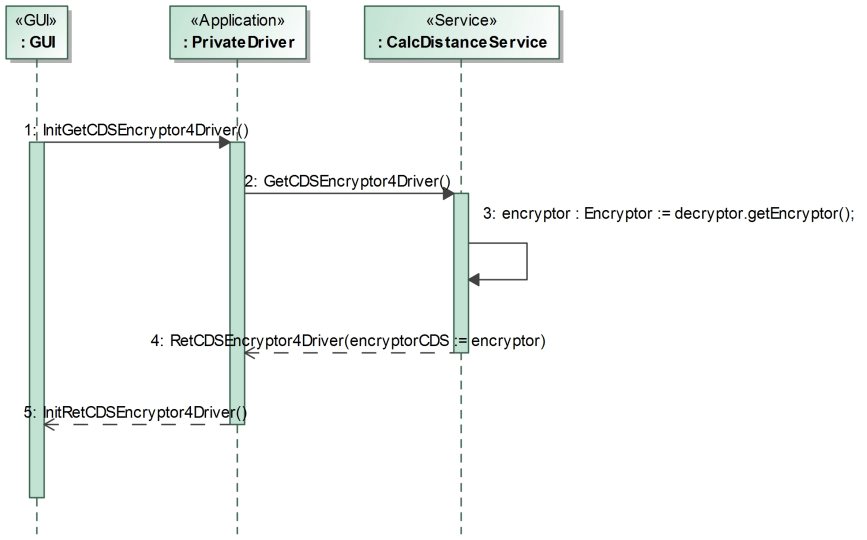


Abbildung 62: Sequenzdiagramm der PrivateTaxi-Fallstudie zum Abholen des öffentlichen Schlüssels des CalcDistanceService für den Fahrer

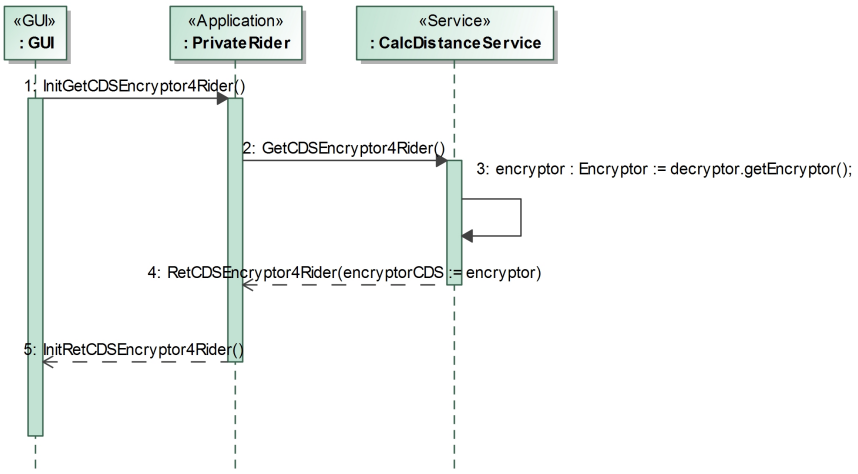


Abbildung 63: Sequenzdiagramm der PrivateTaxi-Fallstudie zum Abholen des öffentlichen Schlüssels des CalcDistanceService für den Mitfahrer

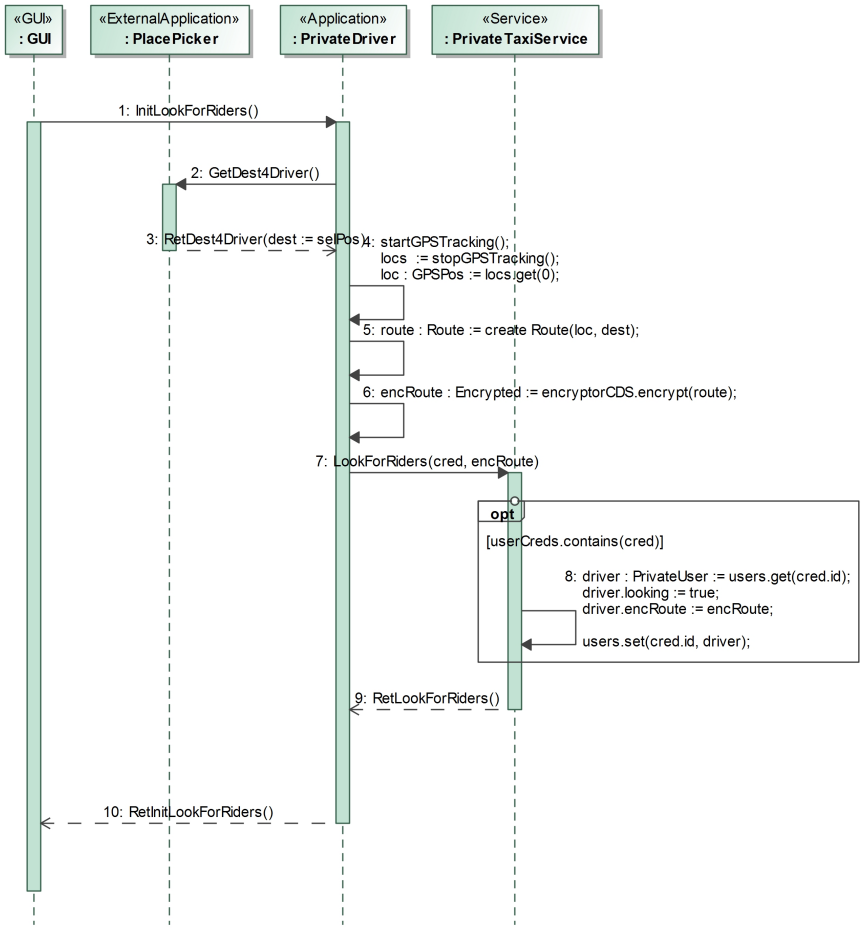


Abbildung 64: Sequenzdiagramm der PrivateTaxi-Fallstudie zum Veröffentlichen der verschlüsselten Route des Fahrers

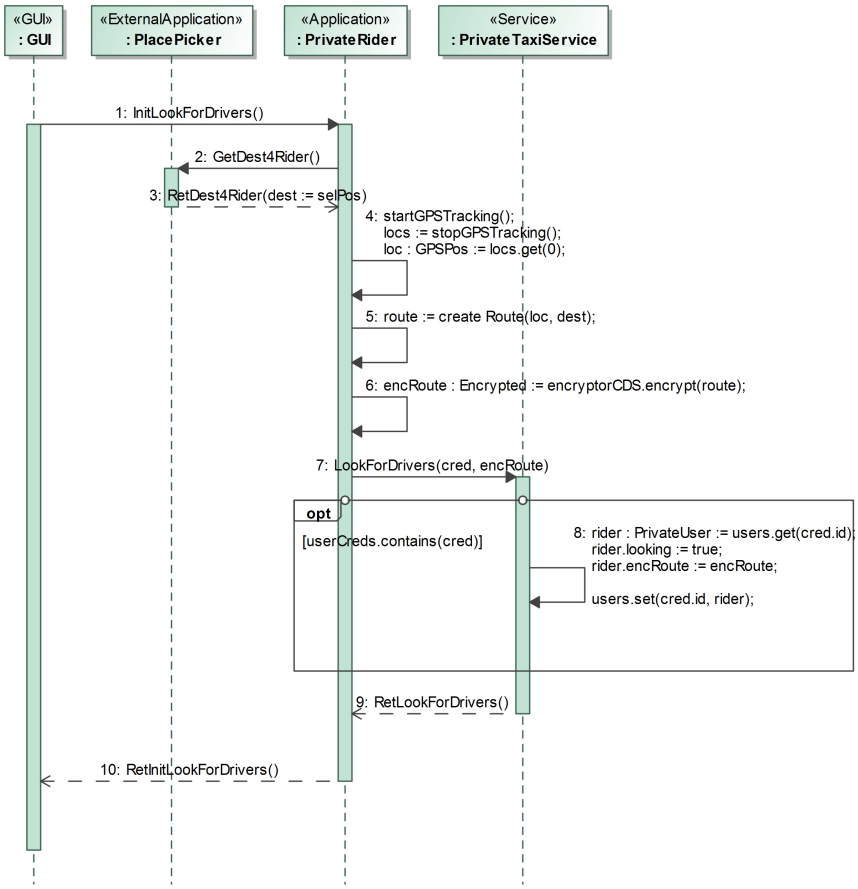


Abbildung 65: Sequenzdiagramm der PrivateTaxi-Fallstudie zum Veröffentlichen der verschlüsselten Route des Mitfahrers

