Universität Paderborn

Fakultät für Elektrotechnik, Informatik und Mathematik

Institut für Informatik

Arbeitsgruppe *Programmiersprachen und Übersetzer*

Bestimmung des Speicherbedarfs für den Java-Card-Keller durch spezialisierte Aufrufgraphanalyse

Diplomarbeit

Konstantin Steinbrecher

November 2005

Vorgelegt bei

Prof. Dr. Uwe Kastens

und

Prof. Dr. Odej Kao

Erklärung

Hiermit versichre ich, dass ich diese Diplomarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle wörtlich oder inhaltlich zitierten Stellen sind als solche kenntlich gemacht.

Paderborn, 15. November 2005

Inhaltsverzeichnis

1 Einleitung

Die Smart Cards, auch als Prozessorkarten bezeichnet, sind Chipkarten, die neben verschiedenen Speicherarten auch einen Prozessor enthalten. Das erlaubt nicht nur die Speicherung der Daten, sondern auch die Durchführung von z. B. kryptographischen Berechnungen auf der Karte. Die Größe der auf den Chipkarten verwendeten Speichermodule ist relativ klein. Die aktuelle Generation von Chipkarten hat einen 8-bit-Prozessor, 196 kByte ROM, 64 kByte EEPROM und 6 kByte RAM.

Mit der Definition der Java-Card-Plattform wird ermöglicht, die in Java geschriebenen Anwendungen auf der Chipkarte ausführen zu können. Diese Tatsache bringt die bekannten Vorteile der Java-Plattform mit sich. Eine der wichtigsten ist die Plattformunabhängigkeit. Diese Eigenschaft wird umgesetzt, indem der prozessorunabhängige Java Card Bytecode generiert und von der virtuellen Java-Card-Maschine ausgeführt wird. Java-Card-Technologie bietet die Möglichkeit, zusätzlich neue Java-Card-Anwendungen auf die Chipkarte nach der Herstellung installieren zu lassen. Mit dem Starten der virtuellen Java-Card-Maschine wird unter anderem ein Teil des RAM auf der Karte für den Aufrufkeller reserviert, um die Methodenschachtel der aufgerufenen Methoden dort abzulegen. Um mit Ressourcen sparsam umzugehen, soll dieser Bereich klein wie möglich und so groß wie nötig gewählt werden. Man könnte vor der Installation einer Anwendung auf die Karte die Größe von deren Aufrufkeller ermitteln und – wenn nötig – die Optimierungen an der Anwendung vorschlagen bzw. durchführen.

In dieser Diplomarbeit wird ein Verfahren zur Bestimmung der Größe des Aufrufkellers einer Java-Card-Anwendung erarbeitet und umgesetzt. Das Verfahren basiert auf Aufrufgraphen, die mit statischen Analyseverfahren erzeugt wurden. Dabei steht der Aspekt der dynamischen Methodenbindung im Vordergrund. Außerdem werden Besonderheiten der Java-Card-Plattform berücksichtigt, um hierdurch möglichst präzise Ergebnisse zu ermitteln.

Das Kapitel 2 bereitet Grundlagen für das Verfahren vor. Mit Hilfe einer kleinen Einführung in die Chipkartenarchitektur werden die zentralen Eigenschaften der Java-Card-Plattform beschrieben. Außerdem werden die Datenstruktur Aufrufgraph erläutert und einige statische Aufrufgraphanalyseverfahren präsentiert. Im Kapitel 3 wird das erarbeitete Verfahren zur Ermittlung der Größe des Aufrufkellers einer Java-Card-Anwendung vorgestellt. Kapitel 4 stellt interessante Aspekte der Realisierung des entworfenen Verfahrens vor und Kapitel 5 behandelt die Auswertung der Messergebnissen. Eine Zusammenfassung im Kapitel 6 schließt die Arbeit ab.

2 Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, aus denen das Verfahren zur Bestimmung der maximalen Kellertiefe einer Java-Card-Anwendung sich entwickelt hat. Nach einer kurzen Einführung in die Welt der Chipkarten werden für die gestellte Aufgabe interessante Eigenschaften der Java-Card-Plattform vorgestellt und deren Auswirkungen auf das zu entwickelnde Verfahren diskutiert. Es geht dabei grundsätzlich um die Einschränkungen der Programmiersprache Java, die die Java-Card-Plattform definiert, und die spezifischen Merkmale der Java-Card-Architektur. Diese Eigenschaften werden beim Entwickeln des Verfahrens berücksichtigt. Als Nächstes wird der Begriff des Aufrufgraphen eingeführt. Dabei wird auf die Problematik bei der Erzeugung von Aufrufgraphen für Java-Card-Anwendungen mit statischen Analyseverfahren eingegangen. Abschließend werden einige statische Analyseverfahren zum Aufbau von Aufrufgraphen vorgestellt und anhand eines kleinen Beispiels erläutert.

2.1 Chipkarten

Eine Chipkarte ist ein Plastikkörper, der eine integrierte Schaltung enthält. Die integrierte Schaltung ermöglicht sowohl die Speicherung von Daten als auch die Durchführung von Berechnungen auf diesen Daten. Die Kommunikation mit der Außenwelt ist über die Kontakte an der Oberfläche der Karte oder kontaktlos durch elektromagnetische Felder möglich. Grundsätzlich gibt es zwei Arten von Chipkarten: Speicher- und Prozessorkarten.

Die Speicherkarten besitzen einen lesbaren Speicher ROM und einen wiederbeschreibbaren Speicher, meist EEPROM. Das ROM enthält Daten, die im weiteren Verlauf nicht geändert werden sollen, und Identifizierungsdaten. Im EEPROM werden Anwendungsdaten untergebracht. Die Berechnungen werden auf dem Lesegerät durchgeführt und auf die Karte zurückgeschrieben. Die Speicherkarten sind für eine bestimmte Aufgabe optimiert, sodass eine weitere Erweiterung mit zusätzlichen Anwendungen nicht möglich ist.

Die Prozessorkarten enthalten, wie der Name schon sagt, zusätzlich einen Mikroprozessor. Die Anwesenheit des Prozessors auf der Karte erhöht den Sicherheits- und Multifunktionalitätsfaktor der Karte erheblich. Aus diesem Grund werden sie auch als Smart Card bezeichnet. Der Mikroprozessor steuert die Datenverarbeitung und die Speicherzugriffe auf der Schaltung. Es können kryptographische Berechnungen lokal auf der Karte durchgeführt werden. Der Prozessor auf der Karte ist in der Regel mit weiteren Funktionsblöcken umgeben: einem lesbaren Speicher (ROM), einem nicht flüchtigen wiederbeschreibbaren Speicher (EEPROM) und einem flüchtigen wiederbeschreibbaren Speicher (RAM). Das ROM beinhaltet das Betriebssystem der Karte,

das während der Herstellung eingebrannt wird. Im EEPROM werden Daten und Teile der dynamisch hinzugeladenen Anwendungen gespeichert. Das RAM wird während der Ausführung der Berechnungen auf dem Mikroprozessor zur Aufnahme von Prozedurschachteln benutzt. Die Ein- und Ausgabeschnittstellen dienen der seriellen Kommunikation mit der Außenwelt. Die Prozessorkarten sind in der Anwendung sehr flexibel. Es ist möglich, verschiedene Anwendungen in einer einzigen Karte zu integrieren. Die Funktionalität der Karte ist durch geringe Speicherkapazität und Rechenleistung beschränkt [Rankl].

2.2 Java-Card-Technologie

Java-Card-Technologie ermöglicht die Ausführung von mit der Programmiersprache Java geschriebenen Anwendungen auf Chipkarten bzw. Geräten mit begrenzten Ressourcen. Aus der Entwicklung der Programme für Chipkarten in Java ergeben sich die grundsätzlichen Vorteile der Java-Plattform. Zusätzlich wird die Verwendung von mehreren Anwendungen auf der Karte unterstützt. Java-Card-Technologie kombiniert eine Untermenge der Programmiersprache Java mit der für Chipkarten optimierten Laufzeitumgebung. Die für Java Card entwickelten Java-Programme werden als Java Card Applets bezeichnet.

2.2.1 Untermenge der Java-Sprache

Auf Grund der Beschränkungen der Ressourcen auf der Chipkarte wurden einige Einschränkungen der Spezifikation der Java-Sprache hervorgerufen. Es folgt eine Aufzählung der ausgewählten Eigenschaften der Programmiersprache Java, die von der Java-Card-Plattform nicht unterstützt werden.

Dynamisches Laden von Klassen: In einer Implementierung der Java-Card-Plattform ist es nicht möglich, Klassen dynamisch zu laden. Klassen werden entweder während der Herstellung auf die Karte gebrannt oder nach der Herstellung installiert. Programme, die auf der Karte ausgeführt werden, können nur die auf der Chipkarte vorhandenen Klassen referenzieren. Dies erleichtert die Analyse von Java-Card-Anwendungen mit Hilfe statischer Verfahren. Bei der Verwendung von statischen Analyseverfahren sollte man vorsichtige Annahmen über das Verhalten der Methoden von unbekannten Klassen machen. Deswegen können präzisere Ergebnisse bei der Analyse der Java-Card-Anwendungen erwartet werden.

Ausführungsstränge: die virtuelle Java-Card-Maschine bietet keine Unterstützung im Hinblick auf die Ausführung von mehreren Ausführungssträngen. Es wird immer nur ein Ausführungsstrang ausgeführt. Laut Java-Spezifikation wird ein Aufrufkeller pro Ausführungsstrang angelegt. Im Fall der virtuellen Java-Card-Maschine wird ein Aufrufkeller verwendet.

Datentypen: Einige einfache Datentypen werden nicht unterstützt: z. B. `char`, `double`, `float` und `long`. Mehrdimensionale Felder finden auf Java Card ebenfalls

keine Verwendung. Folglich werden zugehörige Schlüsselwörter und Bytecode-Instruktionen nicht unterstützt.

Es sind einige Eigenschaften der Java-Card-Plattform zu nennen, die weiterhin unterstützt werden. *Dynamische Objekterzeugung*: Die Java-Card-Plattform unterstützt dynamisch erzeugte Objekte wie Klasseninstanzen und Arrays. *Virtuelle Methoden*: Die Java-Card-Objekte sind identisch mit den Java-Objekten. Folglich sind virtuelle Methodenaufrufe in Java Card Applets exakt die gleichen wie in Java-Programmen. Die Vererbung und dynamische Methodenbindung werden ebenfalls unterstützt. Beim Aufruf einer dynamisch gebundenen Methode wird der Typ des Empfängerobjektes erst zur Laufzeit bekannt. Während der Ausführung können an dieser Stelle der zur Übersetzungszeit statisch bestimmte Typ selbst und alle seine Untertypen auftreten. Die dynamische Methodenbindung spielt eine wichtige Rolle bei der Erzeugung von Aufrufgraphen mit Hilfe statischer Analyseverfahren. *Schnittstellen*: Die Schnittstellen werden von Java-Card-Klassen implementiert, so wie es in der Java-Spezifikation definiert ist. Es werden weiterhin folgende *einfache Datentypen* unterstützt: `boolean`, `byte`, `short` und optional `int` [Chen]. Die verwendeten Datentypen beeinflussen die Größe der Methodenschachtel, die beim Ausführen der Java-Card-Anwendung auf den Java-Card-Aufrufkeller geladen wird.

2.2.2 Java-Card-Laufzeitumgebung

Die Java-Card-Laufzeitumgebung übernimmt die Ressourcenverwaltung, Appletausführung und weitere Dienste eines Betriebssystems. Sie bietet eine einheitliche Schnittstelle für den Zugriff der Java Card Applets auf die herstellerspezifischen Eigenschaften der Chipkarte. Die Laufzeitumgebung enthält die virtuelle Java-Card-Maschine (Bytecode-Interpretierer), Java Card API mit anwendungsgebietspezifischen Erweiterungen und Systemklassen.

Die virtuelle Java-Card-Maschine führt den Bytecode aus, verwaltet die Speicherzuordnung und gewährleistet die Laufzeitsicherheit. Java Card API besteht aus vier Java-Paketen. Die Systemklassen spielen die Rolle des Betriebssystemkerns. Sie bieten Dienste für die Transaktionsverwaltung sowie die Kommunikation zwischen Java Card Applet und Hostanwendung. Auf dem Bild 2.1 ist die Java-Card-Laufzeitumgebung schematisch dargestellt.

Die Lebenszeit der Java-Card-Laufzeitumgebung und somit der virtuellen Java-Card-Maschine beginnt nach der Herstellung der Chipkarte. Die Laufzeitumgebung wird während der Initialisierungsphase der Karte initialisiert. Nach dem Entziehen der Energie bleiben Daten im nicht flüchtigen wiederbeschreibbaren Speicher erhalten. Die virtuelle Java-Card-Maschine terminiert nicht, sondern wird angehalten. Bei erneuter Zuführung der Energie wird die Ausführung der virtuellen Java-Card-Maschine fortgesetzt. Die Lebenszeit der Java-Card-Laufzeitumgebung endet mit Vernichtung der Chipkarte.

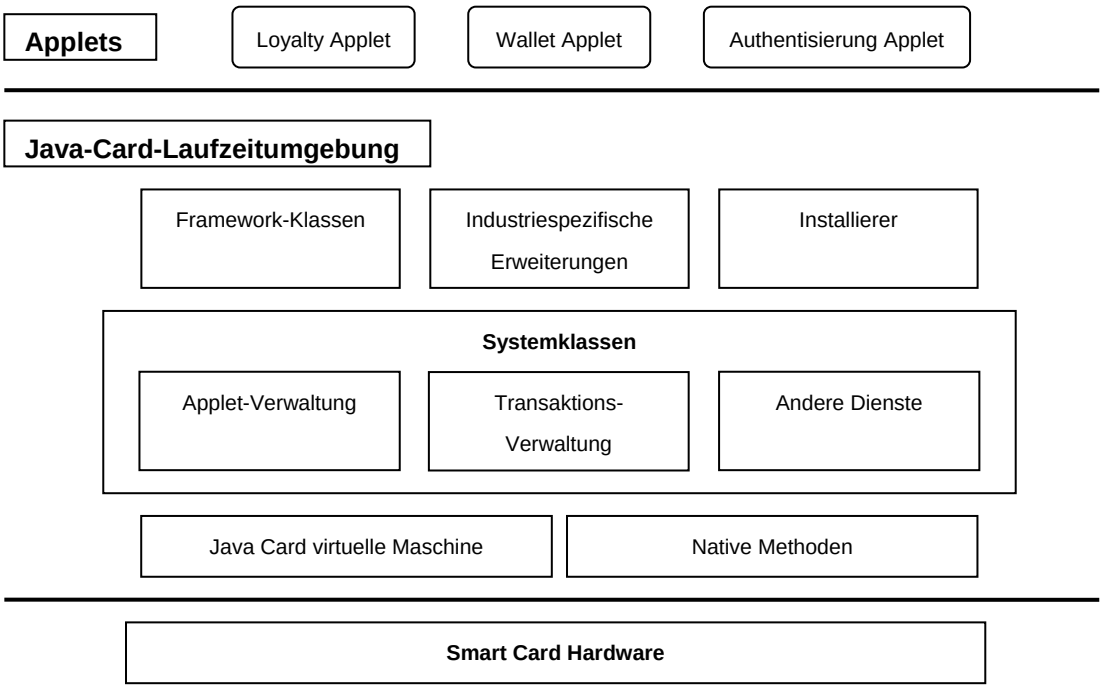

Bild 2.1: Java Card Architektur

2.2.3 Virtuelle Java-Card-Maschine

Die virtuelle Java-Card-Maschine besteht aus zwei Teilen: zum einen aus der virtuellen Maschine auf der Karte (on-card) und zum anderen aus der auf dem PC (off-card). Der auf der Karte laufende Teil enthält den Bytecode-Interpretierer. Seine Aufgabe ist die Ausführung der Java-Bytecode-Instruktionen, die Verwaltung der Objekterzeugung und Speicherzuweisung sowie die Gewährleistung der Laufzeitsicherheit.

Der auf dem PC laufende Teil der virtuellen Java-Card-Maschine wird als Converter bezeichnet. Die Aufgabe des Converters ist es, die Java-Klassendateien, die zu einem Java-Paket gehören, zu laden und daraus eine CAP-Datei (converted applet) zu erzeugen. Während der Konvertierung werden typische Aufgaben gelöst, die die virtuelle Java-Maschine beim Laden der Java-Klassendateien verrichtet, so wie z. B. die Verifikation und Initialisierung statischer Variablen. Nach der Erzeugung kann die CAP-Datei auf die Karte installiert werden.

Die virtuelle Java-Card-Maschine bietet Unterstützung für jeweils nur einen Ausführungsstrang. Infolgedessen sind die mit einem neuen Ausführungsstrang angelegten Speicherbereiche nur einmal vorhanden. Es gibt z. B. nur einen Java-Card-Aufrufkeller auf der Karte. Der Java-Card-Aufrufkeller weist die gleiche Funktionalität wie der Java-Aufrufkeller auf. Für jede aufgerufene Methode wird die Methodenschachtel auf den Keller gelegt. Mit Terminierung der Methode wird deren Schachtel aus dem Keller entfernt.

Eine Methodeschachtel enthält einen Array mit lokalen Variablen und einen Operandenkeller, auf dem die Berechnungen mit lokalen Variablen durchgeführt werden. Allerdings weicht die Anzahl der innerhalb einer Methodenschachtel der virtuellen Java-Card-Maschine verwendeten Datenstrukturen von der Methodenschachtel der virtuellen Java-Maschine in einem Punkt ab. Die Methodenparameter in einer Methodenschachtel der virtuellen Java-Card-Maschine werden nicht in dem Array mit lokalen Variablen, sondern in einer separaten Datenstruktur gespeichert. Die Anzahl der Zellen auf dem Operandenkeller und der Array mit lokalen Variablen werden zur Übersetzungszeit festgelegt, wobei eine Zelle des Arrays mit lokalen Variablen bzw. des Operandenkellers ein Wort groß ist. Bei den Java-Card-Anwendungen werden diese Datenstrukturen bei der Konvertierung in eine CAP-Datei aktualisiert.

Die virtuelle Java-Card-Maschine legt folgende Einschränkungen bezüglich der in einer Schachtel verwendeten Datenstrukturen fest: Die Anzahl der lokalen Variablen in einer Methode ist auf 255 begrenzt; die maximale Tiefe des Operandenkellers ist mit 255 Zellen beschränkt; die Größe des Wortes wurde auf 16 Bit festgelegt.

Die virtuelle Java-Card-Maschine bietet eine begrenzte Unterstützung für die Initialisierung von statischen Feldern in der `<clinit>`-Methode. Statische Felder von Applets werden zu primitiven übersetzungszeitkonstanten Werten bzw. zu Arrays von primitiven Übersetzungszeit-Konstanten initialisiert. Statische Felder von Benutzerbibliotheken können nur zu primitiven konstanten Werten initialisiert werden.

2.2.4 Java Card APIs

Die einheitliche Schnittstelle für Java Card Applets wird mittels der von der Java-Card-Laufzeitumgebung zur Verfügung gestellten Java Card API erreicht. Diese API besteht aus vier Paketen. Drei dieser Pakete – `java.lang`, `javacard.framework` und `javacard.security` – sind als so genannte Kernpakete für alle Java Cards verpflichtend. Das vierte – `javacardx.crypto` – ist ein Erweiterungspaket, das bei Bedarf hinzugefügt werden kann. Darüber hinaus gibt es noch eine Reihe weiterer anwendungsspezifischer Pakete.

Die Basis für Java auf Chipkarten ist das Paket `java.lang`. In diesem werden die Basisklasse `java.lang.Objekt` und elementare Klassen für Ausnahmen definiert. Das Paket `javacard.framework` stellt Kernfunktionen für eine Java-Card-Anwendung (Applet) bereit. Darunter sind vor allem elementare „Klassen für Appletverwaltung, Datenaustausch mit dem Terminal und diverse Konstanten im Rahmen von ISO/IEC 7816-4“ [Rankl]. Das Paket `javacard.security` stellt Schnittstellen für eine Reihe von Kryptoalgorithmen zur Verfügung. Die Schnittstellen zur Entschlüsselungsmethode bietet das Erweiterungspaket `javacardx.crypto` an.