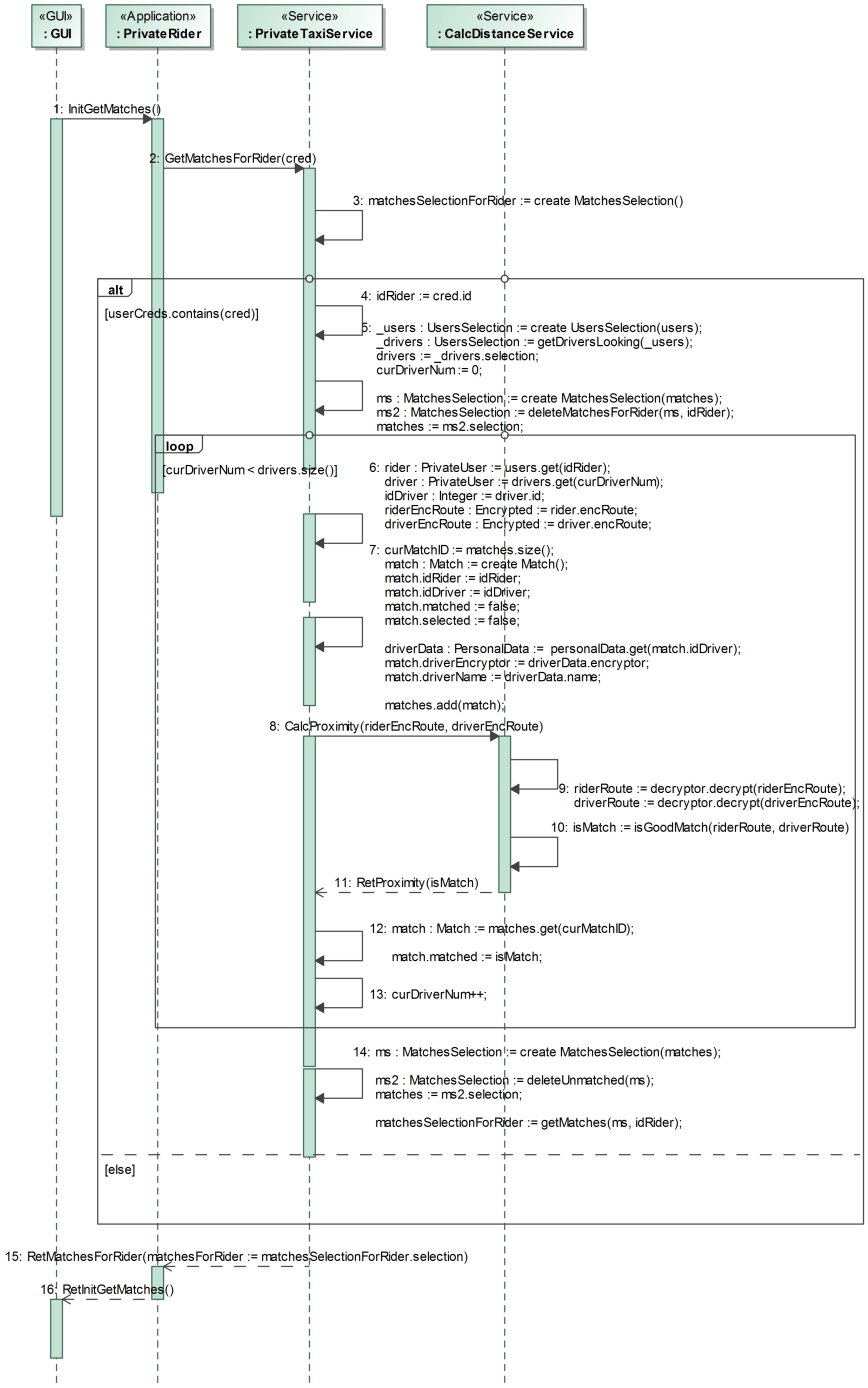


Abbildung 66: Sequenzdiagramm der PrivateTaxi-Fallstudie zum Berechnen einer Liste von passenden Fahrern für einen Mitfahrer

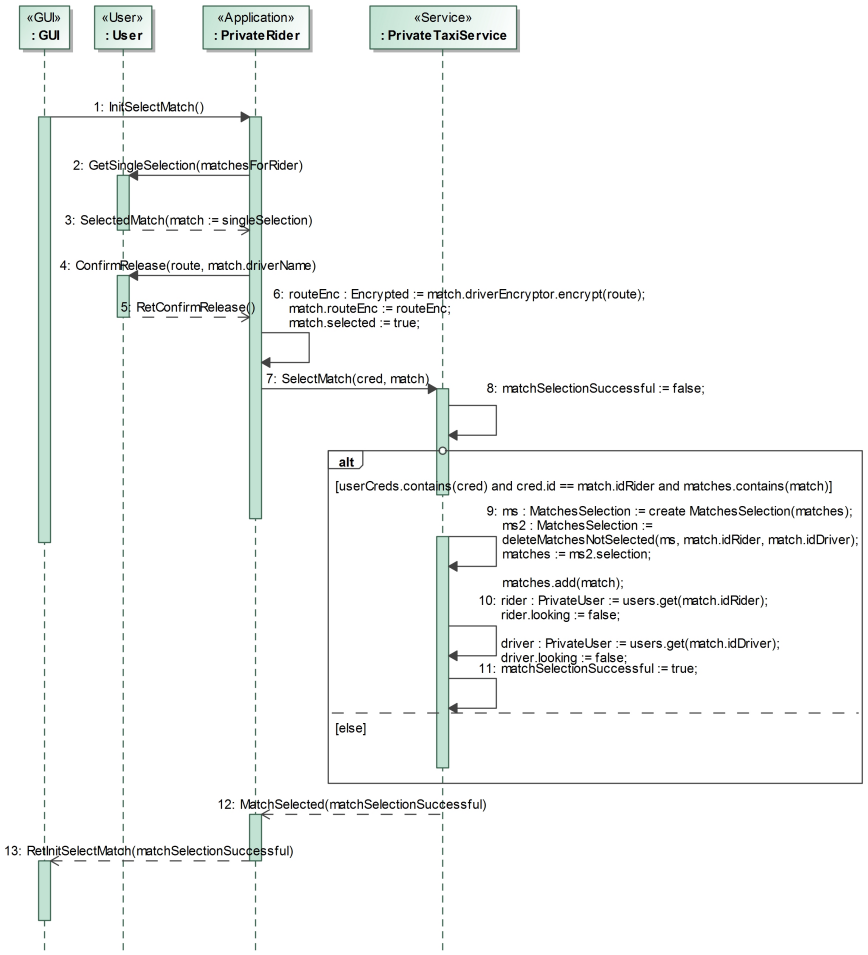


Abbildung 67: Sequenzdiagramm der PrivateTaxi-Fallstudie zur Auswahl eines vorgeschlagenen Fahrers

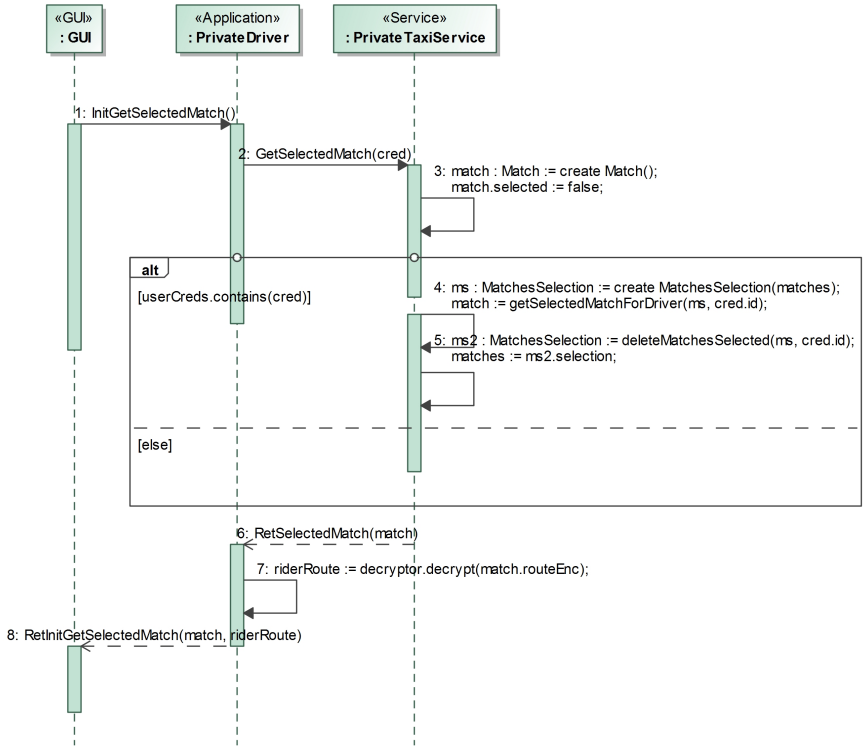


Abbildung 68: Sequenzdiagramm der PrivateTaxi-Fallstudie zur Abholung des zugeordneten Mitfahrers

C.2 FILTERMETHODEN

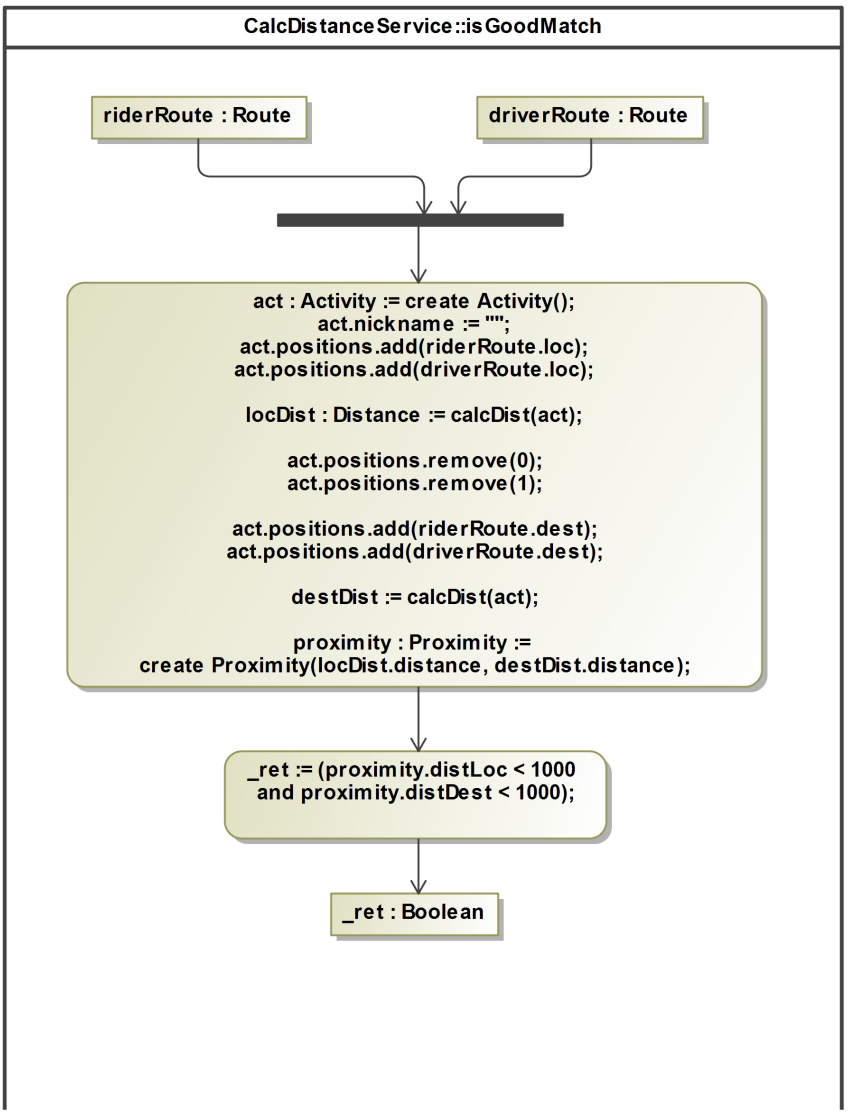


Abbildung 69: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zur Berechnung, ob die Routen zweier Nutzer kompatibel sind

C.3 HILFSMETHODEN

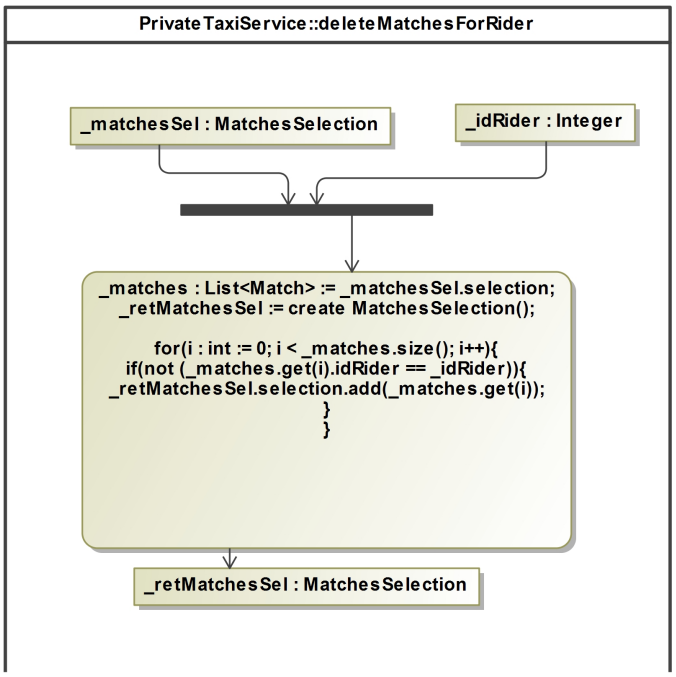


Abbildung 70: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zum Löschen aller Zuordnungen für einen Mitfahrer

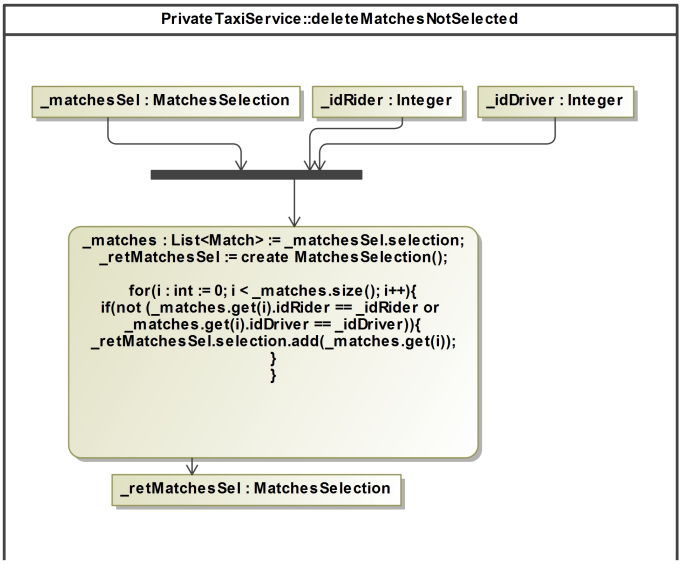


Abbildung 71: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zum Löschen aller nicht ausgewählten Zuordnungen für einen Fahrer und Mitfahrer