2.2.5 Java Card Applets

Java Card Applet ist eine Java-Programm, das in der Java-Card-Laufzeitumgebung ausgeführt werden kann. Dafür muss eine Programmklasse von der Klasse

`javacard.framework.Applet` der Java Card API abgeleitet werden. Diese Klasse definiert unter anderem die Methoden `install`, `select`, `process` und `deselect`. Mittels dieser Methoden wird das Verhalten einer Java-Card-Anwendung festgelegt. Dafür werden diese in der abgeleiteten Klasse überschrieben. Die Java-Card-Laufzeitumgebung ruft diese Methoden zu den verschiedenen Zeitpunkten der Java-Card-Applet-Erzeugung und Ausführung auf. Sie sind die Eintrittspunkte eines Java Card Applets.

Die Java-Card-Laufzeitumgebung bietet Unterstützung für die Koexistenz von mehreren Java-Card-Anwendungen auf einer Chipkarte. Es gibt die Möglichkeit, ein Java Card Applet nachträglich auf die Karte installieren zu lassen. Dies trägt zur Multifunktionalität der Java Card bei. Die Voraussetzung dafür ist genügend Speicherplatz für die Installation der zusätzlich zu der Java Card API benötigten Pakete (EEPROM) und für die spätere Ausführung des Applets (RAM). Im letzten Schritt der Appletinstallation wird die Methode `install` aufgerufen. In dieser Methode wird ein Exemplar der Unterklasse `javacard.framework.Applet` erzeugt. In dem Konstruktor dieser Klasse werden benötigte Objekte und Variablen initialisiert. Anschließend wird das installierte Applet bei der Java-Card-Laufzeitumgebung registriert. Für die Bearbeitung einer Anfrage muss ein Applet aktiviert werden, indem die von ihm implementierte `select`-Methode aufgerufen wird. Diese Methode enthält die für die Ausführung benötigte Initialisierungen bzw. Allokation der verwendeten Ressourcen. Die Anfragen von der Host-Anwendung werden mittels `process`-Methode empfangen und weiter verarbeitet. Wenn ein anderes Applet ausgeführt werden soll, wird aktuell laufender mit Aufruf der Methode `deselect` deaktiviert. In der Methode kann die Freigabe der von dem Applet verwendeten Ressourcen implementiert werden.

2.3 Aufrufgraph

Ein Aufrufgraph repräsentiert Aufrufbeziehungen zwischen den Methoden in einem Programm. Dabei handelt es sich um einen gerichteten Graph. Die Knoten repräsentieren die Methoden und die gerichteten Kanten die Aufrufbeziehungen zwischen den Methoden. Es liegt eine Kante zwischen zwei Knoten genau dann vor, wenn im Rumpf einer Methode eine Aufrufstelle existiert, deren Aufrufziel die andere Methode ist, bzw. sein könnte.

In der einfachen Ausführung eines Aufrufgraphen werden Aufrufziele der aufrufenden Methode zugeordnet. Außerdem gibt es die Möglichkeit, die Aufrufziele den jeweiligen Aufrufstellen zuzuordnen. Dadurch kann man erkennen, welche Instruktion mehrere Aufrufziele hat. Auf dem Bild 2.2 ist ein Aufrufgraph dargestellt. In diesem Aufrufgraph sind die Aufrufziele den aufrufenden Methoden zugeordnet.

```
public class A {
  int b;
  void main() {
    i();
    p();
  }

  void i() {
    b = 1;
  }

  void p() {
    r();
  }

  void r() {
    b = b + 2;
  }
}
```

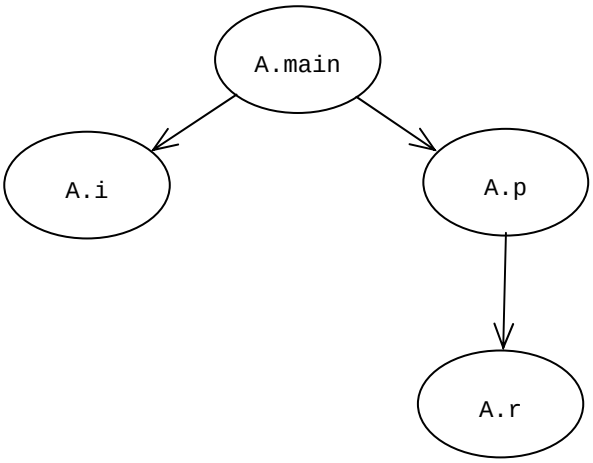

Bild 2.2: Die Auflistung eines Programms und dazugehörige Aufrufgraph mit der Zuordnung der Aufrufziele den aufrufenden Methoden.

In einem Programm gibt es mindestens eine ausgezeichnete Methode, mit der die Ausführung des Programms beginnt. In einer Java-Anwendung ist das die `main`-Methode. Außerdem gibt es einige Methoden, die unmittelbar von der virtuellen Java-Maschine aufgerufen werden, wie z. B. die `<clinit>`-Methode. Solche Methoden werden als Eintrittspunkte des Programms bezeichnet. In einem Aufrufgraph sind dies die Eintrittsknoten. Sie haben die Eigenschaft, dass es keine eingehenden, sondern nur ausgehende Kanten zu diesen Knoten gibt. Auf dem Bild 2.2 ist das der Knoten, der durch Methode `A.main` repräsentiert ist. Vom Eintrittsknoten aus erreichbare Knoten repräsentieren potenziell benutzte Methoden in der Anwendung.

Im Gegensatz zu den Eintrittsknoten in einem Aufrufgraph gibt es Knoten, die lediglich eingehende, aber keine ausgehenden Kanten besitzen. Solche Kanten werden als Blätter in einem Aufrufgraph bezeichnet. Im Aufrufgraph auf dem Bild 2.2 sind es Knoten, die durch Methoden `A.i` und `A.r` repräsentiert sind. Übertragen auf die Programmebene repräsentieren solche Knoten Methoden, die in einer möglichen Methodenausführungskette an letzter Stelle stehen. Die möglichen Methodenausführungskette bzw. Aufrufketten einer Anwendung sind in einem Aufrufgraph durch Pfade vom Eintrittspunkt bis zu den Blättern dargestellt. Eine Aufrufkette ist eine Folge von Methodenaufrufen, die während der Ausführung nacheinander aufgerufen werden. In dem vorgestellten Beispiel eine mögliche Aufrufkette ist z. B. Methodenaufruffolge `A.main`, `A.p`, `A.r`.

Für die Bestimmung von Aufrufgraphen sind verschiedene statische Analyseverfahren bekannt. Die statischen Analysen versuchen zur Übersetzungszeit unter anderem Aussagen über das Laufzeitverhalten eines Programms zu machen, ohne dabei das Programm auszuführen. Auf Grund dynamischer Methodenbindung ist es nicht immer möglich, für eine Aufrufstelle das Aufrufziel eindeutig zu bestimmen. Um in solchen Fällen keine falschen Ergebnisse abzuliefern, wird konservativ abgeschätzt. Potenziell

könnten an diesen Stellen während der Ausführung der zur Übersetzungszeit statisch bestimmte Typ selbst und alle seine Untertypen auftreten, so dass ohne zusätzliche Informationen, mit deren Hilfe die Aufrufziele eingeschränkt werden könnten, alle Methoden mit passender Signatur in den Aufrufgraph aufgenommen werden müssen. Dadurch werden in einem Aufrufgraph mehrere Aufrufziele einer Aufrufstelle zugeordnet. Dementsprechend können unpräzise Ergebnisse entstehen. Dafür haben statische Analysen gegenüber Beobachtungen der Anwendung zur Laufzeit den Vorteil, dass die ermittelten Eigenschaften für alle möglichen Programmausführungen und Programmeingaben gelten, während die dynamischen Analysen nur Aussagen über die ausgeführten Programmteile mit speziellen Eingaben machen können.

Die für die Bestimmung des Aufrufgraphen bekannten Ansätze beruhen auf der interprozeduralen Kontrollflussanalyse von Anwendungen. Allen diesen Analyseverfahren ist gemein, dass sie eine Menge der erreichbaren Methoden aufbauen. Ausgehend von einem Anfangszustand, wo diese Menge mit Eintrittspunkten initialisiert wird, werden alle Methoden der Menge iterativ nach Aufrufstellen analysiert und deren potenziellen Aufrufziele der Menge hinzugefügt.

Die Verfahren unterscheiden sich nach der Art und Weise, wie die Aufrufziele bestimmt werden. Einige Verfahren gehen dabei flussinsensitiv und andere flusssensitiv vor. Bei der flussinsensitiven Analyse werden Informationen über die Datenmanipulation des Programms nicht berücksichtigt. Es werden alle Methoden zu der Menge hinzugefügt, die potentiell an dieser Stelle aufgerufen werden können. Die flusssensitive Analyse verwendet die Kontrollflussinformationen bzw. Datenflussinformationen innerhalb eines Programms. Dadurch können die Aufrufziele innerhalb einer Methode genauer identifiziert werden.

2.4 Verfahren zur Aufbau der Aufrufgraphen

In diesem Kapitel sind einige statische Analyseverfahren zur Bestimmung des Aufrufgraphen einer Java-Anwendung vorgestellt. Sie bestimmen den Aufrufgraph mit steigender Genauigkeit mit immer höherem Zeit- und Platzaufwand. Um die Menge der potenziellen Aufrufziele zu reduzieren, verwenden die Verfahren eine unterschiedliche Anzahl von Hilfsmengen. Die allgemeine Idee ist dabei, ein Objekt zu seiner Klasse zu abstrahieren und eine Menge von Objekten zu der Menge von Klassen. Die Funktionsweise der Aufrufgraphanalyseverfahren wird anhand eines Beispiels veranschaulicht.

2.4.1 Beispielprogramm

Die Funktionsweise der im Weiteren vorgestellten Aufrufgraphanalyseverfahren wird mit Hilfe eines kleinen Java-Programms demonstriert. Auf dem Bild 2.3 ist das Beispielprogramm aufgelistet. Das Programm enthält vier Typen von Aufrufinstruktionen des Java Byte Codes. Das sind zwei statisch gebundenen (`invokestatic` und `invokespetial`) und zwei dynamisch gebundenen

(`invokevirtual` und `invokeinterface`). Ein Aufrufgraph, der die Aufrufbeziehungen zwischen den Methoden des betrachteten Beispielprogramms repräsentiert, ist auf dem Bild 2.4 dargestellt. Auf Grund der Übersichtlichkeit werden Knoten und Kanten, die Konstruktoren und Konstruktoraufrufe repräsentieren, im Weiteren nicht betrachtet.

```
public interface S {
  public void c();
}

public class T implements S {
  public void c() {
    int x = 2;
    int y = x + 3;
  }
}

public class U implements S {
  public void c() {
    int x = 4;
    int y = x;
  }
}

public class X {
  void m() {
    int a = 0;
    int b = a + 1;
  }
}

public class Y extends X {
  void m() {
    int a = 1;
  }
}
```

```
public class A {
  S[] s;
  X x;
  static void main() {
    A a = new A();
    a.i();
    a.p();
  }

  void i() {
    s = new S[1];
    T t = new T();
    s[0] = t;
    x = new X();
    x.m();
  }

  void p() {
    A.r();
    for (int i=0;i<s.length;i++) {
      s[i].c();
    }
  }

  static void r() {
    X y;
    y = new X();
    y = new Y();
    y.m();
  }
```

Bild 2.3: Die Auflistung des Beispielprogramms.

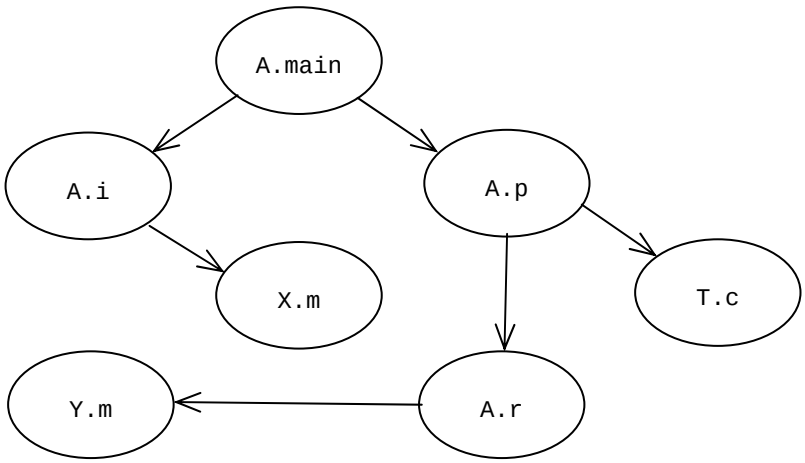

Bild 2.4: Ein Aufrufgraph für das Beispielprogramm.

2.4.2 Class Hierarchy Analysis

Die Class Hierarchy Analysis (CHA) baut eine Menge der von den Einrittspunkten aus erreichbaren Methoden auf. Für die Bestimmung der Aufrufziele verwendet die CHA die Signatur einer Methode und die Klassenhierarchieinformationen der Anwendung.

Die Signatur einer Methode enthält den Namen der Methode und die Parameter. Die Klassenhierarchie in einer objektorientierten Anwendung beschreibt, wie die Klassen bzw. Schnittstellen in einer Vererbungsrelation zueinander stehen.

Die Analyse wird mit der Initialisierung der Menge erreichbarer Methoden mit Eintrittspunkten gestartet. Daraufhin werden Methoden dieser Menge iterativ analysiert. In jeder Methode werden Aufrufstellen identifiziert. Für jede Aufrufstelle werden potenzielle Aufrufziele wie folgt bestimmt und der Menge der erreichbaren Methoden hinzugefügt: In Java gibt es vier Aufrufinstruktionen. Zwei davon sind statische und zwei dynamische. Dementsprechend gibt es im Programm statische und dynamische Aufrufstellen. Bei statischen Aufrufstellen sind die Aufrufziele zur Übersetzungszeit eindeutig bestimmbar, im Falle eines dynamischen Aufrufs auf Grund der dynamischen Methodenbindung nicht. Es wird der statische Typ des Empfängerobjekts bestimmt. In der bestimmten Klasse und den mit Hilfe der Klassenhierarchie ermittelten Unterklassen wird nach Implementierungen der aufgerufenen Methode durchsucht. Die auf diese Weise identifizierten Methoden werden in die Menge der erreichbaren Methoden aufgenommen. Für den Fall, dass die aufgerufene Methode von der statisch bestimmten Klasse geerbt und nicht überschrieben wird, muss die geerbte Implementierung in den Oberklassen identifiziert werden.

Die auf diese Weise bestimmten Aufrufziele schließen Methoden mit gleicher Signatur, aber anderem Kontext aus. Auf dem Bild 2.5 ist ein Aufrufgraph dargestellt, der mit CHA für das vorgestellte Beispielprogramm erzeugt wurde. Für drei dynamisch gebundene Aufrufstellen wurden jeweils zwei Aufrufziele bestimmt. In der Methode `A.i` wurde die Methode `m` aufgerufen. Der statische Typ des Empfängerobjektes ist `X`. Von CHA wird der bestimmte Typ selbst und alle seine Untertypen nach Implementierungen der aufgerufenen Methode durchsucht. Alle dabei gefundenen Methoden werden als potenzielle Aufrufziele betrachtet. Für den Aufruf der Methode `m` sind das `X.m` und `Y.m`. Genauso wurde in der Methode `A.r` vorgegangen.

Die Methode `A.p` enthält einen Aufruf von Methode `c` der Schnittstelle `S`. Es wurde ähnlich vorgegangen. Es wurden alle Klassen identifiziert, die diese Schnittstelle implementieren, und die Methoden als Aufrufziele betrachtet (`T.c` und `U.c`).

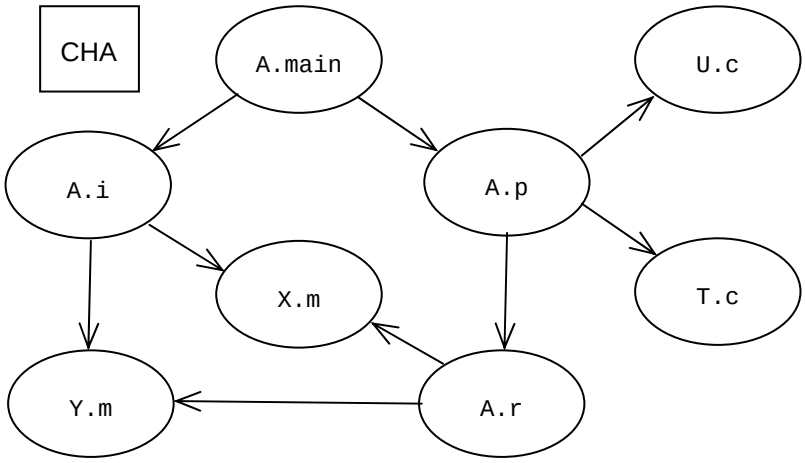

Bild 2.5: Aufrufgraph, der mit CHA für das Beispielprogramm erzeugt wurde.

2.4.3 Rapid Type Analysis

Die Rapid Type Analysis (RTA) erweitert die CHA, indem zusätzlich eine Hilfsmenge gebildet wird. In dieser Menge werden Typen gespeichert, von denen in der Anwendung Objekte erzeugt wurden. Diese Menge dient als eine zusätzliche Einschränkung der potenziellen Aufrufziele. Hintergrund dieses Ansatzes ist die Beobachtung, dass die Instanzmethoden nur auf den erzeugten Objekten aufgerufen werden können.

Auf der Bytecodeebene werden drei Instruktionen für die Erzeugung von Objekten unterschieden. Mit dem `new`-Befehl werden Klassenexemplare erzeugt. Die ein- bzw. mehrdimensionalen Objekte werden mit den Befehlen `anewarray` bzw. `multianewarray` angelegt. Die Erzeugungsstellen können im Bytecode für jede Methode identifiziert werden.

Man geht wie bei der CHA vor. Die Menge der Klassen mit potentiellen Aufrufzielen werden mit Hilfe der Klassenhierarchie eingeschränkt. Zusätzlich wird eine Schnittmenge zwischen der im letzten Schritt resultierenden Menge und der Menge der Klassen, von denen in der Anwendung Objekte erzeugt wurden, gebildet. In den Klassen dieser Schnittmenge werden Implementierungen aufgerufener Methode identifiziert.

Dadurch kann die Menge der Klassen, die potentiellen Aufrufziele enthalten können, reduziert werden. Es werden Methoden der Klassen ausgeschlossen, von denen keine Exemplare erzeugt wurden. Auf dem Bild 2.6 ist ein Aufrufgraph dargestellt, der mit RTA für das Beispielprogramm erzeugt wurde. Für die Aufrufstelle in der Methode `A.p`, an der die Methode `c` aufgerufen wird, wurde das eindeutige Aufrufziel bestimmt. Es wurde festgestellt, dass in der Anwendung der Objekt von Typ `U` nicht erzeugt wurde. Folglich hat man die Methode `U.s` aus der Menge der potentiellen Aufrufziele ausgeschlossen.

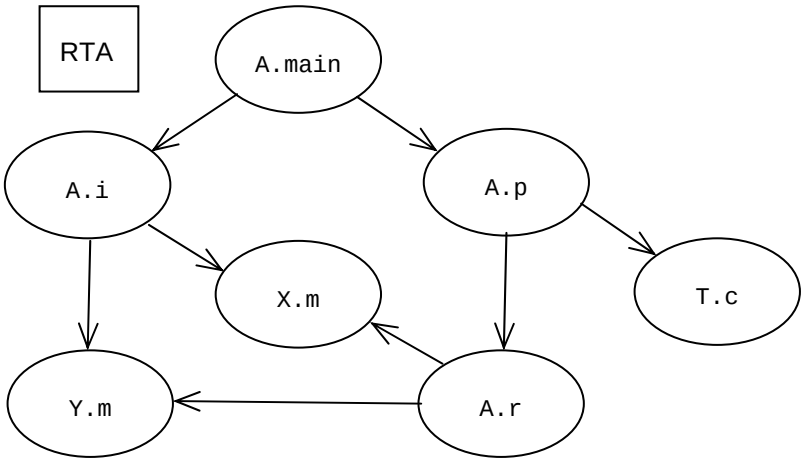

Bild 2.6: Der mit RTA für Beispielprogramm erzeugte Aufrufgraph.

2.4.4 Field Type Analysis

Die Field Type Analysis (FTA) verwendet beim Erzeugen des Aufrufgraphen prinzipiell die gleichen Informationen wie die RTA. Allerdings werden diese in ihrem lokalen Kontext verwendet und nicht im globalen wie in der RTA. Die Field Type Analysis benutzt beim Erzeugen des Aufrufgraphen eine Typenmenge pro Klasse und eine Typenmenge pro Methode. In der Menge pro Klasse werden Typen gespeichert, die von den Feldern dieser Klasse durch schreibende Zugriffe aufgenommen werden können. Die schreibenden Feldzugriffe repräsentieren den Datenfluss von der Methode in das Feld und somit die Klasse, die dieses Feld enthält. In der Menge pro Methode werden Typen gespeichert, die durch Erzeugungsstellen, Eingabeparameter, Rückgabewerte und lesende Feldzugriffe in der Methode bekannt sein könnten. Die Eingabe- und Rückgabeparameter repräsentieren den Datenfluss zwischen den Methoden. Die lesenden Feldzugriffe repräsentieren den Datenfluss von den Feldern in die Methode.

Zusätzlich zu den Informationen, die in der RTA bestimmt wurden, sollen noch die lesenden und schreibenden Zugriffe auf Felder der Klassen identifiziert werden. Java bietet vier Bytecode-Instruktion an: `getfield`, `getstatic`, `putfield` und `putstatic`.

Das Zusammenspiel dieser zwei Arten von Typmengen untereinander wird anhand des Bildes 2.2 erläutert. Auf dem Bild sind drei Typmengen dargestellt: Die Typmenge S_A für eine Klasse A und zwei Typmengen S_{m_1} und S_{m_2} für Methoden m_1 und m_2. Der gestrichelte Pfeil zwischen den Mengen S_{m_1} und S_{m_2} bedeutet, dass die Methode m_2 von der Methode m_1 aufgerufen wird. Die nicht gestrichelten Pfeile zeigen den Datenfluss zwischen den jeweiligen Programmelementen: zwischen der Klasse A und Methode m_1 mittels lesenden und schreibenden Zugriffe auf Felder dieser Klasse sowie zwischen Methoden m_1 und m_2 mittels Eingabe- und Rückgabeparameter.

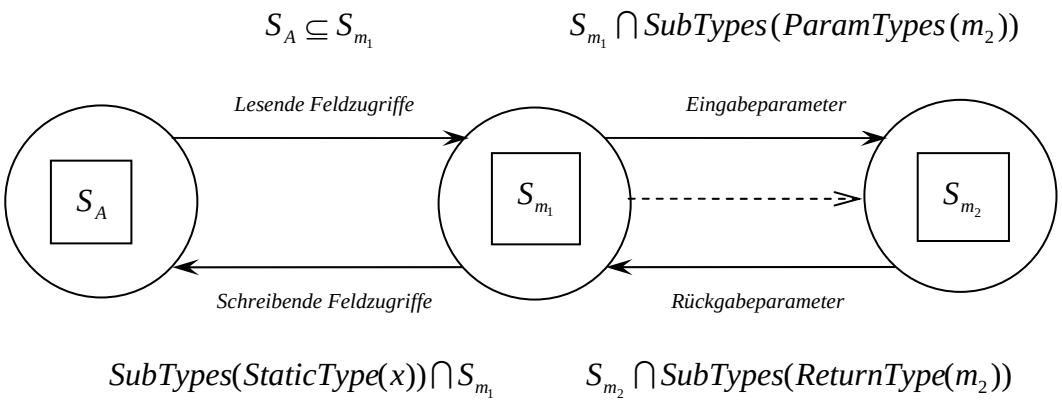

Bild 2.7: Der Informationsfluss zwischen den verschiedenen Typmengen

Die Typenmenge S_A einer Klasse wird mit Typen gefüllt, die den Feldern dieser Klasse innerhalb einer Methode zugewiesen werden könnten. An der Zugriffsstelle werden statische Typ des Feldes $StaticType(x)$ und alle seine Untertypen $SubTypes(StaticType(x))$ bestimmt. Als Nächstes wird eine Schnittmenge zwischen den im vorherigen Schritt ermittelten Typen und in der aktuellen Methode bekannten Typen bestimmt: $SubTypes(StaticType(x)) \cap S_{m_1}$. Diese Schnittmenge wird disjunkt mit der Menge S_A vereinigt.

Die Typenmenge einer Methode wird an erster Stelle mit potenziellen `this`-Referenzen und mit Typen, von denen in dieser Methode Objekte erzeugt worden sind, gefüllt. Danach werden lesende Feldzugriffe betrachtet. Bei einem lesenden Feldzugriff innerhalb einer Methode m_1 werden alle in der Menge S_A bekannten Typen der Menge S_{m_1} hinzugefügt: $S_A \subseteq S_{m_1}$.

Außerdem findet der Datenaustausch mit anderen Methoden statt. In Richtung der aufgerufenen Methode m_2 werden für jeden Parameter der statische Typ $ParamTypes(m_2)$ mit allen seinen Untertypen $SubTypes(ParamTypes(m_2))$ ermittelt und eine Schnittmenge mit den in der aufrufenden Methode m_1 bekannten Typen gebildet: $S_{m_1} \cap SubTypes(ParamTypes(m_2))$. In Richtung der aufrufenden Methode m_1 werden der statische Typ des Rückgabeparameters $ReturnType(m_2)$ mit allen seinen Untertypen $SubTypes(ReturnType(m_2))$ ermittelt und eine Schnittmenge mit den in der aufgerufenen Methode m_2 bekannten Typen gebildet: $S_{m_2} \cap SubTypes(ReturnType(m_2))$.

Dadurch wird eine lokale Sicht auf die Methoden der Anwendung geschaffen. Es erfolgt ein Ausschluss der Typen, die zwar im Programm aktiv sind, aber lokal in einer Methode keine Auswirkung haben. Auf dem Bild ist ein Aufrufgraph dargestellt, der mit FTA für das betrachtete Programm erzeugt wurde.

In der Methode `A.i` wird schreibend und lesend auf das Feld `A.x` zugegriffen. Deswegen findet Informationensaustausch zwischen entsprechenden Typmengen statt: $S_{A.i}$ und S_A. Die Menge S_A enthält die this-Referenz und Typen der in der Methode `A.i` erzeugten Objekte: $S_A = \{A, S, T, X\}$. Bei einem Schreibzugriff auf das Feld `A.x` fließt Information aus der Typmenge $S_{A.i}$ in die Menge S_A: $S_A = SubTypes(StaticType(x)) \cap S_{A.i} = \{X\}$. Bei einem lesenden Feldzugriff fließt Information zurück: $S_A \subseteq S_{A.i}$. Aber dadurch kommen keine neuen Typen in die Typmenge $S_{A.i}$ hinzu. Auf andere Weise wird die Menge $S_{A.i}$ mit neuen Typen erweitert. Also bleibt der Typ Y für die Methode `A.i` nicht bekannt. Deswegen wurde die Methode `Y.m` als potentielles Aufrufziel nicht betrachtet.

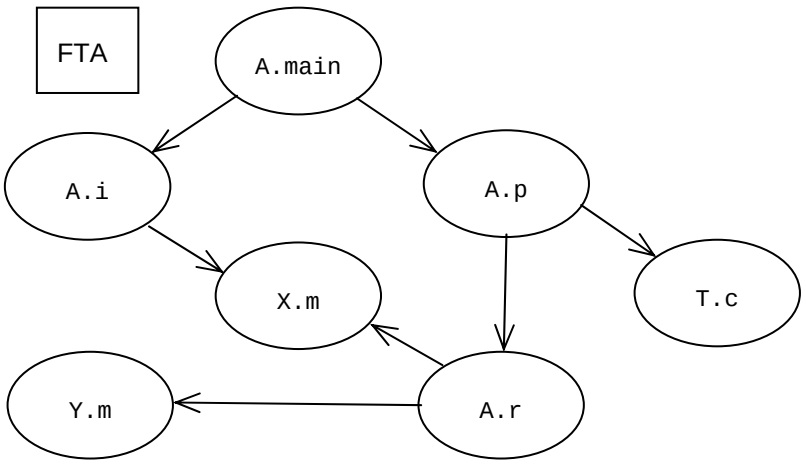

Bild 2.8: Aufrufgraph, der mit FTA für Beispielprogramm erzeugt wurde.

2.4.5 DefUse-Analyse

Die DefUse-Analyse beruht auf dem Konzept der Datenflussanalyse. Dabei werden so genannte DefUse-Ketten mittels eines intraprozeduralen Kontrollflussgraphen erzeugt. Der intraprozedurale Kontrollflussgraph ist ein gerichteter Graph zur Repräsentation von Kontrollstrukturen innerhalb einer Methode. Die Idee der DefUse-Analyse ist die innerhalb des Kontrollflussgraphs lesenden bzw. schreibenden Zugriffe auf eine Variable zu identifizieren. Die lesenden Zugriffe werden als Verwendungsstellen klassifiziert und die schreibenden Zugriffe als Definitionsstellen. Zu jeder Verwendungsstelle werden alle Definitionsstellen ermittelt. Somit wird eine Menge der Klassen gebildet, die an der Aufrufstelle aufgenommen werden können.

Der Aufruf einer Methode ist als eine Verwendungsstelle zu interpretieren. Die Methoden werden von der DefUse-Analyse nach Aufrufstellen bzw. Verwendungsstellen durchsucht. Für jede Verwendungsstelle wird eine DefUse-Kette bestimmt. An den Definitionsstellen können Typen der an den Ausdruck zugewiesenen Werten

identifiziert werden. In diesen Typen wird weiterhin nach Aufrufzielen gesucht. Wenn die Methode in dieser Klasse geerbt wurde, wird in der Oberklasse nach der implementierten Methode gesucht. Die auf diese Weise identifizierten Methoden werden der Menge der erreichbaren Methoden hinzugefügt.

Bei der Verwendung dieses Verfahrens werden die Klassen ausgeschlossen, von denen zwar im Programm Objekte erzeugt werden, jedoch nicht im Kontext der analysierten Aufrufstelle. Auf dem Bild 2.9 ist ein Aufrufgraph dargestellt, der mit DafUse-Analyse für das Beispielprogramm erzeugt wurde. In der Methode A.r wurde festgestellt, dass vor dem Aufruf der Methode `m` dem Empfängerobjekt ein Wert vom Typ `Y` zugewiesen wurde. Dadurch wurde die Methode `X.c` aus der Menge der potenziellen Aufrufziele ausgeschlossen.

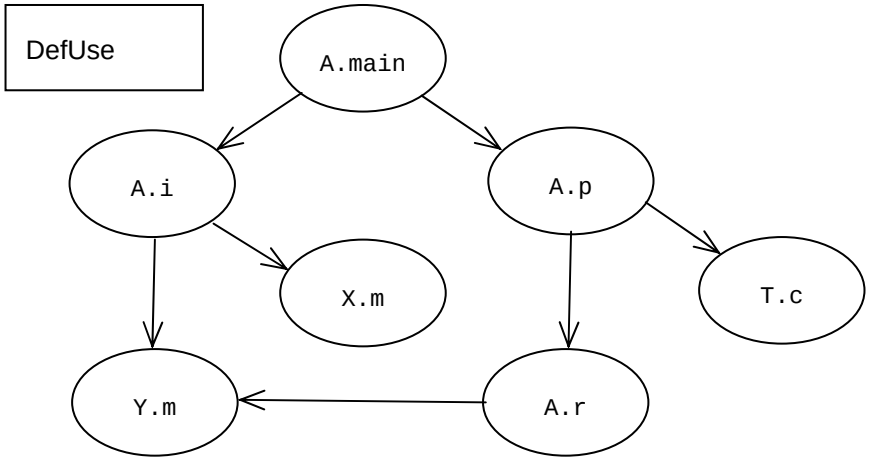

Bild 2.9: Der mit DefUse-Analyse für Beispielprogramm erzeugte Aufrufgraph.

2.4.6 Kombinierte Analyseverfahren

Es gibt in der Literatur Vorschläge, die Analyseverfahren miteinander zu kombinieren [Wessels]. Dabei wird die Initiallösung mit einfachen Verfahren bestimmt und anschließend mit aufwändigeren Analyseverfahren verfeinert. Die Auswahl der Aufrufrelationen für die weitere Verfeinerung wird nach bestimmten Auswahlkriterien durchgeführt. Die Initiallösung kann z. B. mit der RTA bestimmt und dann mit der FTA bzw. DefUse verbessert werden. Der Vorteil der kombinierten Verfahren ist, „den Aufrufgraph genauer als das Initialverfahren und schneller als das verfeinernde Verfahren zu bestimmen“ [Wessels]. Dadurch reduziert sich der Laufzeitaufwand der Analysen. Das aufwändige Verfahren wird nicht auf ganze zu analysierende Anwendung angewendet, sondern gezielt an den Stellen, wo z. B. die dynamische Bindung aufgelöst werden soll.

3 Konzeption

In diesem Kapitel wird das Verfahren zur Bestimmung des Speicherbedarfs für den Java-Card-Aufrufkeller mittels spezialisierter Aufrufgraphanalysen vorgestellt. Das Verfahren basiert auf statischen Analyseverfahren zur Erzeugung der Aufrufgraphen für eine Anwendung. Dabei steht die Auflösung der dynamischen Methodenbindung im Vordergrund. Als Erstes wird im Kapitel 3.1 die Funktionsweise des Verfahrens als Ganzes vorgestellt. In den nachfolgenden Kapiteln wird auf die Details der einzelnen Phasen des Verfahrens eingegangen. Im Kapitel 3.2 erfolgt die Beschreibung von spezifischen Aspekten der Erzeugung eines Aufrufgraphen. Im Kapitel 3.3 werden notwendige Bedingungen für die Bestimmung der Größe des Aufrufkellers einer Java-Card-Anwendung auf der Grundlage des erzeugten Aufrufgraphen diskutiert. Im Kapitel 3.4 wird das Konzept zur Ermittlung des Fehlers bei der Bestimmung der Größe des Aufrufkellers vorgestellt und abschließend im Kapitel 3.5 die Rolle der Eintrittspunkte bei der Bestimmung der maximalen Belegung des Aufrufkellers einer Java-Card-Anwendung mit Hilfe der Aufrufgraphen diskutiert.

3.1 Das Verfahren

Das von mir erarbeitete Verfahren zur Bestimmung der maximalen Größe des Aufrufkellers einer Java-Card-Anwendung gliedert sich in zwei voneinander unabhängige Phasen, wobei das Ergebnis der ersten Phase als Eingabe für die zweite Phase dient. Auf dem Bild ist das gesamte Verfahren als Diagramm dargestellt.

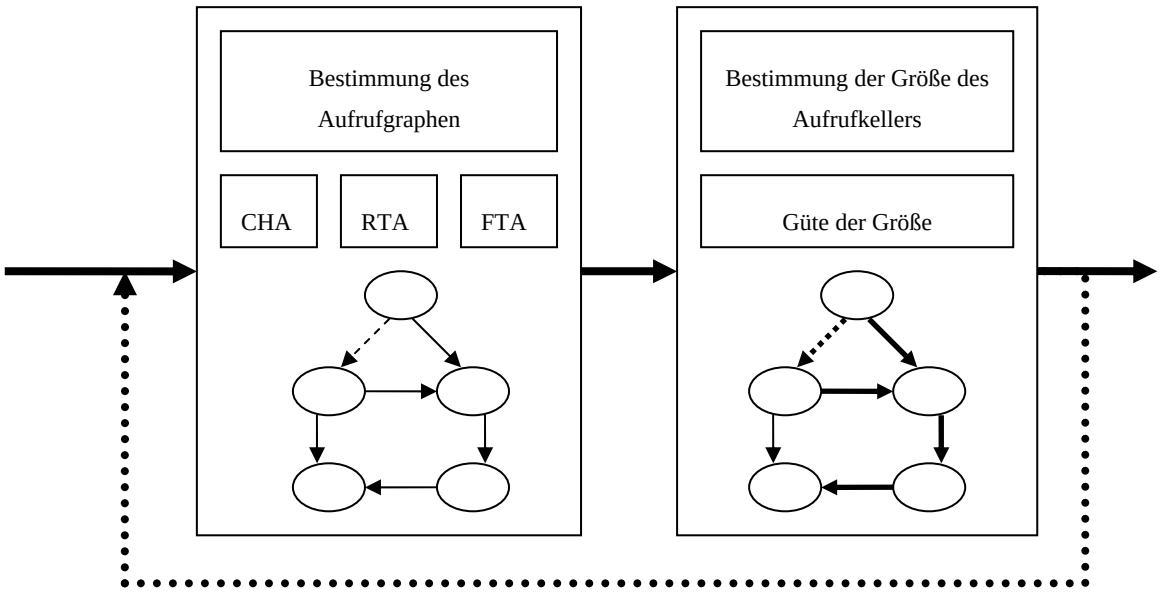

Bild 3.1: Schematische Darstellung des Verfahrens zur Bestimmung der maximalen Größe des Aufrufkellers einer Java-Card-Anwendung mit Hilfe statischer Aufrufgraphanalyseverfahren

In der ersten Phase wird für die gegebene Java-Card-Anwendung ein Aufrufgraph erzeugt. Dafür werden statische Aufrufgraphanalyseverfahren verwendet. Einige davon sind im Kapitel 2.3 vorgestellt worden. Der kritische Punkt bei der Erzeugung eines Aufrufgraphen für eine Java-Card-Anwendung mittels des statischen Aufrufgraph-analyseverfahrens ist die dynamische Methodenbindung. Um keine falschen Ergebnisse zu liefern, wird von den Analyseverfahren konservativ abgeschätzt, indem mehrere Methoden als potenzielle Aufrufziele in den Aufrufgraph aufgenommen werden. Dadurch nimmt die Genauigkeit des Aufrufgraphen ab. Aus diesem Grund wird eine Klassifizierung der Kanten eingeführt. Das Ergebnis dieser Berechnungsphase ist ein gerichteter Graph.

In der zweiten Phase wird auf der Grundlage des erzeugten Aufrufgraphen unter der Voraussetzung, dass der Graph azyklisch ist, die maximale Größe des Aufrufkellers berechnet. Auf Grund der konservativen Abschätzung der statischen Analyseverfahren können im Aufrufgraph Kanten entstehen, die unter Verwendung der aufwändigeren Verfahren aus dem Aufrufgraph entfernt werden könnten. Damit entsteht die Problematik, dass die Kellertiefe auf der Grundlage des bestimmten Aufrufgraphen ungenau berechnet werden könnte. Um beurteilen zu können, wie präzise die maximale Größe des Aufrufkellers bestimmt worden ist, werden zusätzlich obere und untere Schranken bezüglich der mit statischen Analyseverfahren erzeugten Aufrufgraphen ermittelt.

Das Zerlegen des Verfahrens in zwei Teile trägt zur Flexibilität in dem Sinne bei, dass die Erzeugung des Aufrufgraphen unabhängig von der Bestimmung der Kellertiefe ist und umgekehrt. Dabei können unterschiedliche Verfahren zur Erzeugung des Aufrufgraphen eingesetzt werden und für jedes verwendete Verfahren kann die Kellertiefe bestimmt werden. Ebenfalls ist es möglich, unterschiedliche Implementierungen der Algorithmen zur Berechnung der maximalen Größe des Aufrufkellers zu verwenden.

Außerdem ist die Rückkopplung dieser Phasenkette möglich. Das Ergebnis der zweiten Berechnungsphase ist die Eingabe für die erste Phase. Das ist sinnvoll, um die Kellertiefe in mehreren Iterationen auf der Grundlage der von den verschiedenen Analyseverfahren bestimmten Aufrufgraphen zu berechnen. Dieser Fall tritt auf, wenn z. B. im Graph Zyklen entdeckt wurden oder die Kellergröße hinsichtlich ihrer Güte nicht ganz exakt bestimmt wurde. Bei solchen Ergebnissen kann man dann versuchen, den Aufrufgraph mit aufwändigeren Analyseverfahren gezielt zu verfeinern.

3.2 Erzeugung der Aufrufgraphen

Zur Bestimmung der Aufrufgraphen werden in dieser Arbeit statische Aufrufgraph-analyseverfahren eingesetzt. Das Ziel dieser Analyseverfahren ist es, für eine gegebene Java-Card-Anwendung zur Übersetzungszeit einen Aufrufgraph zu erzeugen, indem für

jede Aufrufstelle ein Aufrufziel ermittelt wird. Dabei steht der Aspekt der dynamischen Methodenbindung im Vordergrund.