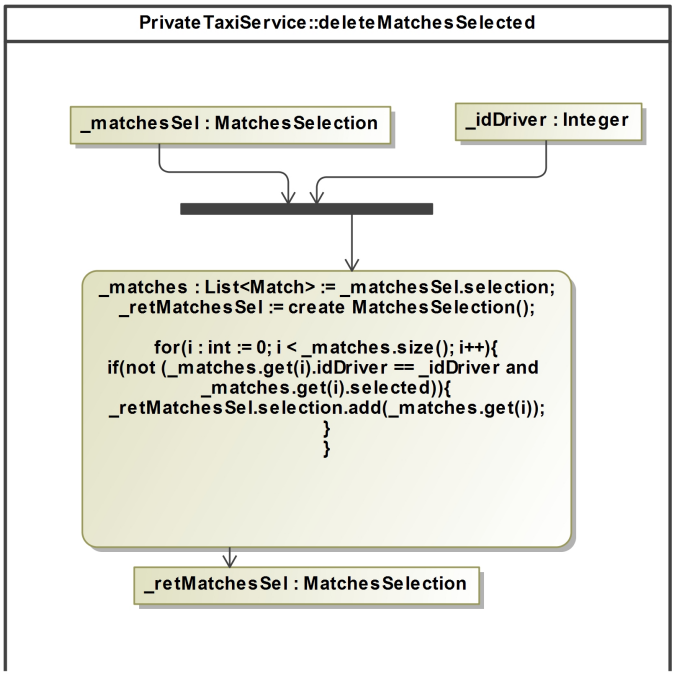


Abbildung 72: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zum Löschen aller ausgewählten Zuordnungen für einen Fahrer

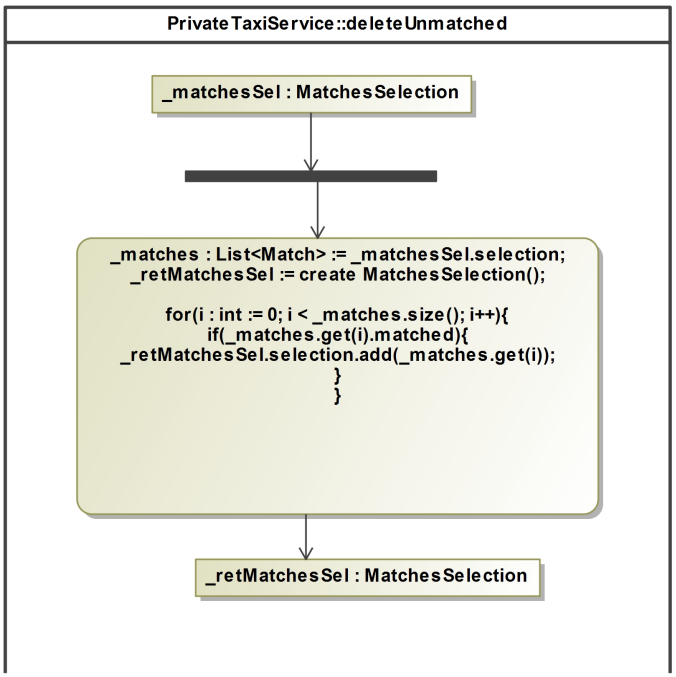


Abbildung 73: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zum Löschen aller nicht erfolgreichen Zuordnungen

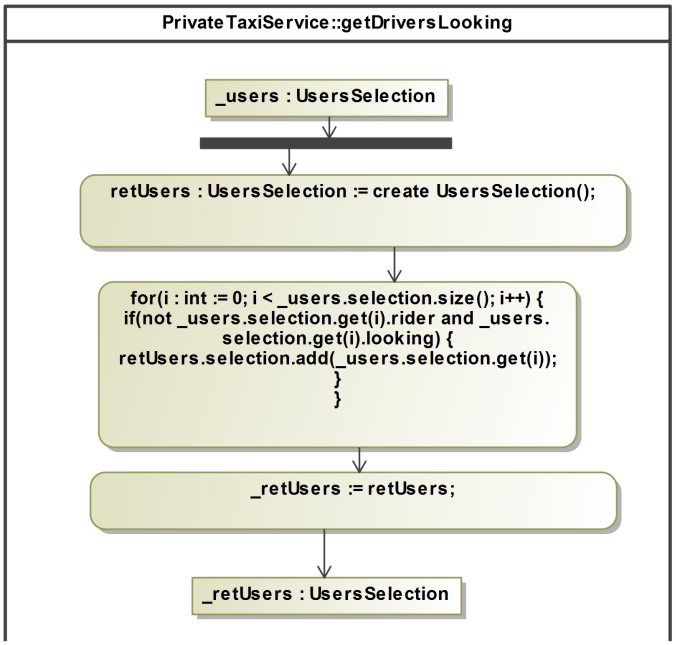


Abbildung 74: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zur Abfrage aller Fahrer, die auf der Suche nach einem Mitfahrer sind

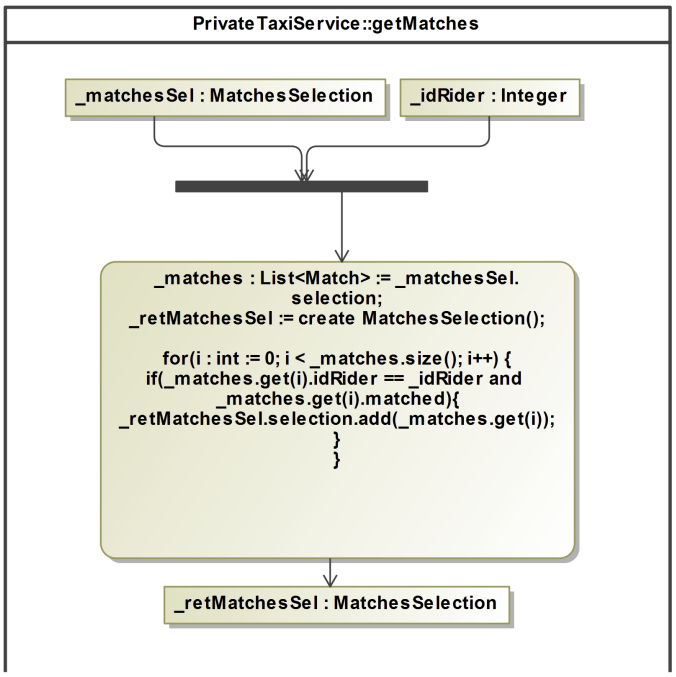


Abbildung 75: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zur Abfrage von Zuordnungen für einen Mitfahrer

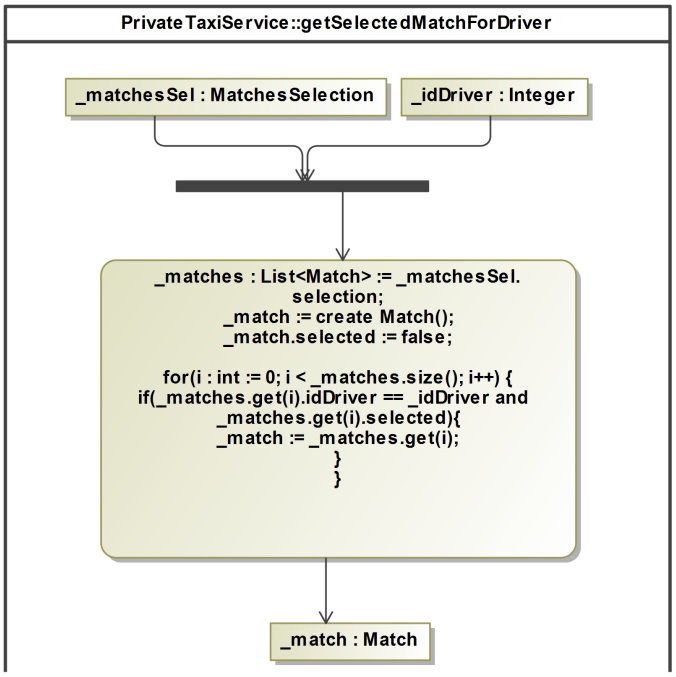


Abbildung 76: Aktivitätsdiagramm der PrivateTaxi-Fallstudie zur Abfrage einer Zuordnung für einen Fahrer

D

DEPLOYMENT- UND NUTZUNGSVORGABEN

Abschnitt 17.3 verdeutlicht, weshalb es nicht ausreichend ist, lediglich die Anwendung auf Sicherheitsmängel zu untersuchen, ohne das Gesamtsystem zu betrachten. Der gesamte Anwendungskontext bestehend aus den Hardware- und Software-Plattformen sowie den Anwendungsnutzern selbst spielt eine Rolle bei der Sicherheitsbetrachtung.