Die dynamisch gebundenen Methoden werden erst zur Laufzeit eindeutig aufgelöst. Um keine falschen Ergebnisse zu liefern, machen statische Analyseverfahren eine konservative Abschätzung, indem alle Methoden, die an der Aufrufstelle aufgerufen werden könnten, als potentielle Aufrufziele in den Aufrufgraph aufgenommen werden. Infolgedessen können von den statischen Analyseverfahren einer Aufrufstelle mehrere Aufrufziele zugeordnet werden. Um das Aufrufziel möglichst genau zu bestimmen, versuchen die Analyseverfahren die Menge der potenziellen Aufrufziele auf verschiede Art und Weise einzuschränken. Dadurch unterscheiden sich die Verfahren in der Genauigkeit, mit der die Aufrufgraphen erzeugt werden. Das Verwenden von aufwändigeren Analyseverfahren führt nicht unbedingt dazu, dass die dynamisch gebundenen Methodenaufrufe eindeutig aufgelöst werden. Anhand statischer Analyseverfahren ist es schwer zu beurteilen, ob es sich um eine Stelle handelt, an der ein nicht ausreichend präzises Analyseverfahren verwendet wurde oder ob es sich um einen Fall handelt, in dem mehrere Methoden während der Programmausführung tatsächlich aufgerufen werden.

3.2.1 Stabile und instabile Kanten

Auf Grund dynamischer Methodenbindung können von statischen Aufrufgraphanalyse-verfahren einer dynamisch gebundenen Aufrufstelle mehrere Aufrufziele zugeordnet werden. Dadurch entstehen im Aufrufgraph Kanten, die bei der Verwendung von aufwändigeren Analyseverfahren aus dem Aufrufgraph eventuell entfernt werden können. Angesichts dieses Aspektes scheinen sich solche Kanten in einem instabilen Zustand zu befinden: Sie sind zwar im Aufrufgraph vorhanden, aber könnten eventuell mit Einsatz eines besseren statischen Analyseverfahrens entfernt werden. Im weiteren Verlauf der Arbeit werde ich solche Kanten als instabile Kanten bezeichnen.

Es gibt auch Aufrufstellen, für die ein eindeutiges Aufrufziel von den Aufrufgraph-analyseverfahren bestimmt wird. Dazu gehören statisch gebundene und mit einem Aufrufziel dynamisch gebundene Aufrufstellen. Dadurch entstehen im Aufrufgraph Kanten, die auch unter Verwendung von aufwändigeren Analyseverfahren nicht mehr aus dem Aufrufgraph entfernt werden. Dementsprechend bezeichne ich solche Kanten als stabile Kanten.

Unter dem Gesichtspunkt, dass eine Aufrufinstruktion im Programm als Ursache einer Kante im Aufrufgraph betrachtet wird, kann man die Klassifizierung der Kanten in stabile und instabile folgendermaßen beschreiben: Eine stabile Kante wird von genau einer Instruktion verursacht. Darunter fallen sowohl durch statische als auch durch dynamisch gebundene Aufrufinstruktionen verursachte Kanten. Die statisch gebundenen Aufrufinstruktionen können nur ein eindeutiges Aufrufziel haben und somit nur eine Kante im Aufrufgraph verursachen. Für einige dynamisch gebundene

Aufrufinstruktionen wird durch die Analysen ebenfalls ein eindeutiges Aufrufziel bestimmt und dadurch entsteht im Aufrufgraph eine Kante. Siehe dazu das Bild 3.2.

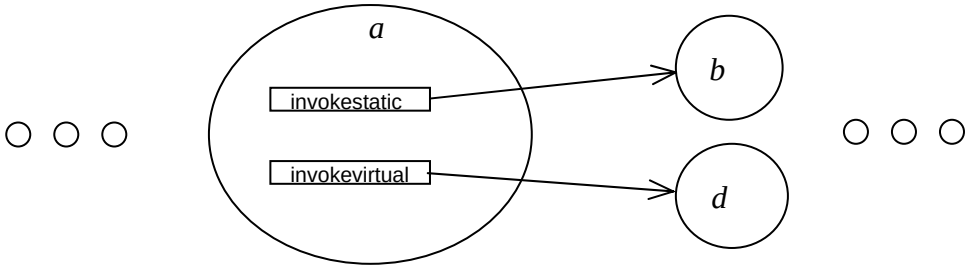

Bild 3.2: Stabile Kanten

Eine instabile Kante wird von einer Instruktion verursacht, die mehrere Aufrufziele hat. Die instabilen Kanten können nur durch dynamisch gebundene Aufrufinstruktionen entstehen, und zwar durch solche, für die mehrere Aufrufziele bestimmt wurden. Siehe dazu das Bild 3.3.

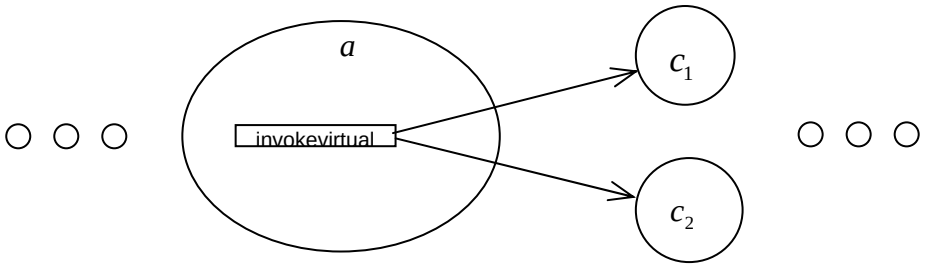

Bild 3.3: Instabile Kanten.

Auf dem Bild 3.4 ist ein Aufrufgraph dargestellt, der mit CHA für das Beispielprogramm aus dem Bild 2.4 erzeugt wurde. Die instabilen Kanten sind durch gestrichelte Linien veranschaulicht. Der Aufrufgraph enthält sechs instabile Kanten. Die wurden von drei dynamisch gebundenen Aufrufstellen mit jeweils zwei Aufrufzielen verursacht.

Bei der Verwendung eines aufwändigeren Analyseverfahrens kann eine instabile Kante entweder instabil bleiben, eine stabile werden oder aus dem Aufrufgraph verschwinden. Wird für eine Instruktion die Anzahl der Aufrufziele bis auf eins reduziert, so wird die Bedingung für die stabile Kante erfüllt.

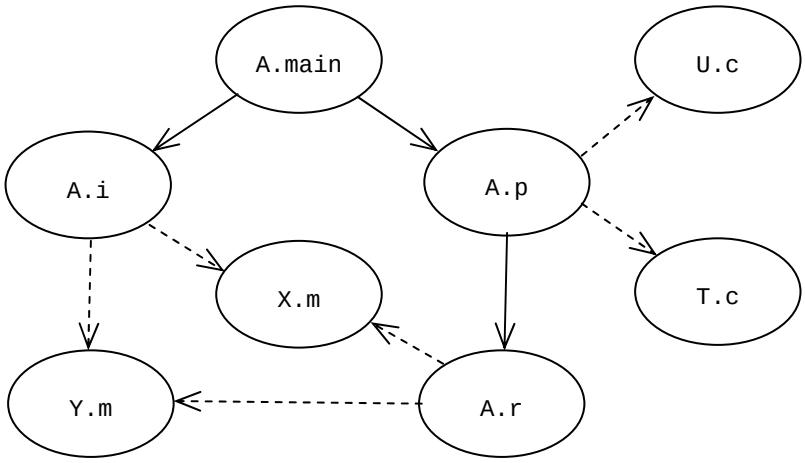

Bild 3.4: Der mit CHA erzeugte Aufrufgraph mit Klassifizierung der Kanten in stabile und instabile. Die instabilen Kanten sind mit gestrichelten Linien gekennzeichnet.

3.3 Bestimmung der maximalen Größe des Aufrufkellers

Die Bestimmung der Größe des Aufrufkellers erfolgt auf der Grundlage des für eine Java-Card-Anwendung erstellten Aufrufgraphen. Das Ergebnis des eingesetzten Analyseverfahrens ist ein gerichteter Graph. Nach der Definition im Hinblick auf den Aufrufgraphen aus Kapitel 2.2 repräsentiert ein Knoten im Graph eine Methode in der Anwendung. Ein Pfad vom Eintrittsknoten bis zum Blatt ist eine mögliche Aufrufkette während der Ausführung der zu analysierenden Anwendung. Für das auf dem Bild 2.3 vorgestellte Programm kann man zwei mögliche Aufrufketten nennen: (`A.main`, `A.i`, `X.m`) und (`A.main`, `A.p`, `A.r`, `Y.m`).

Bei der Abarbeitung einer Aufrufkette werden die Methodenschachteln der aufgerufenen Methoden auf den Aufrufkeller gelegt, bis die letzte Methode in der Aufrufkette abgearbeitet wird. Danach werden die Methodenschachteln mit der Abarbeitung der jeweiligen Methode von dem Aufrufkeller entfernt, bis die Ausführung einer neuen Aufrufkette gestartet wird. So entsteht eine Belegung des Aufrufkellers bei der Ausführung der jeweiligen Aufrufkette. Der Zustand des Aufrufkellers vor der Terminierung der letzten Methode einer Aufrufkette ist die Belegung des Aufrufkellers bei der Ausführung dieser Aufrufkette. Die letzte Methode in der Aufrufkette ist durch ein Blatt im Aufrufgraph repräsentiert. Auf dem Bild 3.4 sind Belegungen des Aufruf-kellers bei der Ausführung der beiden Aufrufketten schematisch dargestellt.

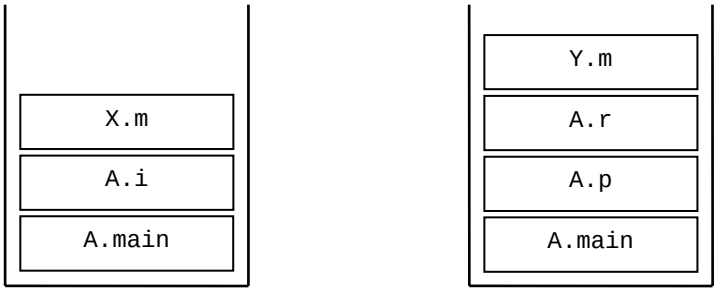

Bild 3.5: Zustand des Aufrufkellers vor der Abarbeitung der letzten Methode der beiden Aufrufketten (`A.main`, `A.i`, `X.m`) und (`A.main`, `A.p`, `A.r`, `Y.m`) aus dem Beispiel auf dem Bild 2.3.

Um eine mögliche Belegung des Aufrufkellers mit Hilfe des Aufrufgraphen zu berechnen, ist jedem Knoten die Größe der Schachtel der entsprechenden Methode zuzuweisen. Eine Methodenschachtel enthält lokale Variablen und den Operandenkeller. Eine Zelle des Operandenkellers und eine lokale Variable sind ein Wort groß. Die Summe der Gewichte der einzelnen Knoten eines Pfades vom Eintrittsknoten bis zum Blatt ist die Belegung des Aufrufkellers während der Ausführung der jeweiligen Aufrufkette. Die maximale Belegung des Aufrufkellers während der Ausführung einer Anwendung ist das maximale Gewicht eines der Pfade vom Eintrittsknoten bis zu allen Blättern im Aufrufgraph.

Auf dem Bild 3.5 ist ein Aufrufgraph mit Knotengewichten dargestellt. Die Knotengewichte repräsentieren die Größe der Methodenschachtel der entsprechenden Methode. Der Pfad (`A.main`, `A.p`, `A.r`, `X.m`) besitzt das maximale Gesamtgewicht und ist mit dickeren Linien gekennzeichnet. Die Ausführung der entsprechenden Aufrufkette würde die größte Belegung des Aufrufkellers für das betrachtete Programm verursachen.

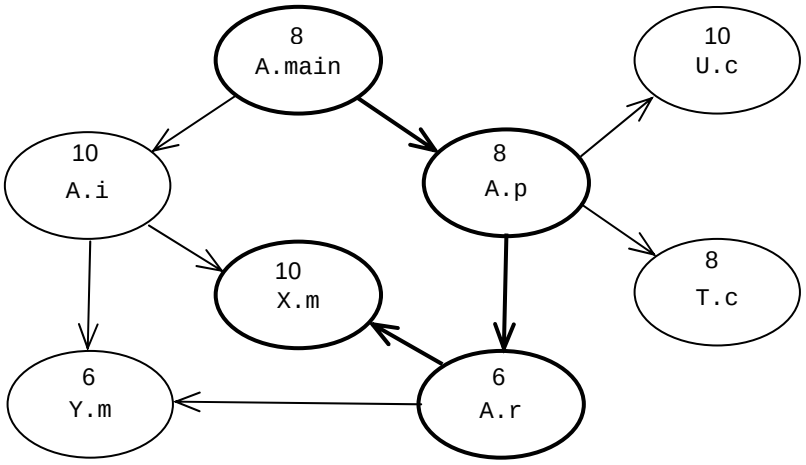

Bild 3.6: Aufrufgraph mit den Knotengewichten und dem maximal gewichteten Pfad.

Die Berechnung des maximalen Gewichtes aller Pfade von dem gegebenen Eintrittsknoten eines Aufrufgraphen lässt sich auf das Problem des längsten Pfades in einem gewichteten, gerichteten Graph hin abbilden. Das allgemeine Problem des längsten Pfades zwischen zwei Knoten in einem Graph ist NP-vollständig [Garey, Johnson]. Die NP-Vollständigkeit des Problems lässt sich mittels Reduktion vom Hamilton-Pfad zwischen zwei Knoten beweisen. Allerdings kann für gerichtete, gewichtete, azyklische Graphen das Problem des längsten Pfades in Polynomialzeit gelöst werden [Lawler].

Das Vorkommen von Zyklen im Aufrufgraph deutet auf die Verwendung der Rekursion in der zu analysierenden Anwendung hin. Die Rekursionstiefe ist mit statischen Mitteln schwer zu bestimmen. Dementsprechend ist es schwierig, die maximale Belegung des Aufrufkellers bei einem rekursiven Aufruf abzuschätzen. Deshalb soll für eine erfolgreiche Bestimmung der maximalen Belegung des Aufrufkellers für eine Java-Card-Anwendung mit Hilfe von Aufrufgraphen eine Bedingung erfüllt sein: Der Aufrufgraph soll azyklisch sein.

Das Vorkommen von rekursiven Aufrufen in den Java-Card-Anwendungen ist eher unwahrscheinlich. Es ist zwar nicht spezifiziert, aber es wird dringend empfohlen, die Rekursion in den Java-Card-Anwendungen nicht zu verwenden [Chen]. Es wird deshalb angenommen, dass keine Verwendung von rekursiven Aufrufen stattfindet.

Es gibt auch andere Ursachen für die Entstehung der Zyklen im Aufrufgraph. Die konservative Abschätzung der Analyseverfahren führt dazu, dass viele Aufrufziele potenziell in den Aufrufgraph aufgenommen werden. Dadurch können im Aufraufgraph Zyklen entstehen. Unter Verwendung von aufwändigeren Analyseverfahren kann die Anzahl der Aufrufstellen an einigen Stellen reduziert werden und dementsprechend können Zyklen wegfallen. Das einzige Mittel gegen im Aufrufgraph eventuell auftretende Zyklen ist die Verwendung der präziseren Aufrufgraphanalysen.

3.4 Güte der Größe des Aufrufkellers

Der Aufrufgraph wird wie im Kapitel 3.2 beschrieben bestimmt. Die Knoten des Aufrufgraphen besitzen Gewichte, die die Größe der Methodenschachteln der entsprechenden Methoden repräsentieren. Die Kanten sind in zwei Arten klassifiziert: stabile und instabile. Für die Bestimmung der Kellertiefe sind Pfade vom Eintrittspunkt bis zu den Blättern entscheidend. Genauso wie die Kanten können die Pfade ebenfalls in zwei Arten klassifiziert werden. Auf dem Bild 3.7 ist ein gewichteter Aufrufgraph, der mit CHA erzeugt wurde, dargestellt. Instabile Kanten sind mit gestrichelten Linien gekennzeichnet.

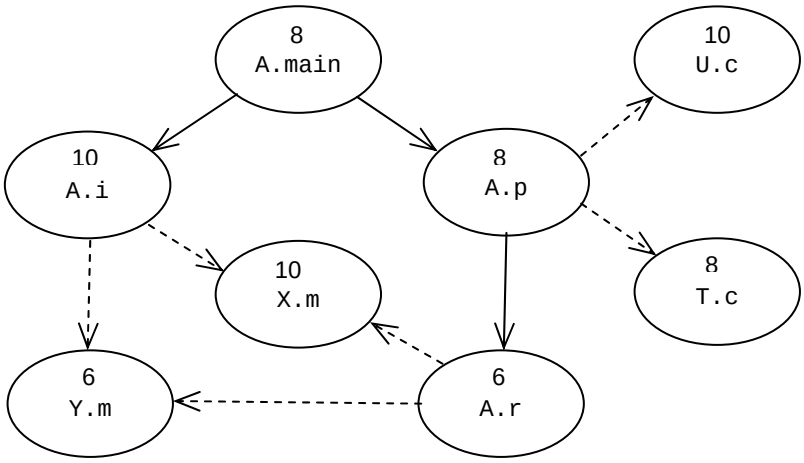

Bild 3.7: Gewichteter Aufrufgraph, der mit CHA erzeugt wurde, mit definierter Kantenklassifizierung.

Alle von dem Eintrittsknoten erreichbaren stabilen Kanten bilden einen stabilen Teilgraph. Dieser Teilgraph besitzt ebenfalls seine eigenen Blätter. Enthält der Aufrufgraph mindestens eine instabile Kante, so besitzt der stabile Teilgraph andere Blätter als der ursprüngliche Aufrufgraph. Die Pfade von dem Eintrittsknoten bis zu den Blättern des Teilgraphen sind stabile Pfade. Ein stabiler Pfad bleibt bei der Verwendung von aufwändigeren Analyseverfahren – mindestens so wie er war – im Aufrufgraph erhalten, denn keine der Kanten dieses Pfades kann dabei entfernt werden. Dieser Pfad kann höchstens beim Entstehen von weiteren stabilen Kanten verlängert, aber nicht verkürzt werden. Die Menge aller stabilen Pfade stellt damit einen Teilgraph dar, der entweder im gleichen Zustand bleibt oder nur größer werden kann. Das Gewicht des längsten stabilen Pfades stellt somit eine untere Schranke für die maximale Belegung des Aufrufkellers dar, die anhand des gegebenen Aufrufgraphen bestimmt werden kann.

Der längste stabile Pfad im Aufrufgraph auf dem Bild 3.7 ist (`A.main`, `A.p`, `A.r`). Die untere Schranke beträgt somit 22 Bytes. Nach der Verwendung von RTA wurde die Kante zwischen den Knoten `A.p` und `U.c` entfernt und die Kante zwischen den Knoten `A.p` und `T.c` ist stabil geworden (das Bild 3.8). Dadurch hat sich auch der längste stabile Pfad geändert: (`A.main`, `A.p`, `A.r`). Die untere Schranke hat sich vergrößert und ist 24 Bytes groß.

Enthält ein Pfad im Aufrufgraph auf dem Weg vom Eintrittsknoten bis zum Blatt mindestens eine instabile Kante, so wird dieser als instabil charakterisiert – mit der Begründung, dass eine der instabilen Kanten dieses Pfades bei der Verwendung eines besseren Analyseverfahrens aus dem Aufrufgraph eventuell entfernt wird und der entsprechende Pfad in dieser Form im Aufrufgraph nicht mehr existieren wird. Bestimmt man einen längsten Pfad auf dem gesamten Aufrufgraph, so kann dieser Pfad sowohl stabil als auch instabil sein. Ein längster instabiler Pfad stellt dabei eine konservative Abschätzung der maximalen Kellertiefe der zu analysierenden Anwendung dar. Es könnte bessere statische Analyseverfahren geben, die eine der instabilen Kanten dieses Pfades entfernen, damit auch der Pfad als solcher nicht mehr

im Aufrufgraph erhalten bleibt. Der längste aller Pfade stellt damit die obere Schranke für die mittels eines Aufrufgraphen bestimmbare maximale Belegung des Aufrufkellers dar. Der allgemein längste Pfad im Aufrufgraph auf dem Bild 3.7 ist (`A.main`, `A.p`, `A.r`, `X.m`). Dieser Pfad ist ein instabiler Pfad. Deshalb beträgt die obere Schranke 32 Bytes.

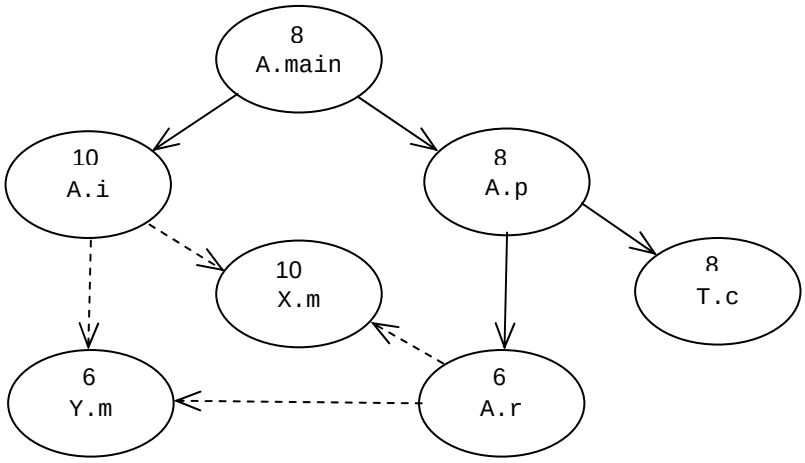

Bild 3.8: Gewichteter Aufrufgraph, der mit RTA erzeugt wurde, mit definierter Kantenklassifizierung.

Es ist offensichtlich, dass die beiden Schranken von dem Anteil der instabilen Kanten im Aufrufgraph abhängig sind und somit in einer Beziehung zueinander stehen. Auf Grund des Entfernens von instabilen Kanten können einige instabile Kanten stabil werden. Dadurch können einige stabile Pfade verlängert werden und als mögliche Folge kann sich die untere Schranke erhöhen. Möglicherweise werden im Gegensatz dazu mit dem Eliminieren von instabilen Kanten instabile Pfade kürzer. Infolgedessen könnte sich die obere Schranke verkleinern.

Auf dem Bild 3.9 ist ein Aufrufgraph dargestellt, der mit DefUse-Analyse erzeugt wurde. Der längste Pfad in diesem Aufrufgraph ist (`A.main`, `A.p`, `A.r`, `Y.m`). Die obere Schranke beträgt 28 Bytes und ist damit kleiner als für den Aufrufgraph, der mit CHA erzeugt wurde.

Im Idealfall konvergieren die beiden Schranken gegeneinander. Das Diagramm auf dem Bild 3.10 zeigt das ideale Verhalten der beiden Schranken bei der Verwendung von verschiedenen Aufrufgraphanalyseverfahren. Es wird angenommen, dass die Genauigkeit der Analyseverfahren in Richtung der x-Achse steigt. Nach der Verwendung von verschiedenen Aufrufgraphanalyseverfahren konvergieren die beiden Schranken gegeneinander.

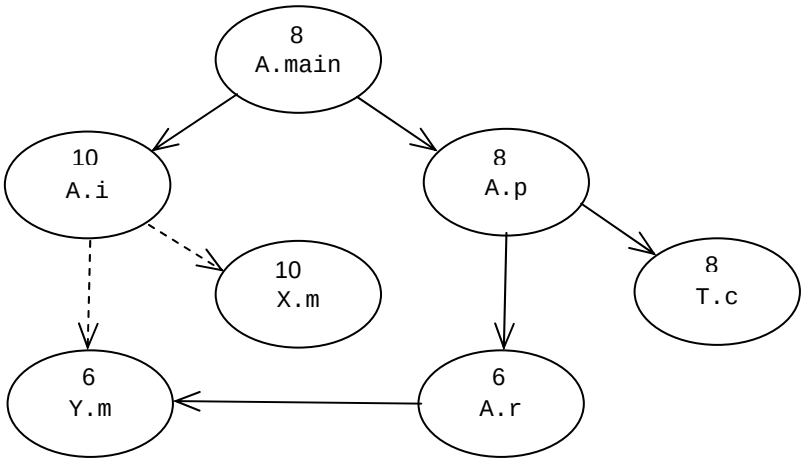

Bild 3.9: Gewichteter Aufrufgraph, der mit DefUse erzeugt wurde, mit definierter Kantenklassifizierung.

Die Differenz zwischen oberen und unteren Schranken eines Aufrufgraphen stellt ein Maß für die Güte der ermittelten Größe des Aufrufkellers dar. Sind die obere und untere Schranke verschieden, so kann man versuchen, den Aufrufgraph bzw. den längsten Pfad mit aufwändigeren Analyseverfahren zu verbessern. Dazu lassen sich manche Verfahren gezielt auf ausgewählte Aufrufstellen anwenden, so wie z. B. die DefUse-Analyse.

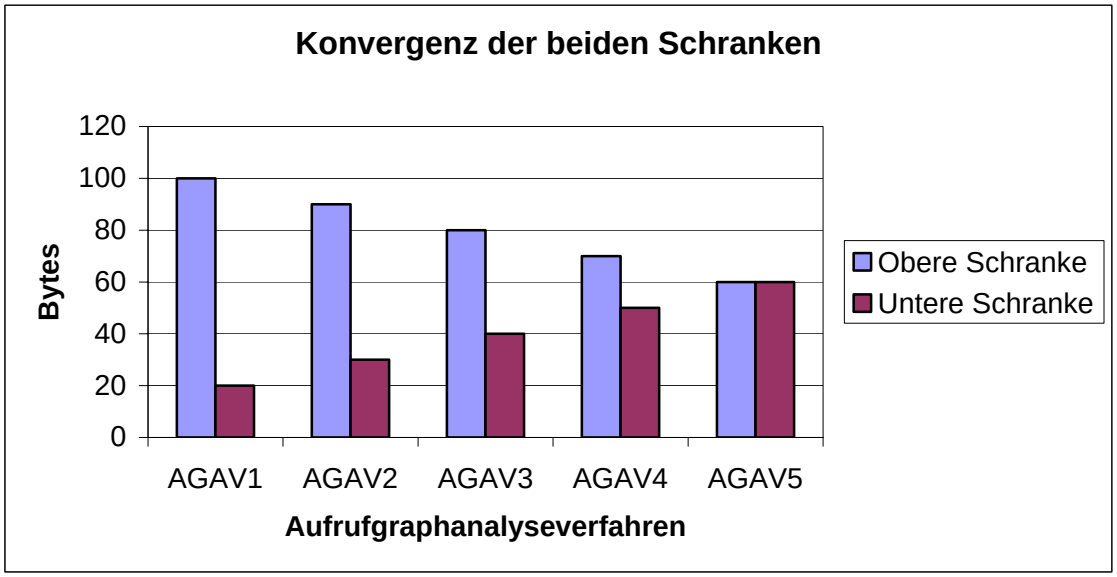

Bild 3.10: Das Diagramm zeigt den Idealfall, indem mit der Verwendung von verschiedenen Aufrufgraphanalyseverfahren die obere und untere Schranke gegeneinander konvergieren.

Wurden die obere und untere Schranke als gleich groß bestimmt, sind keine weiteren Verbesserungen an dem längsten Pfad des Aufrufgraphen mittels Aufrufgraphanalyse-verfahren möglich. Falls andere instabile Kanten im Aufrufgraph enthalten sind, haben weitere Verbesserungen an dem Aufrufgraph keine Auswirkung auf die obere Schranke, denn diese Kanten liegen nicht auf dem längsten Pfad.

Für den Aufrufgraph auf dem Bild 3.9 ist der allgemein längste Pfad gleich dem längsten stabilen Pfad. Deswegen sind die obere und die untere Schranken ebenfalls gleich, sodass damit keine weiteren Verbesserungen an dem Aufrufgraph mit Hilfe der aufwändigeren Analyseverfahren nötig sind, obwohl im Aufrufgraph noch instabile Kanten vorhanden sind. Die maximale Belegung des Aufrufkellers bei der Ausführung dieses Beispielprogramms entspricht in diesem Fall der oberen Schranke.

3.5 Eintrittspunkte

Wie bereits erwähnt wurde besitzt ein Java Card Applet vier reguläre Eintrittspunkte. Diese Methoden werden bei dessen Ausführung zu den unterschiedlichen Zeitpunkten von der Java-Card-Laufzeitumgebung aufgerufen. Die Java-Card-Laufzeitumgebung besitzt ebenfalls die Eintrittspunkte. Diese werden allerdings direkt von der virtuellen Java-Card-Maschine aufgerufen.

Bei der Bestimmung der maximalen Belegung des Aufrufkellers einer Java-Card-Anwendung anhand der Aufrufgraphen sollen die Eintrittspunkte im Hinblick auf zwei Aspekte berücksichtigt werden. Zum einen sind diese bei der Erzeugung von Aufrufgraphen und zum anderen bei der Bestimmung der maximalen Belegung des Aufrufkellers mit Hilfe der erzeugten Aufrufgraphen zu beachten.

3.5.1 Eintrittspunkte bei der Aufrufgrapherzeugung

Die Eintrittspunkte einer Java-Card-Anwendung werden zu den verschiedenen Zeitpunkten ihres Lebenszyklus von der Java-Card-Laufzeitumgebung aufgerufen, wobei in einer Eintrittsmethode auf Objekte zugegriffen werden kann, die in einer anderen Eintrittsmethode zu einem früheren Zeitpunkt initialisiert wurden.

Die vorgestellten Aufrufgraphanalyseverfahren setzen verschiedene Informationen ein, um die Aufrufgraphen für die gegebene Java-Card-Anwendung zu erzeugen. Die CHA z. B. verwendet die Klassenhierarchieinformationen. Die aufwändigeren Aufrufgraph-analyseverfahren benutzen zusätzliche Information, so wie z. B. Erzeugungsstellen von Objekten oder Datenflussinformationen.

Die Nichtbeachtung eines Eintrittspunktes kann dazu führen, dass bei der Bestimmung eines Aufrufziels für eine Aufrufstelle nicht genügend Informationen über das Empfängerobjekt zur Verfügung stehen und dadurch inkorrekte Aufrufgraphen erzeugt werden. Die RTA z. B. verwendet die Erzeugungsstellen, um die Menge der potenziellen Aufrufziele einzuschränken. Das Verfahren beginnt mit vorgegebenen Eintrittspunkten und analysiert alle erreichbaren Methoden. Ist aber eine der Erzeugungsstellen für die Analyse nicht erreichbar, wird die in der entsprechenden Klasse implementierte Methode nicht als potenzielles Aufrufziel betrachtet. Aus diesem Grund sollten die Abhängigkeiten zwischen den Eintrittspunkten bei der Erzeugung der Aufrufgraphen berücksichtigt werden. Allerdings führt die Betrachtung zu vieler Methoden als Eintrittspunkte, die eigentlich nicht als solche auftreten, dazu, dass

wiederum zu viele Informationen betrachtet werden und dadurch die Menge der Aufrufziele einer Aufrufstelle nicht verkleinert werden kann.

3.5.2 Eintrittspunkte bei der Bestimmung der Kellergröße

Bei der Bestimmung der maximalen Belegung des Aufrufkellers ist die Berücksichtigung der Eintrittspunkte ebenfalls wichtig. Wie bereits erwähnt gibt es Methoden der Java-Card-Laufzeitumgebung, die direkt von der virtuellen Java-Card-Maschine aufgerufen werden. Ein Java Card Applet auf der Karte besitzt ebenfalls die Eintrittspunkte, die dann von der Java-Card-Laufzeitumgebung aufgerufen werden.

Unter der Annahme, dass die Eintrittspunkte des Java Card Applets von den Eintrittspunkten der Java-Card-Laufzeitumgebung im erzeugten Aufrufgraph erreichbar sind, berechnet man für Letztere die längsten Pfade. Allerdings gibt es für den Fall, dass in der betrachteten Anwendung eine Aufrufkette existiert, in der zwei Eintrittspunkte mit einem nativen Methodenaufruf verbunden sind, im Aufrufgraph keinen direkten Weg von dem ersten Eintrittspunkt bis zum zweiten. Zum Zeitpunkt des Aufrufs der zweiten Eintrittsmethode von einer nativen Methode sind auf dem Aufrufkeller die Methodenschachteln der von dem ersten Eintrittsknoten aufgerufenen Methoden geladen. Deshalb darf man nicht davon ausgehen, dass der Aufruf der zweiten Eintrittsmethode mit dem leeren Aufrufkeller beginnt.

In dieser Arbeit werden Java-Card-Aufrufkeller analysiert und dabei statische Analyseverfahren zur Analyse von Java-Methoden verwendet. Die betrachteten Aufrufgraphanalyseverfahren sind nicht in der Lage, die nativen Methoden zu analysieren. Deshalb ist in diesem Fall keine direkte Berechnung der maximalen Kellertiefe mittels erzeugten Aufrufgraphen möglich.

Als mögliche Erweiterung des Verfahrens könnte man sich die Verwendung der Analyseverfahren vorstellen, die in der Lage sind, auch die nativen Methoden zu berücksichtigen. Dadurch werden die Aufrufketten der Anwendung noch präziser im Aufrufgraph dargestellt.

4 Realisierung

In diesem Abschnitt werden einige interessante Realisierungsaspekte des im Kapitel 3 vorgestellten Verfahrens zur Bestimmung der Größe des Aufrufkellers einer Java-Card-Anwendung erläutert. Die Umsetzung des erarbeiteten Verfahrens erfolgte unter Verwendung der Analyseumgebung PAULI. Im Abschnitt 4.1 wird eine kleine Einführung in die Umgebung gegeben. Einige Erweiterungen an den bestehenden Implementierungen der Aufrufgraphanalyseverfahren werden im Abschnitt 4.2 beschrieben. Darunter befinden sich Anpassungen der vorhandenen Analyseverfahren, um die Zuordnung der Aufrufziele zu den Aufrufstellen zu ermöglichen, und die Erstellung einer einheitlichen Schnittstelle für den Zugriff auf Ergebnisse der Aufrufgraphanalysen. Im Abschnitt 4.3 werden einige Aspekte der Bestimmung der Kellertiefe auf der Grundlage des erzeugten Aufrufgraphen erläutert.

4.1 Die Analyseumgebung PAULI

PAULI ist ein Framework für statische Analysen und die Manipulation von Java-Programmen. Das Framework wurde im Rahmen verschiedener Projekte in der AG Kastens der Universität Paderborn entwickelt und wird beständig erweitert. Auf dem Bild 4.1 ist eine schematische Darstellung der Struktur der Analyseumgebung zu sehen.

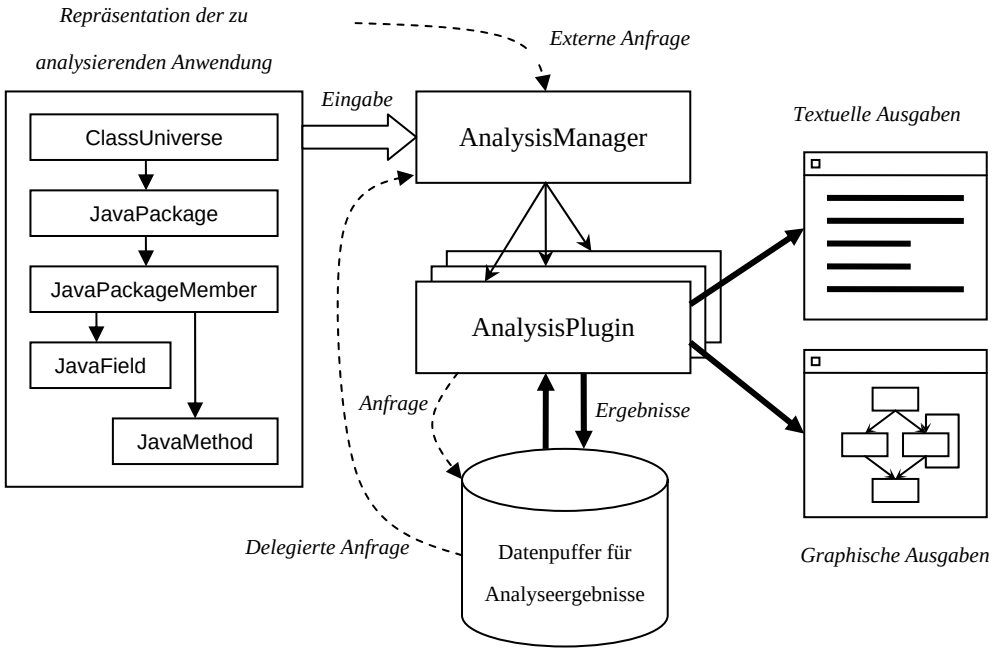

Bild 4.1: Schematische Darstellung der Struktur der Analyseumgebung PAULI

Die Analyseumgebung beinhaltet verschiedene Arten von Analysen, die in Form von Plugins in die Umgebung integriert sind, wobei einige Plugins für ihre Berechnungen die Ergebnisse von anderen Plugins verwenden.

Die zentrale Schnittstelle der Umgebung ist die Klasse `AnalysisManager`. Der `AnalysisManager` verwaltet die Erzeugung von Plugins, die Zugriffe der Client-Anwendung auf diese und die von den Plugins berechneten Ergebnisse. Eine Client-Anwendung ruft niemals direkt einen Plugin auf, sondern überlässt diese Aufgabe dem `AnalysisManager` und bekommt das berechnete Ergebnis zurück.

Die Analyseumgebung beinhaltet eine Unterstützung des Caching-Mechanismus. Die berechneten Ergebnisse eines Plugins können in einem Datenpuffer abgelegt werden. Bei der Anforderung eines Ergebnisses wird geprüft, ob dieses von dem entsprechenden Plugin in dem gegebenen Analysekontext bereits berechnet wurde. Außerdem gibt es die Möglichkeit, die berechneten Ergebnisse als Text oder grafisch als Graph oder Diagramm darzustellen.

Eine Menge von Java-Paketen bildet die Eingabe für die Umgebung. Alle für das zu analysierende Programm benötigten Bibliotheken sind einzugeben. Für gegebene Java-Pakete wird mittels BCEL-Bibliothek eine objektorientierte Repräsentation aufgebaut [Dahm]. Die BCEL-Bibliothek bietet für jedes Programmelement eine Java-Klasse – für eine Methode z. B. die Klasse `Method` und für ein Feld die Klasse `Field`.

Die Analyseumgebung besitzt ebenfalls Klassen zur Repräsentation von Programmelementen. Sie bilden sozusagen eine Hülle um die entsprechenden Klassen der BCEL-Bibliothek. An der Spitze der Repräsentation steht die Klasse `ClassUniverse`. Ein Exemplar dieser Klasse beinhaltet alle für die Analyse benötigten Java-Pakete. Um den Analysekontext zu verkleinern, kann man zusätzlich definieren, welche Pakete vom gesamten Analysekontext analysiert werden sollen. Die Beziehung zwischen den Klassen aus der Analyseumgebung zur Objektrepräsentation und den Klassen der BCEL-Bibliothek ist auf dem Bild 4.2 als UML-Klassendiagramm dargestellt.

Für die Integration eines neuen Plugins sind zwei Unterklassen zu definieren. Eine Unterklasse wird von der Klasse `AnalysisPlugin` abgeleitet. Diese Klasse definiert den eigentlichen Plugin. Hier wird der Analysealgorithmus implementiert. Die Oberklasse `AnalysisPlugin` enthält Basisimplementierungen für einige Methoden. Andere sollen in den Unterklassen implementiert werden.

Die zweite Unterklasse erweitert die Klasse `PluginResults`. Diese Klasse kapselt das mit entsprechendem Plugin berechnete Ergebnis und bietet eine Zugriffsschnittstelle darauf. Für jede Plugin-Klasse wird eine Ergebnisklasse definiert. Von jedem Plugin wird erwartet, dass nach der Berechnung ein Objekt entsprechender Ergebnisklasse erzeugt wird.

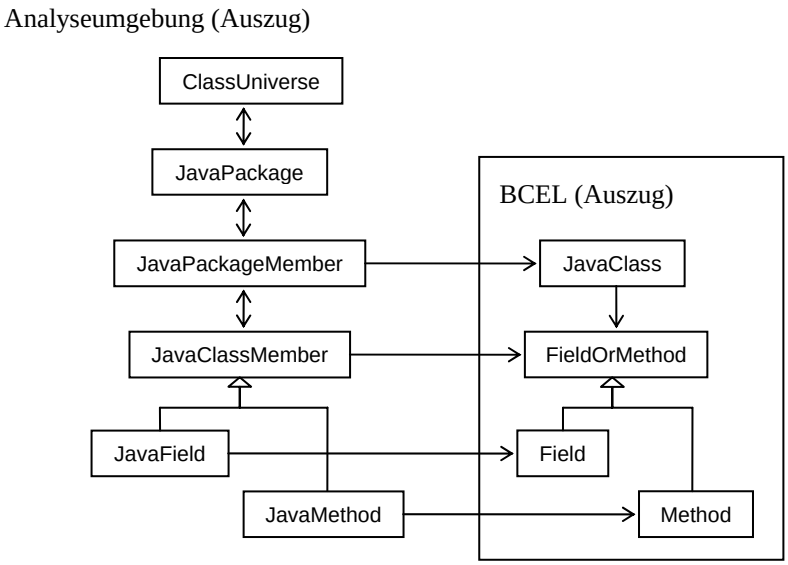

Bild 4.2: Klassen für objektorientierte Repräsentation der zu analysierenden Anwendungen als UML-Klassendiagramm

4.2 Erweiterungen der vorhandenen Aufrufgraphanalyseverfahren

Die Analyseumgebung beinhaltet einige Aufrufgraphanalyseverfahren. Das sind unter anderem CHA, RTA und FTA. Außerdem gibt es zwei kombinierte Analyseverfahren: RTA mit FTA und RTA mit DefUse. Die Verfahren sind im Kontext des Entfernens ungenutzter Programmelemente implementiert worden [Wessels]. Einige Erweiterungen waren nötig, um die vorhandenen Implementierungen für das im Kapitel 3 entworfenen Verfahren zur Bestimmung der Größe des Aufrufkellers einsetzen zu können.