Jedoch ist es ebenfalls nicht ausreichend, in einem Ansatz zur Entwicklung (informationsfluss-)sicherer Anwendungen nur den Prozess der Entwicklung zu betrachten, ohne auf den Lebenszyklus und Benutzung der resultierenden Anwendung zu erwähnen. Annahmen, die in IFlow bei der Informationsflussanalyse getroffen werden, stützen sich auch auf den sicheren Deployment und Betrieb der entwickelten Anwendungen. Es muss also folglich eine Reihe von Vorgaben zum Deployment und Nutzung der Apps, Services, sowie mobiler und Server-Hardware eingehalten werden.

D.1 VORGABEN ZUM DEPLOYMENT

Beim Ausliefern und Installation von mit IFlow entwickelten Anwendungen müssen die folgenden Punkte beachtet werden:

- Bei der Auslieferung von IFlow-Apps muss sichergestellt werden, dass der Benutzer bereits die Anwendungen zur sicheren Eingabe (siehe Unterunterabschnitt 16.2.4.5) sowie zum Spezifizieren von Informationsflussanforderungen (siehe Unterunterabschnitt 16.2.4.1) installiert hat, oder diese zusammen mit den IFlow-Apps installiert werden
- Vertrieb der Anwendung soll über vertrauenswürdige Kanäle geschehen, die die Integrität und Authentizität der Installationspakete garantieren

D.2 VORGABEN FÜR ENTWICKLER

Entwickler von IFlow-Anwendungen sind angewiesen, sich an die folgenden Vorgaben zu halten:

- IFlow-Apps, die auf demselben Gerät installiert werden sollen, müssen mit demselben geheimen Schlüssel signiert sein (siehe Unterunterabschnitt 17.3.2.4)

- Die zum Signieren verwendeten geheimen Schlüssel müssen vertraulich aufbewahrt werden, da sie u.a. verwendet werden müssen, um aktualisierte Versionen der IFlow-Apps zu signieren (Unterunterabschnitt 17.3.2.3)

D.3 VORGABEN FÜR DIE ENDNUTZER

Auch die Endnutzer von IFlow-Apps müssen selbstständig eine Reihe von Punkten beachten, damit die Vertraulichkeit ihrer Daten sichergestellt ist:

- Entwickler-Optionen auf der Zielplattform wie etwa Sideloading (manuelle Installation fremder APKs) oder Debugging über die USB-Schnittstelle bei Android-Geräten dürfen nicht aktiviert sein
- Nutzung von mobilen Geräten, die nicht auf den neuesten Stand gebracht werden können soll gemieden werden
- Kritische Benutzereingaben dürfen nicht an öffentlichen Plätzen getätigt werden (um beispielsweise das sogenannte "Shoulder-Surfing" zu vermeiden)
- Kritische Benutzereingaben sollten nicht über nicht vertrauenswürdige Software-Tastaturen von Drittanbietern eingegeben werden
- Den Sicherheitsanweisungen von IFlow-Anwendungen am Bildschirm muss Folge geleistet werden
- Empfohlene Android-Version ist 6.0 oder höher, um u.a. vom Overlay-Schutz Gebrauch zu machen

Siehe Unterunterabschnitt 17.3.2.6 für Details zum Overlay-Schutz

D.4 VORGABEN FÜR DIE SERVICE-BETREIBER

Der Betreiber eines Webservices bzw. der System-Administrator der Server, auf dem diese Services eingesetzt werden, müssen den aufgeführten Vorgaben Folge leisten:

- Es muss sichergestellt werden, dass der Webserver nur über eine TLS-gesicherte Leitung erreichbar ist
- Der Server muss als sicher eingestufte *Cipher Suits* unterstützen, und Clients mit unsicheren *Cipher Suits* ablehnen
- Der Server muss ein gültiges TLS-Zertifikat besitzen
- Um höhere Sicherheit zu garantieren, sollten geteilte bzw. virtualisierte Hosting-Provider gemieden werden, da dies unter Umständen zusätzliche Angriffsvektoren ermöglicht (vgl. etwa [76])

- Der virtuelle und physikalische Zugang zum Server muss nur für autorisiertes Personal möglich sein
- Der Webservice darf nicht mit System- oder Administratorrechten laufen
- Der Betrieb des Webservices muss protokolliert werden, und die Protokolle müssen sicher und zugriffsbeschränkt aufbewahrt werden
- Es dürfen keine geheime Benutzerdaten außerhalb der mit IFlow entwickelten Anwendung gesammelt werden
- Fehlermeldungen dürfen nicht an den Benutzer ausgegeben werden, weil man daraus für einen Angriff potentiell notwendige Informationen erschließen kann
- Sicherheitsmechanismen wie Angriffserkennungssysteme (*Intrusion Detection Systems*), Angriffsabwehrsysteme (*Intrusion Prevention Systems*), und Firewalls sollten eingesetzt werden

D.5 ALLGEMEINE VORGABEN

Alle Nutzergruppen, die IFlow-Anwendungen entwickeln, einsetzen, oder pflegen, sind angewiesen, Folgendes zu beachten:

- Genutzte Software (Betriebssystem, installierte Anwendungen, Firmware etc.) muss auf dem neuesten Stand gehalten werden
- Sicherheitsmechanismen des genutzten Betriebssystems dürfen nicht durch den Nutzer deaktiviert werden (beispielsweise durch das "rooting" des Geräts, wodurch unautorisierte Anwendungen zu viele Rechte erhalten können)
- Die auf der Zielplattform verfügbaren Sicherheitsmechanismen müssen auch tatsächlich aktiviert und genutzt werden (wie etwa die Bildschirmsperre bei mobilen Geräten)
- Geheime Authentifikationsinformation für die genutzten Geräte oder Webserveraccounts wie PINs, Passwörter, geheime Schlüssel usw. müssen geheim gehalten werden
- "Operational security" (OPSEC): Achtsamkeit bei Interaktion mit anderen Personen oder Diensten, um nicht Opfer eines "social engineering"-Angriffs zu werden oder geheime Informationen versehentlich preiszugeben
- Sicherheit ist kein Zustand, sonder ein Prozess, bei dem zu jedem Zeitpunkt das Risiko eines erfolgreichen Angriffs abgewogen werden muss. Alle Nutzer und Betreiber eines sicherheitskritischen Systems müssen einen Plan parat haben für den Fall,

dass ihre Systeme kompromittiert werden und/oder geheime Daten gestohlen werden. Für manche Benutzergruppen ist das Risiko besonders hoch und die Folgen besonders gravierend; auf keine Fall dürfen diese sich ausschließlich Sicherheit der genutzten Software verlassen.

[1] K. Alghathbar, C. Farkas und D. Wijesekera. "Securing UML Information Flow Using FlowUML". In: *Journal of Research and Practice in Information Technology* 38.1 (2006), S. 111–121.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau und P. McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *SIGPLAN Not.* 49.6 (Juni 2014), S. 259–269. ISSN: 0362-1340. DOI: 10.1145/2666356.2594299. URL: http://doi.acm.org/10.1145/2666356.2594299.

[3] M. Balser, W. Reif, G. Schellhorn und K. Stenzel. "KIV 3.0 for Provably Correct Systems". In: *Proc. Int. Wsh. Applied Formal Methods*. Hrsg. von Dieter Hutter, Werner Stephan, Paolo Traverso und Markus Ullmann. Bd. 1641. LNCS. Springer, 1999, S. 330–337.

[4] M. Balser, W. Reif, G. Schellhorn, K. Stenzel und A. Thums. "Formal System Development with KIV". In: *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[5] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. H. Nguyen und J. Sifakis. "Rigorous Component-Based System Design Using the BIP Framework". In: *IEEE Software* 28.3 (Mai 2011), S. 41–48. ISSN: 0740-7459. DOI: 10.1109/MS.2011.27.

[6] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A. D. Schmidt und S. Albayrak. "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications". In: *2011 6th International Conference on Malicious and Unwanted Software*. Okt. 2011, S. 66–72. DOI: 10.1109/MALWARE.2011.6112328.

[7] D. E. Bell. "Looking back at the Bell-La Padula model". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. Dez. 2005, 15 pp.–351. DOI: 10.1109/CSAC.2005.37.

[8] N. Ben Said, T. Abdellatif, S. Bensalem und M. Bozga. "Model-Driven Information Flow Security for Component-Based Systems". English. In: *From Programs to Systems. The Systems perspective in Computing*. Hrsg. von Saddek Bensalem, Yassine Lakhneck und Axel Legay. Bd. 8415. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, S. 1–20. ISBN: 978-3-642-54847-5. DOI: 10.1007/978-3-642-54848-2_1. URL: http://dx.doi.org/10.1007/978-3-642-54848-2_1.

[9] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel und G. Vigna. "What the App is That? Deception and Countermeasures in the Android User Interface". In: *2015 IEEE Symposium on Security and Privacy*. Mai 2015, S. 931–948. DOI: 10.1109/SP.2015.62.

[10] N. Biasini, A. Chiu, J. Schultz und C. Williams. *Hundreds of Thousands of Google Apps Domains' Private WHOIS Information Disclosed*. http://blogs.cisco.com/security/talos/whoisdisclosure.

[11] M. Borek, K. Katkalov, N. Moebius, W. Reif, G. Schellhorn und K. Stenzel. "Integrating a Model-Driven Approach and Formal Verification for the Development of Secure Service Applications". English. In: *Correct Software in Web Applications and Web Services*. Hrsg. von Bernhard Thalheim, Klaus-Dieter Schewe, Andreas Prinz und Bruno Buchberger. Texts & Monographs in Symbolic Computation. Springer International Publishing, 2015, S. 45–81. ISBN: 978-3-319-17111-1. DOI: 10.1007/978-3-319-17112-8_3. URL: http://dx.doi.org/10.1007/978-3-319-17112-8_3.

[12] M. Borek, K. Stenzel, K. Katkalov und W. Reif. "Secure Integration of Third Party Components in a Model-Driven Approach". In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXX: Special Issue on Cloud Computing*. Hrsg. von Abdelkader Hameurlain, Josef Küng, Roland Wagner, Klaus-Dieter Schewe und Karoly Bosa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, S. 66–86. ISBN: 978-3-662-54054-1. DOI: 10.1007/978-3-662-54054-1_3. URL: http://dx.doi.org/10.1007/978-3-662-54054-1_3.

[13] E. Börger. "The Abstract State Machines method for high-level system design and analysis". In: *Formal Methods: State of the Art and New Directions* (2010), S. 79–116.

[14] Filip C. *Retailer's apps reveal your Christmas list to the public*. https://blog.avast.com/2015/12/15/retailers-apps-reveal-my-christmas-list-to-the-public/.

[15] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano und H. Srinivasan. "Dependence Analysis for Java". In: *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*. LCPC '99. London, UK, UK: Springer-Verlag, 2000, S. 35–52. ISBN: 3-540-67858-1. URL: http://dl.acm.org/citation.cfm?id=645677.663931.

[16] T. Chen und S. Zhong. "Privacy-preserving Backpropagation Neural Network Learning". In: *Trans. Neur. Netw.* 20.10 (Okt. 2009), S. 1554–1564. ISSN: 1045-9227. DOI: 10.1109/TNN.2009.2026902. URL: http://dx.doi.org/10.1109/TNN.2009.2026902.

[17] S. Chiasson, A. Forget, R. Biddle und P. C. van Oorschot. "User interface design affects security: patterns in click-based graphical passwords". In: *International Journal of Information Security* 8.6 (2009), S. 387. ISSN: 1615-5270. DOI: 10.1007/s10207-009-0080-7. URL: http://dx.doi.org/10.1007/s10207-009-0080-7.

[18] T. Close, Hewlett-Packard Labs und P. Alto. *Petname Tool: Enabling web site recognition using the existing SSL infrastructure*. https://www.w3.org/2005/Security/usability-ws/papers/02-hp-petname/. Blog. 2016.

[19] L.F. Cranor und S. Garfinkel. *Security and Usability: Designing Secure Systems that People Can Use*. O'Reilly Media, 2005. ISBN: 9780596553852.

[20] M. Dam. "Decidability and Proof Systems for Language-based Noninterference Relations". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, S. 67–78. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111044. URL: http://doi.acm.org/10.1145/1111037.1111044.

[21] T. Dierks und E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. Fremont, CA, USA: RFC Editor, Aug. 2008. DOI: 10.17487/RFC5246.

[22] D. Dolev, und A. C. Yao. "On the Security of Public Key Protocols". In: *Proceedings of the 22Nd Annual Symposium on Foundations of Computer Science*. SFCS '81. Washington, DC, USA: IEEE Computer Society, 1981, S. 350–357. DOI: 10.1109/SFCS.1981.32. URL: http://dx.doi.org/10.1109/SFCS.1981.32.

[23] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. 1st. San Francisco, CA, USA: No Starch Press, 2014. ISBN: 1593275811, 9781593275815.

[24] W. Enck, D. Octeau, P. McDaniel und S. Chaudhuri. "A study of android application security". In: *Proceedings of the 20th USENIX conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, S. 21–21. URL: http://dl.acm.org/citation.cfm?id=2028067.2028088.

[25] S. Ereth, H. Mantel und M. Perner. *Towards a Common Specification Language for Information-Flow Security in RS3 and Beyond: RIFL 1.0 - The Language*. Techn. Ber. TUD-CS-2014-0115. TU Darmstadt, 2014.

[26] A. P. Felt, E. Chin, S. Hanna, D. Song und D. Wagner. "Android permissions demystified". In: *Proceedings of the 18th ACM conference on Computer and communications security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, S. 627–638. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046779. URL: http://doi.acm.org/10.1145/2046707.2046779.

[27] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J A. Halderman, Z M. Mao und A. Prakash. "Android UI Deception Revisited: Attacks and Defenses". In: *Financial Cryptography and Data Security*. IFCA. 2016.

[28] J. Ferrante, K. J. Ottenstein und J. D. Warren. "The program dependence graph and its use in optimization". In: *International Symposium on Programming: 6th Colloquium Toulouse, April 17–19, 1984 Proceedings*. Hrsg. von M. Paul und B. Robinet. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, S. 125–132. ISBN: 978-3-540-38809-8. DOI: 10.1007/3-540-12925-1_33. URL: http://dx.doi.org/10.1007/3-540-12925-1_33.

[29] P. Fischer, K. Katkalov, K. Stenzel und W. Reif. *Formal Verification of Information Flow Secure Systems with IFlow*. Technical Report 2012-05. Universität Augsburg, 2012. URL: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

[30] D. Giffhorn und C. Hammer. "Precise Analysis of Java Programs using JOANA (Tool Demonstration)". In: *8th IEEE International Working Conference on Source Code Analysis and Manipulation*. Sep. 2008, S. 267–268. DOI: 10.1109/SCAM.2008.17.

[31] J.A. Goguen und J. Meseguer. "Security policies and security models". In: *Symposium on Security and privacy*. Bd. 12. IEEE, 1982.

[32] J. Graf, M Hecker und M. Mohr. "Using JOANA for Information Flow Control in Java Programs - A Practical Guide". In: *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*. Lecture Notes in Informatics (LNI) 215. accepted at ATPS 2013, replaces techreport 2012,24. Springer Berlin / Heidelberg, Feb. 2013.

[33] J. Graf, M. Hecker, M. Mohr und G. Snelting. "Checking Applications using Security APIs with JOANA". In: 8th International Workshop on Analysis of Security APIs. Juli 2015. URL: http://www.dsi.unive.it/~focardi/ASA8/.

[34] J. Graf, M. Hecker, M. Mohr und G. Snelting. "Tool Demonstration: JOANA". In: *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Hrsg. von

Frank Piessens und Luca Viganò. Bd. 9635. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016, S. 89–93. DOI: 10.1007/978-3-662-49635-0_5.