4.2.1 Zuordnung der Aufrufziele zu den Aufrufstellen

Die in der Analyseumgebung implementierten Aufrufgraphanalyseverfahren basieren auf den Ergebnissen eines Plugins, der einen Aufrufgraph nach dem CHA ähnlichen Prinzip aufbaut. Dieser Aufrufgraph unterstützt eine Zuordnung von Aufrufzielen zu den aufrufenden Methoden. Die vorliegenden Implementierungen der Aufrufgraph-analyseverfahren versuchen auf verschiedene Art und Weise diesen Aufrufgraph zu verfeinern, indem die in der zu analysierenden Anwendung auftretenden dynamisch gebundenen Methodenaufrufe aufgelöst und entsprechende Kanten aus dem Aufrufgraph entfernt werden. Der verwendete Aufrufgraph entspricht in diesem Sinne nicht dem im Kapitel 3.2 beschriebenen Aufrufgraph, der nach einer Zuordnung von Aufrufzielen zu den Aufrufstellen verlangt.

Um den im Kapitel 3.2 beschriebenen Aufrufgraph zu realisieren und dabei von der inneren Implementierung und der verwendeten Datenstrukturen der vorhandenen Aufrufgraphanalyse-Plugins zu abstrahieren, habe ich eine Schnittstelle eingebaut.

Diese Schnittstelle besteht aus einer Klasse, die den Aufrufgraph repräsentiert und die Unterstützung für die Zuordnung von Aufrufzielen zu den Aufrufstellen bietet. Die Zuordnung wird mittels Annotationen an den Kanten realisiert. Beim Einfügen einer Kante zwischen zwei Knoten wird die verursachte Instruktion mit gespeichert.

4.2.2 Entkopplung der Berechnungsphasen

Das im Kapitel 3 vorgestellte Verfahren besteht aus zwei Phasen. Die Aufgabe der ersten Phase ist die Erzeugung des Aufrufgraphen. Es stehen verschiedene Aufrufgraph-analyseverfahren zur Verfügung. In der zweiten Phase wird die maximale Größe des Aufrufkellers der zu analysierenden Anwendung anhand des in der vorherigen Phase erzeugten Aufrufgraphen berechnet.

Um die Entkopplung der beiden Phasen zu gewährleisten, habe ich ebenfalls eine Schnittstelle eingebaut. Dadurch bleiben die Details der Aufrufgrapherzeugung für die zweite Phase verborgen. Außerdem wird damit ein einheitlicher Zugriff auf die Ergebnisse der ersten Phase ermöglicht, denn für die Erzeugung der Aufrufgraphen werden verschiedene Aufrufgraphanalyseverfahren verwendet.

Die Schnittstelle ist nach dem Prinzip des Aufbaus der Plugins in der Analyseumgebung konstruiert worden. Auf dem Bild 4.3 sind wichtige Komponenten der Schnittstelle als UML-Klassendiagramm dargestellt.

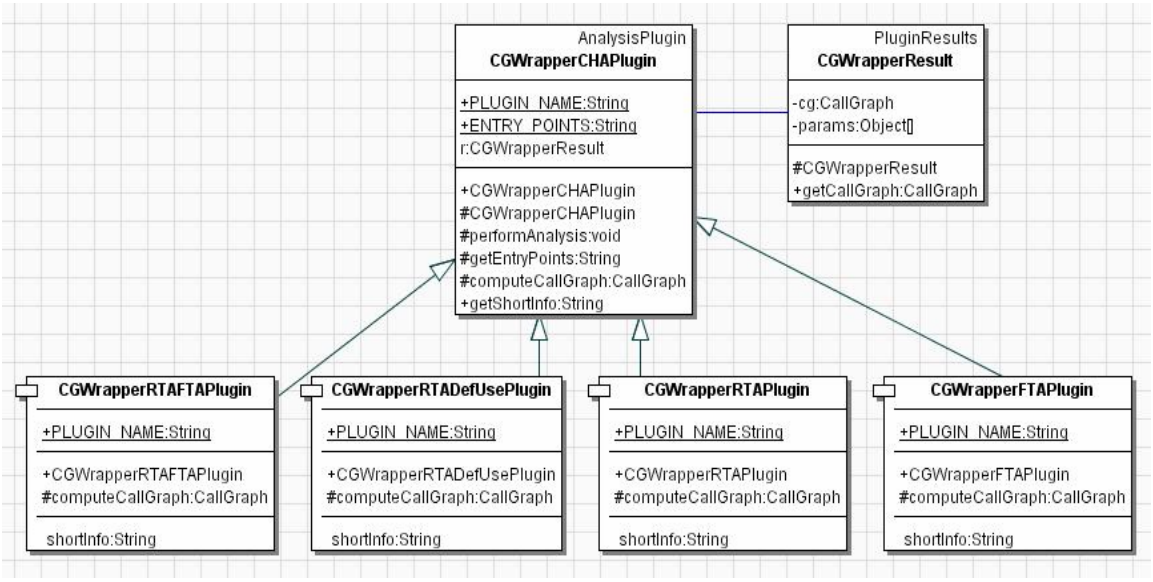

Bild 4.3: Ein Ausschnitt aus dem UML-Klassendiagram

Für jeden Aufrufgraphanalyse-Plugin wurde eine Wrapper-Klasse definiert. Diese Wrapper-Klassen kapseln in sich die Verarbeitung der Parameter, die Bestimmung des Aufrufgraphen und die Speicherung des Ergebnisobjektes für den entsprechenden Plugin. Für alle Wrapper-Klassen gibt es eine Ergebnisklasse (`CGWrapperResult`), die den erzeugten Aufrufgraph beinhaltet.

4.3 Bestimmung der Größe des Aufrufkellers

In diesem Kapitel wird auf die Integrationsaspekte des Verfahrens in die Analyseumgebung eingegangen. Außerdem erfolgt die Erläuterung wichtiger Randbedingungen bei der Bestimmung der Knotengewichte. Anschließend wird ein Algorithmus zur Berechnung der längsten Pfade vorgestellt.

4.3.1 Einbindung des Verfahrens in die Analyseumgebung

Das entworfene Verfahren wurde ebenfalls in die Analyseumgebung integriert. Hierdurch kann das erarbeitete Verfahren auf gleiche Weise wie die anderen Plugins der Analyseumgebung für die weiteren Analysen von Programmen verwendet werden. Nach dem im Kapitel 4.1 beschriebenen Muster der Integration von neuen Plugins in die Analyseumgebung habe ich zwei Klassen definiert. Auf dem Bild 4.4 ist ein UML-Klassendiagramm mit beiden Klassen dargestellt.

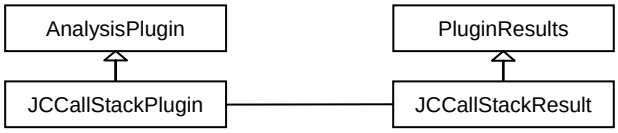

Bild 4.4: Integration des Verfahrens in die Analyseumgebung

Die Klasse `JCCallStackPlugin` wird von der Klasse `AnalysisPlugin` abgeleitet. Die Klasse `JCCallStackPlugin` implementiert den für die Bestimmung des längsten Pfades in einem gewichteten, gerichteten azyklischen Graph (DAG) benötigten Algorithmus. Der Algorithmus greift auf das Ergebnis eines der im Kapitel 4.2.2 beschriebenen Plugins zu, deren Aufgabe es ist, einen Aufrufgraph zu erzeugen.

Für den erzeugten Aufrufgraph sind die Gewichte der Knoten zu bestimmen. Allgemein gehorcht die Bestimmung der Gewichte dem im Kapitel 3.3 vorgestellten Verfahren. Allerdings gibt es einige Aspekte, die bei der Implementierung des Verfahrens berücksichtigt werden müssen. Auf diese gehe ich im nächsten Unterkapitel näher ein.

Als Nächstes werden zwei längste Pfade bestimmt. Der Erste ist der allgemein längste Pfad ausgehend von dem Eintrittsknoten. Der Zweite ist der längste Pfad für die Knoten, die über stabile Kanten von dem Eintrittsknoten erreichbar sind. Die beiden längsten Pfade repräsentieren die oberen und unteren Schranken. Der Algorithmus zur Berechung des längsten Pfades in einem gerichteten azyklischen Graph wird im Kapitel 4.3.3 vorgestellt.

Die Klasse `JCCallStackResult` stellt eine Schnittstelle zum Zugriff auf innere Eigenschaften der berechneten längsten Pfade zur Verfügung. Darunter sind die Zugriffe auf die von einem Eintrittspunkt erreichbaren Blätter.

Der Plugin wurde so entworfen, dass er mit Hilfe des Template-Method-Musters sich erweitern lässt. Wenn es z. B. die Knotengewichte anders bestimmt werden sollen, wird die entsprechende Methode in der Unterklasse überschrieben.

4.3.2 Bestimmung der Knotengewichte

Die verwendeten Analyseverfahren greifen auf Java Bytecode zu, der für einen Java-Interpretierer generiert wurde, deren Wort die Werte höchstens vom Typ Integer fassen kann. D. h. ein Wort ist 32 Bits groß. Eine CAP-Datei wird mit dem Konverter aus dem Java Bytecode der Klassendateien erzeugt. Der Bytecode der CAP-Datei wird für die Ausführung mit virtueller Java-Card-Maschine angepasst (optimiert). Wie im Kapitel 2.2.3 erwähnt wurde, verwendet virtuelle Java-Card-Maschine 16 Bit große Wörter.

Das hat zur Folge, dass bei der Verwendung des Datentyps Integer in einer Java-Card-Anwendung für solche Variablen zwei Wörter benötigt werden und nicht ein Wort wie bei der Ausführung mit der virtuellen Java-Maschine. Dadurch erhöht sich die Anzahl der Zellen auf dem Operandenkeller und in dem Array mit lokalen Variablen einer Methode. Folglich ist sie nicht mehr identisch mit der in Klassendateien berechneten Anzahl.

Wird in der Java-Card-Anwendung kein Integer-Datentyp verwendet, so ist die Anzahl der Zellen in beiden Dateiformaten gleich groß, denn die Variablen vom Typ Byte und Short werden ebenfalls in einem 32 Bits großen Wort untergebracht. Abgesehen von der Tatsache, dass die Parameter einer Methode in der CAP-Datei nicht im Array mit lokalen Variablen abgelegt werden, sondern in einer separaten Struktur.

In Zusammenhang mit meiner Implementierung bin ich davon ausgegangen, dass in den zu analysierenden Java-Card-Anwendungen kein Integer Datentyp verwendet wird. Somit wird die Größe einer Methodenschachtel mit folgender Formel berechnet:

$$M = (z + v) \cdot w \tag{4.1}$$

Wobei M Größe der Methodenschachtel, z Anzahl der Zellen des Operandenkellers, v Anzahl der Elemente in dem Array mit lokalen Variablen und w Größe des Wortes ist.

4.3.3 Algorithmus zur Berechnung des längsten Pfades

Wie bereits im Kapitel 3.3 erwähnt wurde, lässt sich das Problem des längsten Pfades für einen gerichteten azyklischen Graph in Polynomialzeit lösen. In [Lawler] wird der Einsatz des Bellman-Ford-Algorithmus für diesen Zweck vorgeschlagen.

Der Bellman-Ford-Algorithmus löst das Problem der kürzesten Pfade für den allgemeinen Fall, dass die Kantengewichte negativ sein können. Allerdings wird für einen Graph mit negativen Zyklen kein kürzester Pfad berechnet. Ein negativer Zyklus wird von dem Algorithmus entdeckt und gemeldet.

Bei einem Problem der kürzesten Pfade ist ein gewichteter, gerichteter Graph mit einer Gewichtsfunktion gegeben. Die Gewichtsfunktion bildet die Kanten des Graphs auf reellwertige Gewichte ab. Das Gewicht eines Pfades ist die Summe der Gewichte der einzelnen Knoten des Pfades. Der kürzeste ist derjenige, der das kleinste Gewicht hat.

Die kürzesten Pfade werden durch die Vorgängerfunktion repräsentiert. Jedem Knoten wird ein Vorgänger zugewiesen, so dass man den kürzesten Pfad vom Blattknoten bis zum Startknoten zurückverfolgen kann.

Der Bellman-Ford-Algorithmus verwendet die Methode der Relaxation. Für jeden Knoten wird eine obere Schranke für das Gewicht des kürzesten Pfades vom Startknoten bis zu diesem Knoten zugewiesen. Diese obere Schranke wird auch als Schätzung des kürzesten Pfades bezeichnet. Der Prozess des Relaxierens einer Kante versucht die Schätzung des Zielknotens zu verkleinern und das Vorgängerattribut wenn nötig zu aktualisieren. Nähere Informationen zu dem Bellman-Ford-Algorithmus, zu Relaxation und Korrektheitsbeweisen sind in [Cormen] nachzuschlagen.

Basierend auf der Fähigkeit des Bellman-Ford-Algorithmus, mit negativen Kantengewichten umgehen zu können, findet er seine Verwendung bei der Berechnung der längsten Pfade. Dabei werden alle Kantengewichte des Graphen negiert. Der Algorithmus wird unverändert auf einen solchen Graph angewendet und berechnet nach wie vor den kürzesten Pfad. Setzt man nach der Berechnung das Vorzeichen der Schätzungen wieder zurück, so erhält man nun den längsten Pfad.

Der Bellman-Ford-Algorithmus verwendet für seine Berechnungen die Kantengewichte. Im vorherigen Kapitel wurde eine Zuordnung der Gewichte zu den Kanten in einem Aufrufgraph vorgestellt. Das kann angepasst werden, indem das Gewicht eines Knotens den eingehenden Kanten zugewiesen wird.

5 Evaluation

Das Ziel der Evaluation ist, unterschiedliche Aspekte bezüglich des im Kapitel 3 erarbeiteten Verfahrens zur Bestimmung der maximalen Belegung des Aufrufkellers einer Java-Card-Anwendung anhand einiger Beispielprogramme zu beurteilen. Dabei werden mehrere Java-Card-Anwendungen betrachtet. Für jede Anwendung werden Aufrufgraphen mit verschiedenen Analyseverfahren erzeugt und daraufhin die oberen und unteren Schranken berechnet.

Als Erstes wird das Verhalten der mit verschiedenen Analyseverfahren erzeugten Aufrufgraphen charakterisiert. Dabei werden verschiedene Aufrufgraph-analyseverfahren hinsichtlich der Fähigkeit, den Anteil der instabilen Kanten in den Aufrufgraphen zu reduzieren, verglichen. Auf Grund des Entfernens der instabilen Kanten können sich oberen und unteren Schranken verändern: Die untere Schranke kann sich erhöhen und die obere Schranke verkleinern. Dadurch verbessert sich die Güte der zu bestimmenden maximalen Belegung des Aufrufkellers.

Außerdem wird die Anzahl der Knoten in den Aufrufgraphen präsentiert. Nach dem Entfernen von instabilen Kanten können einige Knoten für die Eintrittsknoten nicht erreicht werden. Das kann dazu führen, dass einige instabile Pfade verkürzt werden – was auch die Verkleinerung der oberen Schranke hervorrufen könnte.

Als Nächstes werden für einige Eintrittspunkte die durch die erzeugten Aufrufgraphen ermittelten oberen und unteren Schranken vorgestellt. Dabei wird noch einmal anhand praktischer Beispiele auf die Abhängigkeit vom Anteil der instabilen Kanten im Aufrufgraph eingegangen. Das Eliminieren von instabilen Kanten kann die Veränderung der oberen und unteren Schranken verursachen.

Fünf Aufrufgraphanalyseverfahren standen zur Verfügung: CHA, RTA, FTA sowie die zwei kombinierten RTA-DefUse und RTA-FTA. Mit jedem dieser Analyseverfahren wurde für jede Testanwendung ein Aufrufgraph erzeugt. Danach wurde die im Kapitel 3.2 beschriebene Klassifizierung der Kanten angewendet.

Als Testprogramme habe ich folgende Beispielapplets aus dem SUN Java Card Development Kit 2.2.1 verwendet: `HelloWorld`, `JavaLoyalty`, `Wallet`, `JavaPurse` und `PhotoCardApplet` und `SecurePurseApplet`.

Als Eintrittspunkte bei der Erzeugung der Aufrufgraphen für die Java-Card-Beispielanwendungen wurden Standardeinrittspunkte eines Java Card Applets, die Methoden `deselect`, `install`, `process` und `select`, festgelegt. Außerdem wurden einige Methoden der Java-Card-Laufzeitumgebung in die Menge der Eintrittspunkte aufgenommen. Das sind die Methode `javacard.framework-.Dispatcher.main` und alle Methoden des Paketes `com.sun.javacard.impl`. Die `main`-Methode wird beim Zurücksetzen der Karte von der virtuellen Java-Card-

Maschine aufgerufen und leitet die von der Host-Anwendung eingehenden Befehle in Form von APDUs an das aktuelle Applet weiter. Mit der Aufnahme aller Methoden des Pakets `com.sun.javacard.impl` in die Menge der Eintrittspunkte wurde an dieser Stelle konservativ abgeschätzt.

Zusätzlich habe ich zur Gegenüberstellung vier Beispielanwendungen aus Mobile Information Device Profile Reference Implementation 2.0 betrachtet. Das Mobile Information Device Profile (MIDP) ist ein Teil der Java-Micro-Edition-Plattform. Die Java-Micro-Edition-Plattform wurde ebenfalls für Geräte mit beschränkten Ressourcen im Bereich der Unterhaltungselektronik definiert. Die Java-Micro-Edition-Plattform stellt im Vergleich zur Java-Card-Technologie eine nicht so stark abgespeckte Version der Java-Plattform dar. Eine für Mobile Information Device Profile entwickelte Anwendung wird als MIDlet bezeichnet. Für mehr Information über Java-Micro-Edition-Plattform und ihre Komponenten verweise ich an dieser Stelle auf [Riggs] und [Schmatz]. Das Ziel der Gegenüberstellung war es in erster Linie, die Verwendung der dynamischen Methodenbindung in beiden Anwendungsarten zu vergleichen. Dabei wurden die Aufrufgraphen betrachtet, die für einige MIDlets ebenfalls mit verschiedenen Analyseverfahren erzeugt worden waren.

5.1 Aufrufgrapherzeugung

Das im Kapitel 3 entworfene Verfahren zur Bestimmung der maximalen Größe des Aufrufkellers basiert auf den statischen Aufrufgraphanalyseverfahren. Ein wichtiger Aspekt bei der Bestimmung der Aufrufgraphen für Java-Card-Anwendungen mit statischen Analyseverfahren ist die dynamische Methodenbindung. Die Analyseverfahren gehen bei der Erzeugung der Aufrufgraphen unterschiedlich damit um. Dadurch unterscheiden sie sich in der Genauigkeit, mit der die Aufrufgraphen bestimmt werden.

In diesem Abschnitt wird das Verhalten der Aufrufgraphen charakterisiert, die mit verschiedenen Aufrufgraphanalyseverfahren erzeugt werden. Es wird die Anzahl der Kanten und Knoten in Aufrufgraphen vorgestellt und diskutiert und dabei insbesondere auf die Anteile der instabilen Kanten in den jeweiligen Aufrufgraphen eingegangen, denn diese Kanten repräsentieren nicht eindeutig aufgelöste dynamisch gebundene Methodenaufrufe.

Anhand der Veränderungen im Hinblick auf die Anzahl der Knoten in den Aufrufgraphen, die mit verschiedenen Analyseverfahren erzeugt werden, kann man Schlussfolgerungen über das Verhalten der Pfade ziehen. Auf Grund der Nichterreichbarkeit einiger Knoten verkürzen sich die Pfade. Das könnte Auswirkungen auf die zu bestimmende maximale Belegung des Aufrufkellers haben.