[35] C. Hammer. "Information Flow Control for Java - A Comprehensive Approach basedon Path Conditions in Dependence Graphs". ISBN 978-3-86644-398-3. Diss. Universität Karlsruhe (TH), Fak. f. Informatik, Juli 2009. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012049.

[36] C. Hammer. "Experiences with PDG-based IFC". In: *International Symposium on Engineering Secure Software and Systems (ESSoS'10)*. Springer LNCS 5965, 2010. DOI: 10.1007/978-3-642-11747-3_4.

[37] C. Hammer und G. Snelting. "Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs". In: *International Journal of Information Security* 8.6 (Dez. 2009). Supersedes ISSSE and ISoLA 2006, S. 399–422. DOI: 10.1007/s10207-009-0086-1.

[38] N. Hardy. "The Confused Deputy: (or why capabilities might have been invented)". In: *SIGOPS Oper. Syst. Rev.* 22.4 (Okt. 1988), S. 36–38. ISSN: 0163-5980. URL: http://dl.acm.org/citation.cfm?id=54289.871709.

[39] Ø. Haugen, K. E. Husa, R. K. Runde und K. Stølen. "STAIRS towards formal design with sequence diagrams". In: *Software & Systems Modeling* 4.4 (2005), S. 355. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0087-0. URL: http://dx.doi.org/10.1007/s10270-005-0087-0.

[40] R. Heldal und F. Hultin. "Bridging Model-Based and Language-Based Security". In: *Computer Security - ESORICS 2003*. Springer LNCS 2808, 2003. URL: http://dx.doi.org/10.1007/978-3-540-39650-5_14.

[41] R. Heldal, S. Schlager und J. Bende. "Supporting Confidentiality in UML: A Profile for the Decentralized Label Model". In: *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*. TU Munich Technical Report TUM-I0415. Munich, Germany, 2004, S. 56–70.

[42] R. Hennicker und N. Koch. "Modeling the User Interface of Web Applications with UML". In: *Workshop of the pUML-Group Held Together with the UML2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*. GI, 2001, S. 158–172. ISBN: 3-88579-335-0.

[43] N. P. Hoang, Y. Asano und M. Yoshikawa. "Your neighbors are my spies: Location and other privacy concerns in dating apps". In: *2016 18th International Conference on Advanced Com-*

munication Technology (ICACT). Jan. 2016, S. 715–721. DOI: 10.1109/ICACT.2016.7423532.

[44] P. Hornyack, S. Han, J. Jung, S. E. Schechter und D. Wetherall. "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications." In: *ACM Conference on Computer and Communications Security'11*. 2011, S. 639–652.

[45] S. Horwitz, T. Reps und D. Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, S. 35–46. ISBN: 0-89791-269-1. DOI: 10.1145/53990.53994. URL: http://doi.acm.org/10.1145/53990.53994.

[46] J. Jürjens. *Secure systems development with UML*. Springer Verlag, 2005.

[47] K. Katkalov, P. Fischer, K. Stenzel und W. Reif. *Model-Driven Code Generation of Information Flow Secure Systems with IFlow*. Technical Report 2012-04. Universität Augsburg, 2012. URL: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

[48] K. Katkalov, N. Moebius, K. Stenzel, M. Borek und W. Reif. "Model-Driven Testing of Security Protocols with SecureMDD". In: *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*. Mai 2012, S. 1–5. DOI: 10.1109/NTMS.2012.6208678.

[49] K. Katkalov, P. Fischer, K. Stenzel, N. Moebius und W. Reif. "Evaluation of Jif and Joana as Information Flow Analyzers in a Model-Driven Approach". In: *Data Privacy Management and Autonomous Spontaneous Security*. Springer LNCS 7732, 2013. ISBN: 978-3-642-35889-0.

[50] K. Katkalov, K. Stenzel, M. Borek und W. Reif. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *Proceedings of 5th ASE/IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*. IEEE Press, 2013.

[51] K. Katkalov, K. Stenzel, M. Borek und W. Reif. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *ASE Science Journal* 2.2 (2013), S. 65–82. URL: http://ojs.scienceengineering.org/index.php/science/article/view/90.

[52] K. Katkalov, N. Moebius, K. Stenzel, M. Borek und W. Reif. "Modeling Test Cases for Security Protocols with SecureMDD". In: *Comput. Netw.* 58 (Jan. 2014), S. 99–111. ISSN: 1389-1286. DOI:

10.1016/j.comnet.2013.08.024. URL: http://dx.doi.org/10.1016/j.comnet.2013.08.024.

[53] K. Katkalov, K. Stenzel, M. Borek und W. Reif. "Modeling Information Flow Properties with UML". In: *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*. DOI: 10.1109/NTMS.2015.7266507. IEEE Conference Publications, 2015.

[54] A. Kosba, A. Miller, E. Shi, Z. Wen und C. Papamanthou . "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy (SP)*. Mai 2016, S. 839–858. DOI: 10.1109/SP.2016.55.

[55] R. Küsters, T. Truderung und J. Graf. "A Framework for the Cryptographic Verification of Java-Like Programs". In: *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*. CSF '12. Washington, DC, USA: IEEE Computer Society, 2012, S. 198–212. ISBN: 978-0-7695-4718-3. DOI: 10.1109/CSF.2012.9. URL: http://dx.doi.org/10.1109/CSF.2012.9.

[56] R. Küsters, E. Scapin, T. Truderung und J. Graf. "Extending and Applying a Framework for the Cryptographic Verification of Java Programs". In: *Principles of Security and Trust, POST 2014, Part of ETAPS 2014, Grenoble, France, April 5-13, 2014*. Bd. 8424. Lecture Notes in Computer Science. Springer, 2014, S. 220–239. URL: http://www.etaps.org/index.php/2014/post.

[57] P. Lam, E. Bodden, O. Lhoták und L. Hendren. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infrastructure Workshop*. Galveston Island, TX, Okt. 2011.

[58] E. Levy. "Interface illusions". In: *IEEE Security Privacy* 2.6 (Nov. 2004), S. 66–69. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.104.

[59] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider und A. Weber. "Cassandra: Towards a Certifying App Store for Android". In: *Proceedings of the 4th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. ACM, 2014, S. 93–104.

[60] L. Luu, D. Chu, H. Olickel, P. Saxena und A. Hobor. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, S. 254–269. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978309. URL: http://doi.acm.org/10.1145/2976749.2978309.

[61] H. Mantel. “Possibilistic Definitions of Security - An Assembly Kit”. In: *IEEE Computer Security Foundations Workshop*. IEEE Press, 2000.

[62] H. Mantel. “A uniform framework for the formal specification and verification of information flow security”. eng. Diss. Postfach 151141, 66041 Saarbrücken: Saarländische Universitäts- und Landesbibliothek, 2003. URL: http://scidok.sulb.uni-saarland.de/volltexte/2004/202.

[63] R. van der Meyden. “What, indeed, is intransitive noninterference? (extended abstract)”. In: *Proc. European Symposium on Research in Computer Security*. Bd. 4734. An extended technical report is available from http://www.cse.unsw.edu.au/~meyden. Springer LNCS, 2007, S. 235–250.

[64] M. Might, Y. Smaragdakis und D. Van Horn. “Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis”. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, S. 305–315. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806631. URL: http://doi.acm.org/10.1145/1806596.1806631.

[65] N. Moebius. “Modellgetriebene Entwicklung sicherer Smart Card-Anwendungen”. Diss. Universität Augsburg, 2013.

[66] N. Moebius, K. Stenzel, H. Grandy und W. Reif. “’SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications’”. In: *Availability, Reliability and Security, 2009. ARES ’09. International Conference on*. März 2009, S. 841 –846. DOI: 10.1109/ARES.2009.22.

[67] M. Mohr, J. Graf und M. Hecker. “JoDroid: Adding Android Support to a Static Information Flow Control Tool”. In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015*. Bd. 1337. CEUR Workshop Proceedings. CEUR-WS.org, 2015, S. 140–145.