5.1.1 Anteil der stabilen und instabilen Kanten

Die durch dynamisch gebundene Methodenaufrufe verursachten Kanten wurden im Kapitel 3.2 in stabile und instabile klassifiziert. In den Tabellen 5.1 und 5.2 sind Anteile der stabilen und instabilen Kanten in den Aufrufgraphen, die mit verschiedenen Analyseverfahren erzeugt wurden, aufgeführt. Es werden weiterhin stabile Kanten betrachtet, die durch dynamisch gebundene Methodenaufrufe verursacht wurden. In der Tabelle 5.1 sind die Messwerte für die betrachteten Java-Card-Anwendungen und in der Tabelle 5.2 für MIDlets aufgeführt.

Tabelle 5.1: Anzahl der durch dynamisch gebundenen Methodenaufrufe verursachten Kanten in Aufrufgraphen, die für einige Java-Card-Anwendungen erzeugt wurden, klassifiziert in stabile und instabile Kanten.

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.
HelloWorld	93	0	93	0	93	0	93	0	93	0
%	100	0	100	0	100	0	100	0	100	0
JavaLoyalty	96	2	96	2	96	2	96	2	97	0
%	97,96	2,04	97,96	2,04	97,96	2,04	97,96	2,04	100	0
Wallet	109	4	109	4	109	4	109	4	111	0
%	96,46	3,54	96,46	3,54	96,46	3,54	96,46	3,54	100	0
JavaPurse	150	2	150	2	150	2	150	2	151	0
%	98,68	1,32	98,68	1,32	98,68	1,32	98,68	1,32	100	0
PhotoCard	132	4	133	2	133	2	134	0	134	0
%	97,06	2,94	98,52	1,48	98,52	1,48	100	0	100	0
SecurePurse	151	9	151	8	151	8	152	6	152	6
%	94,38	5,62	94,97	5,03	94,97	5,03	96,20	3,80	96,20	3,80
Mittel %	97,42	2,58	97,77	2,23	97,77	2,23	98,22	1,78	99,37	0,63

Aus der Tabelle 5.1 ist ersichtlich, dass sich der Anteil der instabilen Kanten mit der Verwendung von aufwändigeren Aufrufgraphanalyseverfahren verkleinert hat. Im Endeffekt hat man in fast allen Fällen erreicht, alle instabile Kanten aus den Aufrufgraphen zu entfernen. Für das `HelloWorld` Applet wurden keine instabilen Kanten erzeugt, sodass auch keine Kanten aus dem Aufrufgraph entfernt werden konnten. Für die `JavaLoyalty`, `Wallet` und `JavaPurse` Applets blieb die Anzahl der instabilen Kanten erst unverändert und mit RTA-FTA wurden diese vollständig eliminiert.

Am Beispiel des `PhotoCard` Applets kann man Verfeinerungsschritte einzelner Analyseverfahren bezüglich instabiler Kanten beobachten. Mit der Verwendung der CHA wurden vier instabile Kanten erzeugt. Es handelt sich dabei um zwei Aufrufstellen mit jeweils zwei instabilen Kanten. Auf dem Bild 5.1 sind zwei dazugehörige Teilgraphen dargestellt. Die instabilen Kanten sind mit gestrichelten Linien gekennzeichnet.

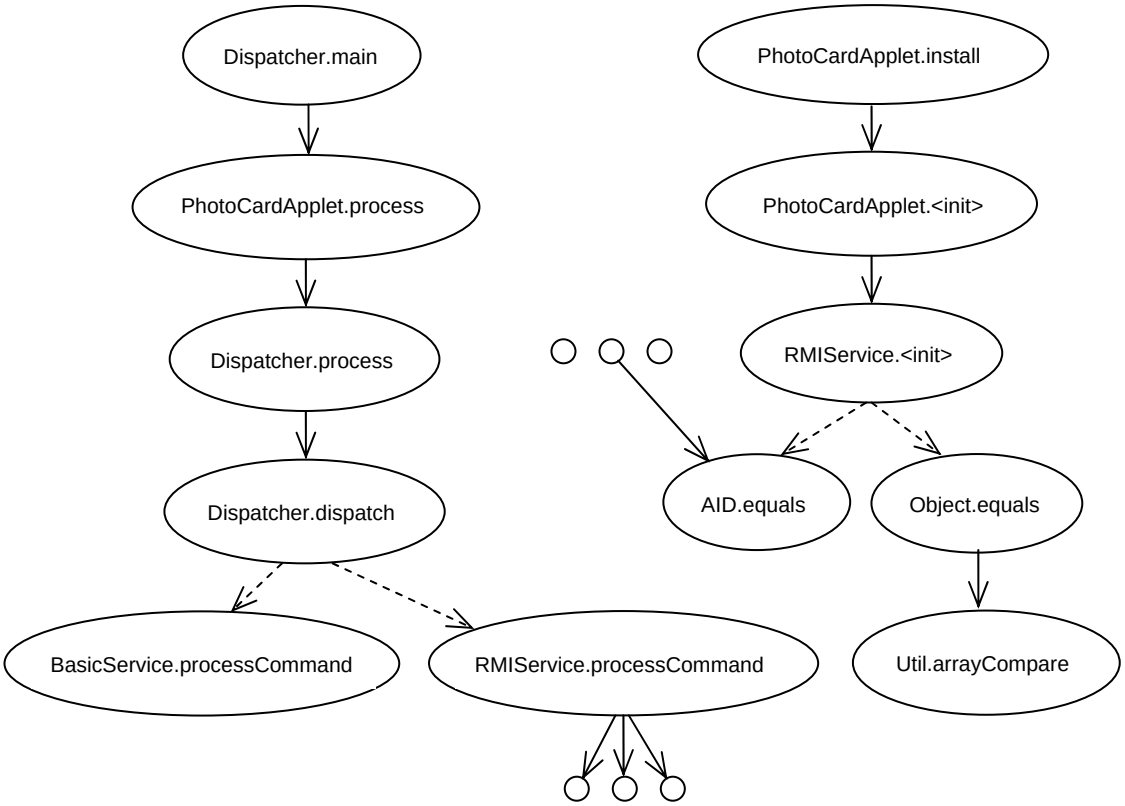

Bild 5.1: Zwei für `PhotoCard` Applet mit CHA erzeugte Teilgraphen mit jeweils zwei instabilen Kanten.

Die ersten beiden instabilen Kanten wurden bei der Analyse der Methode `Dispatcher.dispatch` erzeugt. An der Aufrufstelle zu der Methode `processCommand` wurde der statische Typ `Service` identifiziert. Dies ist eine Schnittstelle. Die Klassen `BasicService` und `RMIService` implementieren diese Schnittstelle. Deshalb werden die beiden Methoden `BasicService.-processCommand` und `RMIService.processCommand` als potenzielle Aufrufziele betrachtet.

Die beiden anderen instabilen Kanten entstanden bei der Analyse der Methode `RMIService.<init>`. An der Aufrufstelle zu der Methode `equals()` wurde `Object` als statischer Typ bestimmt. Die Klasse `Object` implementiert diese Methode. Außerdem wird diese Methode von der Klasse `AID` implementiert, die von der Klasse `Object` vererbt wird. Somit wird die Methode `AID.equals` ebenfalls als potenzielles Aufrufziel betrachtet.

Bei der Verwendung von RTA wurde erkannt, dass kein Objekt vom Typ `BasicService` erzeugt wurde, und somit wird die Methode `BasicService.processCommand` aus der Menge der potenziellen Aufrufziele entfernt. Die Kante zwischen `Dispatcher.dispatch` und `RMIService.processCommand` wird dadurch stabil, und es bleiben zwei instabile Kanten im Aufrufgraph.

FTA löst endgültig das Problem, indem alle instabilen Kanten aus dem Aufrufgraph entfernt werden. Anhand der für `RMIService.<init>`-Methode gebildeten Typmenge $S_{RMIService.<init>}$, in der kein Element vom Typ `AID` enthalten ist, wird entschieden die Kante zwischen den Knoten `RMIService.<init>` und `AID.equals` zu entfernen. Hierdurch wird de Kante zwischen den Knoten `RMIService.<init>` und `Object.equals` stabil.

Das `SecurePurse` Applet liefert ein Beispiel dafür, für das es nicht gelungen ist, alle instabilen Kanten zu entfernen. Nach der Verwendung aller vorhandenen Aufrufgraph-analyseverfahren sind immerhin sechs instabile Kanten geblieben. Eine genauere Betrachtung hat ergeben, dass es sich um drei Aufrufstellen mit jeweils zwei Aufrufzielen handelt. An diesen Aufrufstellen werden zu unterschiedlichen Zeitpunkten der Programmausführung tatsächlich verschiedene Methoden aufgerufen. Deshalb können die entsprechenden Kanten aus dem Aufrufgraph nicht entfernt werden.

Tabelle 5.2: Anzahl der durch dynamisch gebundene Methodenaufrufe verursachten Kanten in Aufrufgraphen, die für einige MIDlets erzeugt wurden, klassifiziert in stabile und instabile Kanten.

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.
Hanoi	2441	1529	2441	1529	2435	1499	2401	1021	2397	1177
%	61,49	38,51	61,49	38,51	61,90	38,10	70,16	29,84	67,07	32,93
WormMain	2555	1564	2555	1564	2548	1537	2519	1109	2533	1248
%	62,03	37,97	62,03	37,97	62,37	37,63	69,43	30,57	66,99	33,01
TilePuzzle	2439	1517	2439	1517	2433	1488	2391	1013	2380	1146
%	61,65	38,35	61,65	38,35	62,05	37,95	70,24	29,76	67,50	32,50
PhotoAlbum	2535	1575	2535	1575	2527	1551	2486	1063	2491	1224
%	61,68	38,32	61,68	38,32	61,97	38,03	70,05	29,95	67,05	32,95
Mittel %	61,71	38,29	61,71	38,29	62,07	37,93	69,97	30,03	67,15	32,85

Die Anteile der instabilen Kanten in den betrachteten MIDlets haben sich laut der Tabelle 5.2 unter Verwendung von aufwändigeren Aufrufgraphanalyseverfahren ebenfalls reduziert. Der mittlere prozentuelle Anteil der instabilen Kanten ist um 8 %

kleiner geworden. Allerdings konnte man nicht eine vollständige Eliminierung der instabilen Kanten erreichen. Der kleinste mittlere Anteil der instabilen Kanten beträgt 30,03 %.

Bei einer Gegenüberstellung der Messwerte aus den beiden Tabellen 5.1 und 5.2 sieht man, dass der ermittelte Anteil der instabilen Kanten für die betrachteten MIDlets erheblich größer ist als für die Java-Card-Anwendungen. Der Anteil der instabilen Kanten in den Aufrufgraphen, die für die MIDlets erzeugt wurden, lag durchschnittlich in einem Bereich von 38,29 % bis 30,03 %. Auf Grund des relativ großen Anteils der instabilen Kanten steigt die Wahrscheinlichkeit, dass Zyklen in Aufrufgraphen vorkommen.

In den Aufrufgraphen für Java-Card-Anwendungen betrug der mittlere Anteil der instabilen Kanten von 2,58 % bis 0,63 %. Die dynamische Methodenbindung in Java-Card-Anwendungen wird benutzt. Das `SecurePurse` Applet enthält Aufrufstellen, an denen abhängig von dem Empfängerobjekt verschiedene Methoden aufgerufen werden.

Ansonsten wurden in den betrachteten Java-Card-Anwendungen alle instabilen Kanten, die auf Grund der Verwendung nicht ausreichend präziser Aufrufgraphanalyseverfahren entstanden sind, mit aufwändigeren Analyseverfahren entfernt. Damit wurde das im Kapitel 3 definierte Konzept der Kantenklassifizierung erfolgreich eingesetzt. Unter der Bedingung, dass die betrachtete Anwendung keine Rekursion enthält bzw. der erzeugte Aufrufgraph keine Zyklen besitzt, können auf diesem Aufrufgraph gleiche obere und untere Schranken berechnet werden.

5.1.2 Anzahl der Knoten

Als Nächstes wird die Anzahl der Knoten in den Aufrufgraphen, die mit verschiedenen Analyseverfahren erzeugt wurden, vorgestellt. Die Anzahl der Knoten im Aufrufgraph hängt gewissermaßen mit der Eliminierung der instabilen Kanten aus dem Aufrufgraph zusammen. Nach dem Entfernen einer Kante kann der Zielknoten für den jeweiligen Eintrittspunkt nicht erreichbar werden. In der Tabelle 5.3 und 5.4 ist die Anzahl der Knoten in den Aufrufgraphen, die mit verschiedenen Analyseverfahren erzeugt wurden, dargestellt.

Man erkennt anhand der Tabelle 5.3, dass die Anzahl der Knoten für fast alle Anwendungen geringfügig zurückgegangen ist. In der Regel erfolgte das genau an der Stelle, an der eine der instabilen Kanten aus dem Aufrufgraph entfernt wurde. Es wurde in allen Fällen ein Blattknoten entfernt. Bei `HelloWorld` ist die Anzahl der Knoten verständlicherweise unverändert geblieben. Wie man in der Tabelle 5.1 sieht, wurden für diese Anwendung keine instabilen Kanten erzeugt. Bei `JavaLoyalty`, `Wallet` und `JavaPurse` könnten alle Fälle des Auftretens der instabilen Kanten erst mit dem RTA-FTA-Verfahren vollständig aufgelöst werden. Die Anzahl der Knoten ist an dieser Stelle mindestens um einen Knoten gefallen. Für `PhotoCard` wurde die Anzahl der Knoten bereits mit dem RTA-Verfahren um eins reduziert. Mit dem Entfernen einer Kante zwischen den Knoten `Dispatcher.dispatch` und `BasicService.-`

`processCommand` ist der Knoten `BasicService.processCommand` nicht erreichbar geworden (siehe dazu das im Kapitel 5.1.1 erwähnte Beispiel). Auf Grund des Entfernens einer Kante zwischen den Knoten `RMIService.<init>` und `AID.equals` mit FTA ist der Knoten `AID.equals` trotzdem erreichbar geblieben, denn dieser Knoten besitzt noch andere eingehende Kanten. Aus demselben Grund ist der oben genannte Knoten in dem Aufrufgraph für `SecurePurse` Applet nicht erreichbar geworden.

Tabelle 5.3: Anzahl der Knoten in Aufrufgraphen, die mit verschiedenen Analyseverfahren für die betrachteten Java-Card-Beispielanwendungen erzeugt wurden.

	CHA	RTA	RTA-DefUse	FTA	RTA-FTA
HelloWorld	252	252	252	252	252
JavaLoyalty	257	257	257	257	256
Wallet	270	270	270	270	268
JavaPurse	301	301	301	301	300
PhotoCard	311	310	310	310	310
SecurePurse	329	328	328	328	328

Tabelle 5.4: Anzahl der Knoten in Aufrufgraphen, die mit verschiedenen Analyseverfahren für betrachtete MIDlets erzeugt wurden.

	CHA	RTA	RTA-DefUse	FTA	RTA-FTA
Hanoi	2240	2240	2229	2157	2149
WormMain	2295	2295	2284	2224	2223
TilePuzzle	2246	2246	2235	2161	2147
PhotoAlbum	2261	2261	2250	2177	2171

Für die betrachteten MIDlets ist die Anzahl der Knoten ebenfalls geringfügig zurückgegangen (Tabelle 5.4). Sie ist zwar größer als bei den betrachteten Java-Card-Anwendungen, allerdings wurden in den für die MIDlets erzeugten Aufrufgraphen bei der Verwendung von aufwändigeren Analysen auch mehr Kanten entfernt.

Anhand der beiden Tabellen 5.3 und 5.4 ist ersichtlich, dass die Anzahl der Knoten in Aufrufgraphen, die mit verschiedenen Analyseverfahren erzeugt wurden, sowohl für die betrachteten Java-Card-Anwendungen als auch für die MIDlets unbedeutend kleiner geworden ist. Damit ist die Wahrscheinlichkeit sehr gering, dass der längste Pfad davon betroffen sein könnte. Die Veränderung der Anzahl der Knoten in den Aufrufgraphen kann nicht als aussagekräftiges Merkmal bei der Bestimmung der maximalen Belegung

des Aufrufkellers einer Java-Card-Anwendung betrachtet werden, so wie dies z. B. bei der Hüllenbildung der Fall ist [Wessels]. Wenn eine Kante aus dem Aufrufgraph entfernt wurde, könnten noch andere Kanten existieren, die den Zielknoten ebenfalls erreichen. Deshalb war die neue Zielsetzung der Kantenklassifizierung und Pfadbestimmung in dieser Arbeit von entscheidender Bedeutung. Damit ist man in der Lage, die Aufrufgraphen gezielt zu verbessern, um präzisere Ergebnisse für die Kellertiefen zu liefern.

5.2 Maximale Größe des Aufrufkellers

Das im Kapitel 3 vorgestellte Verfahren definiert die obere und untere Schranke für die mittels der Aufrufgraphen zu bestimmende maximale Belegung des Aufrufkellers einer Java-Card-Anwendung. Die Definition der beiden Schranken basierte auf der Klassifizierung der Kanten im Aufrufgraph in stabile und instabile.

In diesem Abschnitt werden für die betrachteten Beispielanwendungen die mit Hilfe der erzeugten Aufrufgraphen ermittelten oberen und unteren Schranken vorgestellt. Die Deutung dieser Ergebnisse steht in engem Zusammenhang mit dem Anteil der instabilen Kanten in den jeweiligen Aufrufgraphen. Die Veränderung der Anzahl der instabilen Kanten kann die Größen der oberen und unteren Schranken beeinflussen. Infolgedessen werden die Ergebnisse mit Rücksicht auf die Anzahl der instabilen Kanten in den jeweiligen Teilgraphen diskutiert.

Alle Aufrufgraphen, die für die betrachteten Java-Card-Anwendungen mit verschiedenen Analyseverfahren erzeugt wurden, sind azyklisch. Damit ist die Hauptvoraussetzung für eine erfolgreiche Bestimmung der maximalen Belegung des Aufrufkellers erfüllt.

Im Gegensatz dazu wiesen alle für die MIDlets erzeugte Aufrufgraphen Zyklen auf. Man konnte zwar die Anzahl der Zyklen unter Verwendung von aufwändigeren Aufrufgraphanalyseverfahren verringern, es ist jedoch nicht gelungen, einen azyklischen Graph zu erzeugen. Nach einer detaillierten Betrachtung einiger Zyklen habe ich festgestellt, dass das Vorkommen der Rekursion durchaus möglich ist. Abhängig von dem Typ des Empfängerobjektes kann es zu dem rekursiven Aufruf kommen. Ein Beispiel dafür sind z. B. die Klassen `java.util.Vector` und `java.util.Hashtable` mit der `toString`-Methode. Auf dem Bild 5.2 ist ein Ausschnitt aus dem Aufrufgraph dargestellt der mehrere Zyklen enthält. Natürlich könnte man die Analyseverfahren erweitern, um die Abschätzung der Rekursionstiefe zu ermöglichen. Dies ist aber nicht das Ziel dieser Diplomarbeit. Deshalb werden im weiteren Verlauf nur die ermittelten Ergebnisse für die Java-Card-Anwendungen betrachtet.

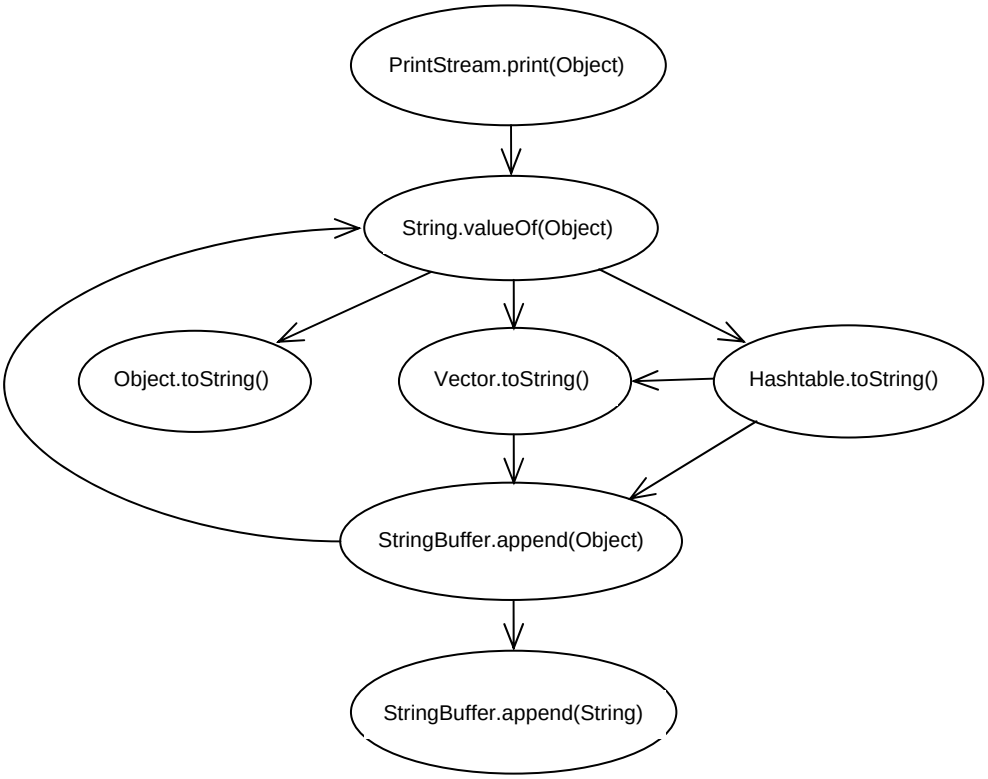

Bild 5.2: Ein Ausschnitt aus dem Aufrufgraph mit mehreren Zyklen.