[68] A.C. Myers und B. Liskov. “Complete, safe information flow with decentralized labels”. In: *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*. Mai 1998, S. 186 –197. DOI: 10.1109/SECPRI.1998.674834.

[69] A.C. Myers und B. Liskov. “Protecting privacy using the decentralized label model”. In: *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*. 2003, S. 89 –116. DOI: 10.1109/FITS.2003.1264929.

[70] A. Nordrum. *Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated*. http://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated.

[71] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1.

[72] M. Ochoa, S. Pape, T. Ruhroth, B. Sprick, K. Stenzel und H. Sudbrock. *Report on the RS3 Topic Workshop "Security Properties in Software Engineering"*. Techn. Ber. 2012-2. Augsburg University, 2012.

[73] A. M. Olteanu, K. Huguenin, R. Shokri, M. Humbert und J. P. Hubaux. "Quantifying Interdependent Privacy Risks with Location Data". In: *IEEE Transactions on Mobile Computing* 16.3 (März 2017), S. 829–842. ISSN: 1536-1233. DOI: 10.1109/TMC.2016.2561281.

[74] F. Pires. *Uber hacking: how we found out who you are, where you are, and where you went*. https://labs.integrity.pt/articles/uber-hacking-how-we-found-out-who-you-are-where-you-are-and-where-you-went/.

[75] S. Preibusch. "Information Flow Control for Static Enforcement of User-Defined Privacy Policies". In: *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*. Juni 2011, S. 133 –136. DOI: 10.1109/POLICY.2011.23.

[76] R. Qiao und M. Seaborn. "A new approach for rowhammer attacks". In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. Mai 2016, S. 161–166. DOI: 10.1109/HST.2016.7495576.

[77] S. Rasthofer, S. Arzt und E. Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *NDSS*. 2014.

[78] A. W. Roscoe und M. H. Goldsmith. "What is Intransitive Noninterference?" In: *Proceedings of the 12th IEEE Workshop on Computer Security Foundations*. CSFW '99. Washington, DC, USA: IEEE Computer Society, 1999, S. 228–. ISBN: 0-7695-0201-6. URL: http://dl.acm.org/citation.cfm?id=794199.795122.

[79] S. Ruoti, J. Andersen, D. Zappala und K. E. Seamons. "Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client". In: *CoRR* abs/1510.08555 (2015). URL: http://arxiv.org/abs/1510.08555.

[80] J. Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies*. Techn. Ber. Dez. 1992. URL: http://www.csl.sri.com/papers/csl-92-2/.

[81] A. Sabelfeld und D. Sands. "Dimensions and principles of declassification". In: *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*. Juni 2005, S. 255 –269. DOI: 10.1109/CSFW.2005.15.

[82] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer und M. Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. Mai 2014, S. 459–474. DOI: 10.1109/SP.2014.36.

[83] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia und X. Wang. "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones". In: *Proc. of the 18th Network and Distributed System Security Symposium (NDSS'11), San Diego, CA, USA*. The Internet Society, Feb. 2011, S. 1–17. URL: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#SchlegelZZIKW11.

[84] F. Seehusen. "Model-Driven Security: Exemplified for Information Flow Properties and Policies". Diss. Faculty of Mathematics und Natural Sciences, University of Oslo, Jan. 2009. URL: http://folk.uio.no/ketils/kst/Theses/2009-01.SeehusenFredrik.pdf.

[85] F. Seehusen und K. Stølen. "Information flow property preserving transformation of UML interaction diagrams". In: *Proceedings of the eleventh ACM symposium on Access control models and technologies*. 2006, S. 150–159.

[86] L. Sfaxi, T. Abdellatif, R. Robbana und Y. Lakhnech. "Information Flow Control of Component-based Distributed Systems". In: *Concurr. Comput. : Pract. Exper.* 25.2 (Feb. 2013), S. 161–179. ISSN: 1532-0626. DOI: 10.1002/cpe.2807. URL: http://dx.doi.org/10.1002/cpe.2807.

[87] D. Silver u. a. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), S. 484–489. DOI: 10.1038/nature16961.

[88] G. Snelting, T. Robschink und J. Krinke. "Efficient Path Conditions in Dependence Graphs for Software Safety Analysis". In: *ACM Trans. Softw. Eng. Methodol.* 15.4 (Okt. 2006), S. 410–457. ISSN: 1049-331X. DOI: 10.1145/1178625.1178628. URL: http://doi.acm.org/10.1145/1178625.1178628.

[89] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr und D. Wasserrab. "Checking Probabilistic Noninterference Using JOANA". In: *it - Information Technology* 56 (Nov. 2014), S. 280–287. DOI: 10.1515/itit-2014-1051.

[90] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[91] K. Stenzel, K. Katkalov, M. Borek und W. Reif. "A Model-Driven Approach to Noninterference". In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 5.3 (Sep. 2014), S. 30–43.

[92] K. Stenzel, K. Katkalov, M. Borek und W. Reif. "Formalizing Information Flow Control in a Model-Driven Approach". In: *Information Communication Technology-EurAsia (ICT-EurAsia) 2014*. Springer LNCS 8407, 2014.

[93] K. Stenzel, K. Katkalov, M. Borek und W. Reif. "Declassification of Information with Complex Filter Functions". In: *Proceedings of the 2nd International Conference on Information Systems Security and Privacy*. 2016, S. 490–497. ISBN: 978-989-758-167-0.

[94] R. Vallee-Rai und L. J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998.

[95] T. Vidas, D. Votipka und N. Christin. "All Your Droid Are Belong to Us: A Survey of Current Android Attacks". In: *Proceedings of the 5th USENIX Conference on Offensive Technologies*. WOOT'11. San Francisco, CA: USENIX Association, 2011, S. 10–10. URL: http://dl.acm.org/citation.cfm?id=2028052.2028062.

[96] D. Volpano, C. Irvine und G. Smith. "A Sound Type System for Secure Flow Analysis". In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), S. 167–187. ISSN: 0926-227X. URL: http://dl.acm.org/citation.cfm?id=353629.353648.

[97] S. Wang, R. Sinnott und S. Nepal. "Protecting the location privacy of mobile social media users". In: *2016 IEEE International Conference on Big Data (Big Data)*. Dez. 2016, S. 1143–1150. DOI: 10.1109/BigData.2016.7840718.

[98] D. Wasserrab und D. Lohner. "Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing". In: *6th International Verification Workshop - VERIFY-2010*. 2010.

[99] D. Wasserrab, D. Lohner und G. Snelting. "On PDG-Based Noninterference and its Modular Proof". In: *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*. ACM, Juni 2009, S. 31–44. DOI: 10.1145/1554339.1554345.

[100] A. Whitten und J. D. Tygar. "Why Johnny Can'T Encrypt: A Usability Evaluation of PGP 5.0". In: *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999, S. 14–14. URL: http://dl.acm.org/citation.cfm?id=1251421.1251435.

[101] A. Whitten und J.D. Tygar. *Usability of Security: A Case Study*. School of Computer Science, Carnegie Mellon University, 1998.

[102] Z. Yang und M. Yang. "LeakMiner: Detect Information Leakage on Android with Static Taint Analysis". In: *2012 Third World Congress on Software Engineering*. Nov. 2012, S. 101–104. DOI: 10.1109/WCSE.2012.26.

[103] Z. Zhao und F. C. Colon Osono. ""TrustDroid™": Preventing the Use of SmartPhones for Information Leaking in Corporate Networks Through the Used of Static Analysis Taint Tracking". In: *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*. MALWARE '12. Washington, DC, USA: IEEE Computer Society, 2012, S. 135–143. ISBN: 978-1-4673-4880-5. DOI: 10.1109/MALWARE.2012.6461017. URL: http://dx.doi.org/10.1109/MALWARE.2012.6461017.

[104] Federal trade commission. "Protecting consumer privacy in an era of rapid change – A proposed framework for business and policymakers; Preliminary FTC staff report". In: *The Journal of Privacy and Confidentiality* 3.1 (2011), S. 67–140.

Sowohl maskuline, als auch feminine Personen- und Funktionsbezeichnungen dieser Arbeit beziehen sich in gleicher Weise auf alle Geschlechter.