Nach der Durchführung der Berechnungen an den Aufrufgraphen, die mit verschiedenen Analyseverfahren für die betrachteten Java-Card-Anwendungen erzeugt wurden, kamen die längsten Pfade mit der `javacard.framework.Dispatcher.main`-Methode als Eintrittsknoten zustande. Diese Methode gehört zu der Java-Card-Laufzeitumgebung und nicht zu dem Java Card Applet selbst. Für Eintrittspunkte des Java Card Applets – das sind die `select`, `deselect`, `install` und `process`-Methoden – wurden die längsten Pfade mit der `process`-Methode als Eintrittspunkt berechnet. Außerdem wurde festgestellt, dass ein Eintrittspunkt des Applets – nämlich die `install`-Methode – von keinem Eintrittspunkt der Java-Card-Laufzeitumgebung in den erzeugten Aufrufgraphen erreichbar ist.

Im Weiteren werden die Ergebniswerte für zwei Eintrittspunkte `main`-Methode und `process`-Methode betrachtet. In den meisten Fällen wurden die gleichen oberen und unteren Schranken bestimmt. Deshalb werden hier nur die Ergebniswerte für die Aufrufgraphen, die für die betrachteten Anwendungen mittels CHA erzeugt wurden, vorgestellt. Die vollständige Auflistung der Auswertungsdaten ist im Anhang A zu finden. In der Tabelle 5.5 sind für die zwei oben genannten Eintrittspunkte die oberen und unteren Schranken in Bytes aufgeführt, die auf Grundlage der mit Hilfe der CHA erzeugten Aufrufgraphen ermittelt wurden.

Tabelle 5.5: Die mit Hilfe der CHA bestimmten oberen und unteren Schranken in Bytes für zwei Eintrittspunkte `Dispatcher.main`*-Methode und* `process`*-Methoden des jeweiligen Applets .*

	HelloWorld		JavaLoyalty		Wallet		JavaPurse		PhotoCard		SecurePurse	
	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.
main	146	146	146	146	146	146	146	146	146	188	146	188
process	116	116	114	114	106	106	136	136	116	178	116	178

Die gleichen Werte für die `main`-Methode als Eintrittspunkt und die Anwendungen `HelloWorld`, `JavaLoyalty`, `Wallet` und `JavaPurse` lassen sich damit erklären, dass ein längster Pfad unabhängig von der jeweiligen Anwendung berechnet wurde. Dieser Pfad besteht aus Knoten, die durch Methoden der Java-Card-Laufzeitumgebung repräsentiert und dadurch unabhängig von den Methoden des betrachteten Java Card Applets sind.

Um den Zusammenhang zwischen den ermittelten oberen und unteren Schranken und den Anteilen der instabilen Kanten zu erläutern, werden die Anteile der stabilen und instabilen Kanten in den Teilgraphen mit den oben genannten Eintrittspunkten vorgestellt (Tabellen 5.6 und 5.7). Dabei werden wiederum die Einträge ausgeblendet, für die mit allen verwendeten Aufrufgraphanalyseverfahren keine instabilen Kanten erzeugt wurden. Die vollständige Auflistung der Messergebisse ist im Anhang A zu finden.

Am Beispiel der beiden Applets `Wallet` und `JavaPurse` aus Tabelle 5.6 sieht man, dass – obwohl in den Aufrufgraphen einige instabile Kanten vorhanden sind – für diese Applets jeweils die gleichen oberen und unteren Schranken bestimmt wurden. Die beiden stabilen und instabilen längsten Pfade sind in diesem Fall gleich und enthalten keine instabilen Kanten, sodass das weitere Anwenden der aufwändigeren Analyseverfahren eigentlich überflüssig ist. Auf Grund des Entfernens von weiteren instabilen Kanten aus dem Aufrufgraph mittels aufwändigeren Analyseverfahren wird der längsten Pfade nicht beeinflusst.

Im Gegensatz dazu kann – bei unterschiedlich bestimmten unteren und oberen Schranken – das Auflösen der instabilen Kanten zu dem Abgleich der beiden Werte führen. Dies ist möglich, wenn es den Analyseverfahren gelungen ist, alle dynamisch gebundenen Methodenaufrufe entlang des längsten Pfades eindeutig aufzulösen.

Am Beispiel des `PhotoCard` Applets für beide Eintrittspunkte (`main`- und `process`-Methoden) kann man erkennen, wie sich auf Grund der Auflösung der instabilen Kanten aus dem Aufrufgraph mit Hilfe der aufwändigeren Verfahren (Tabellen 5.6 und 5.7) die Differenz zwischen den oberen und unteren Schranken reduziert hat. In beiden Fällen wurden die gleichen oberen und unteren Schranken bestimmt.

Bei der Analyse des `PhotoCard` Applets mit CHA wurden für beiden Teilgraphen jeweils zwei instabile Kanten bestimmt. In beiden Fällen ist das die bereits aus dem vorherigen Kapitel bekannte Aufrufstelle in der Methode `Dispatcher.dispatch` (siehe dazu das im Kapitel 5.1.1 vorgestellte Beispiel und das Bild 5.1). Der dynamisch gebundene Methodenaufruf wurde mit Hilfe der RTA eindeutig aufgelöst, sodass in beiden Teilgraphen keine instabilen Kanten enthalten sind. Damit wurden für `PhotoCard` Applet die gleichen oberen und unteren Schranken bestimmt.

Tabelle 5.6: Anteile der stabilen und instabilen Kanten in Aufrufgraphen mit der Methode `javacard.framework.Dispatcher.main` als Eintrittsknoten.

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.
Wallet	88	4	88	4	88	4	88	4	90	0
JavaPurse	133	2	133	2	133	2	133	2	134	0
PhotoCard	116	2	117	0	117	0	117	0	117	0
SecurePurse	131	7	131	6	131	6	131	6	131	6

Tabelle 5.7: Anteile der stabilen und instabilen Kanten in Aufrufgraphen mit der `process`-Methode der jeweiligen Applet-Klasse als Eintrittsknoten.

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.
PhotoCard	71	2	72	0	72	0	72	0	72	0
SecurePurse	86	7	86	6	86	6	86	6	86	6

Für `SecurePurse` Applet ist es nicht gelungen, die gleichen oberen und unteren Schranken zu bestimmen. Dieses Java Card Applet enthält die Aufrufstellen, an denen zu verschiedenen Zeitpunkten der Appletausführung unterschiedliche Methoden aufgerufen werden. Deswegen konnten instabile Kanten, die durch diese Aufruf-instruktionen verursacht wurden, nicht entfernt werden. Einige dieser instabilen Kanten befinden sich auf dem längsten Pfad. Dadurch konnten keine gleichen oberen und unteren Schranken bestimmt werden.

Anhand der vorgestellten Messwerten ist ersichtlich, dass bereits mit RTA gute Ergebnisse bezüglich der definierten Güte erzielt werden konnten. Für fast alle der betrachteten Java-Card-Anwendungen wurden die gleichen oberen und unteren Schranken ermittelt. Das erarbeitete Verfahren bietet die Möglichkeit, bereits mit nicht ganz so aufwändigen Aufrufgraphanalysen die maximale Größe des Aufrufkellers mit

Hilfe der Aufrufgraphen erfolgreich zu bestimmen. Dabei stehen die Kantenklassifizierung und die Berechnung des längsten Pfades im Zentrum des Verfahrens.

5.3 Verteilung der Gesamtgewichte einzelner Blätter

Als Güte bei der Bestimmung der maximalen Belegung des Aufrufkellers der mittels der statischen Analyseverfahren bestimmten Aufrufgraphen wurde im Kapitel 3.4 die Differenz zwischen oberen und unteren Schranken definiert. Werden die beiden Schranken gleich groß berechnet, sind keine weiteren Verfeinerungen mittels der statischen Aufrufgraphanalyseverfahren nötig, denn der längste Pfad kann dadurch nicht mehr verkürzt werden. Wird aber die obere Schranke größer als die untere bestimmt, könnte man versuchen, den längsten Pfad mit besseren Analyseverfahren zu verbessern.

Vor dem Einsatz von aufwändigeren Analyseverfahren können einige Eigenschaften des Aufrufgraphzustandes interessant sein. Anhand dieser Eigenschaften kann man beurteilen, wie groß der Aufwand sein kann, die Anzahl der instabilen Pfade zu reduzieren, um dadurch präzisere Ergebnisse zu liefern.

Vor allem ist die Größe der Differenz zwischen den oberen und unteren Schranken entscheidend. Anhand dieser Differenz kann man beurteilen, ob der weitere Einsatz der aufwändigeren Analyseverfahren lohnenswert ist. Bei geringfügigen Abweichungen ist es denkbar, dass keine weiteren Optimierungen nötig sind.

Die Verteilung der Gesamtgewichte der Blattknoten kann ebenfalls in einigen Fällen hilfsreich sein. Vor allem sind die Blätter bedeutend, deren Gesamtgewicht größer als die untere Schranke ist. Anhand dieser Informationen kann man Aussagen darüber treffen, wie weit die Werte voneinander verstreut sind. Die Verteilung der Gesamtgewichte der Blattknoten habe ich mit Hilfe der folgenden Diagramme veranschaulicht.

Auf dem Diagramm sind die Gesamtgewichte der stabilen und instabilen Blätter in aufsteigender Reihenfolge dargestellt. Hierdurch lässt sich die Verteilung der Gesamtgewichte als Verlauf von zwei Kurven beobachten. Die Differenz zwischen oberen und unteren Schranken lässt sich ebenfalls von dem Diagramm ablesen.

Auf dem Bild 5.1 ist ein Diagramm für die `main`-Methode der Klasse `Dispatcher` dargestellt. Die Differenz zwischen den Schranken beträgt 42 Bytes, sodass damit das Verbesserungspotential vorhanden ist. Man sieht, dass auf dem Intervall von der unteren Schranke bis zur oberen ziemlich viele Punkte enthalten sind. Am Anfang ändern sich die Werte kaum, sodass die Kurve parallel zur X-Achse verläuft. Ab Mitte des Abschnittes weist die Kurve eine gleichmäßige Steigung auf. Es gibt viele Blätter, die ziemlich nah an dem Maximum liegen. Mit einer erfolgreichen Verbesserung der Pfade, die zu diesen Blattknoten gehören, können die gleichen oberen und unteren Schranken bestimmt werden.

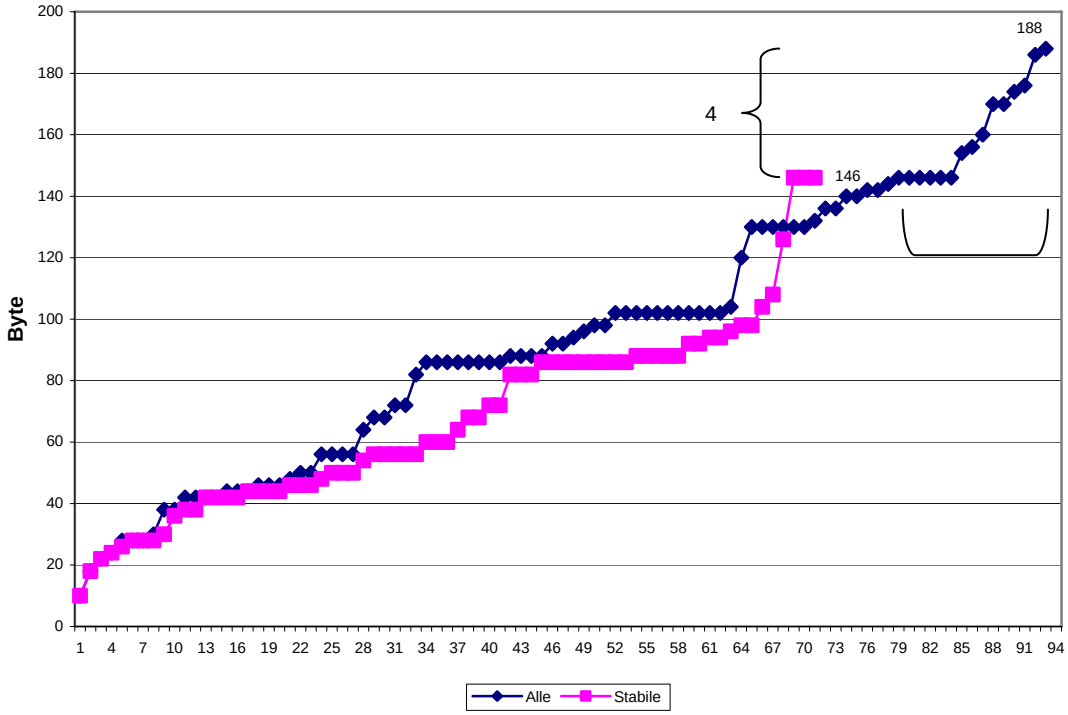

Bild 5.1: Verteilung der Blättergewichte für die `main`-Methode der `Dispatcher`-Klasse als Eintrittspunkt bei der Analyse des `PhotoCard` Applets mit CHA

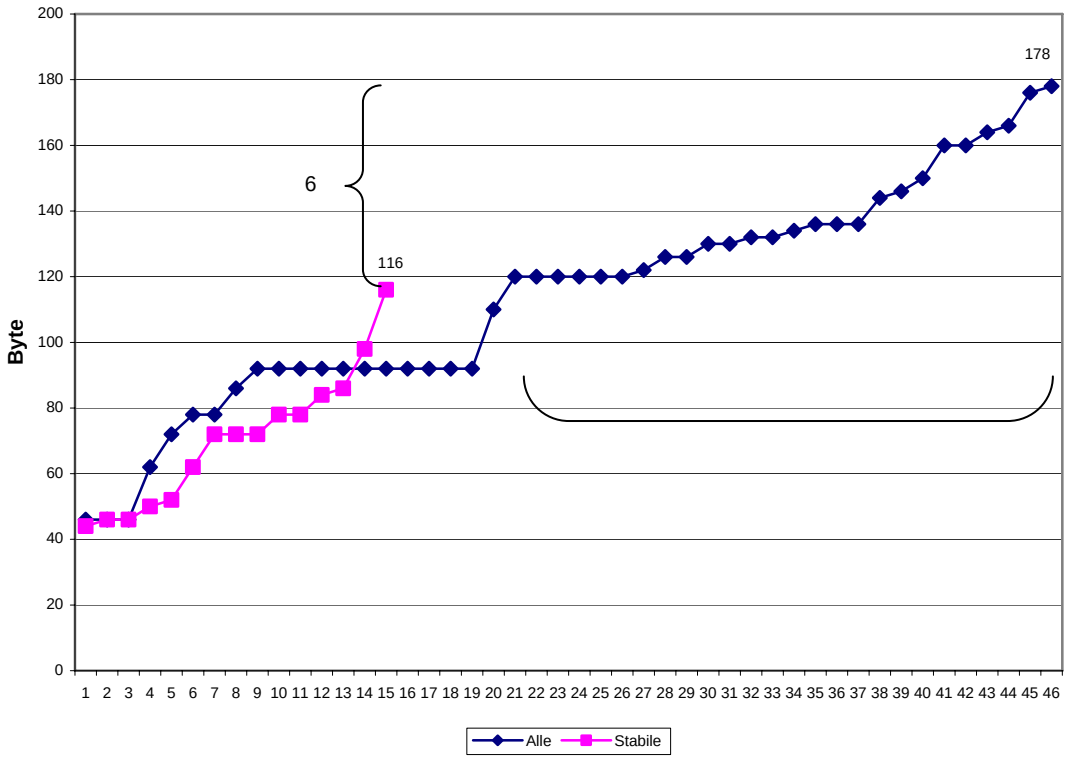

Bild 5.2: Verteilung der Blättergewichte für die `PhotoCardApplet.process`-Methode als Eintrittspunkt bei der Analyse mit CHA

Für die Methode `process` der Klasse `PhotoCardApplet` ist ein Diagramm auf dem Bild 5.2 zu sehen. Die Differenz ist diesmal 62 Bytes groß. Um die gleichen oberen und unteren Schranken zu bestimmen, besteht noch mehr Potential zur Verbesserung von instabilen Pfaden als im vorherigen Beispiel. Abgesehen von dem Sprung am Anfang des Abschnitts verläuft die Kurve bezogen auf den größten Teil des Intervalls gleichmäßig. Um die Differenz zwischen den oberen und unteren Schranken zu verkleinern, müssen in diesem Fall mehrere Pfade verbessert werden.

Die Aussagen über die Verteilung der Gesamtgewichte und somit die vorgestellten Diagramme sind kein direkte Maß für die Optimierung der instabilen Pfaden. Jeder der instabilen Pfade besitzt eine oder mehrere instabile Kanten, die bei Anwendung der aufwändigeren Analyseverfahren ebenfalls berücksichtigt werden sollten. Somit stellt das Zusammenspiel zwischen den instabilen Pfaden und den vorhandenen instabilen Kanten einen wichtigen Aspekt bei der Betrachtung der Verteilung von Gesamtgewichten der Blattknoten dar. Man kann sich dabei eine Zuordnung der instabilen Pfaden zu den instabilen Kanten bzw. zu den Instruktionen, die diese Kanten verursacht haben, und umgekehrt vorstellen. Auf Grund dieser Zuordnung kann man sehen, wie viele instabile Kanten ein Pfad enthält oder an wie vielen Pfaden eine Instruktion beteiligt ist, die instabile Kanten verursacht hat. Daraus lässt sich dann die Erkenntnis ableiten, wie groß der Aufwand sein könnte, um die einzelnen Pfade zu verbessern. Dementsprechend sind anhand dieser Zuordnung unterschiedliche Auswahlstrategien denkbar. Wenn es z. B. mehrere längste Pfade gäbe, könnte man sich vorstellen, dass der Pfad genommen wird, der den größten Anteil an instabilen Kanten hat oder der eine Kante enthält, deren Instruktion am häufigsten an den instabilen Pfaden beteiligt ist.

Im Hinblick auf die beiden vorgestellten Diagramme ist z. B. bekannt, dass alle instabilen Pfade von einer dynamisch gebundenen Aufrufinstruktion verursacht wurden, d. h. mit dem erfolgreichen Auflösen dieses Methodenaufrufs würden die gleichen oberen und unteren Schranken berechnet.

Das erarbeitete Verfahren mit der verwendeten Klassifizierung der Kanten und Bestimmung der Pfade liefert Mittel, um den Fehler der ermittelten Größe des Aufrufkellers einer Java-Card-Anwendung zu bestimmen und den Aufwand für die Verbesserung dieses Fehlers zu beurteilen. Dabei können aufwändigere Aufrufgraphanalyseverfahren gezielt eingesetzt werden. Die Verteilung der Gesamtgewichte der Blätter und die Zuordnung der instabilen Pfade zu den instabilen Kanten bieten gemeinsam einen soliden Ansatz, um aussagekräftige Entscheidungen bei der Entwicklung von Auswahlstrategien zur Optimierung von instabilen Pfaden mit aufwändigeren Aufrufgraphanalyseverfahren zu treffen. Allerdings wurde die oben genannte Zuordnung im Rahmen dieser Arbeit nicht umgesetzt. Dieser Aspekt dient als mögliche Erweiterung des erarbeiteten Verfahrens. Dabei können verschiedene Auswahlstrategien konzipiert und getestet werden.

6 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde das Verfahren zur Bestimmung der Größe des Aufrufkellers einer Java-Card-Anwendung entworfen und umgesetzt. Das Verfahren basiert auf mit statischen Analyseverfahren erzeugten Aufrufgraphen. Dabei stand der Aspekt der dynamischen Methodenbindung im Vordergrund. Der Typ des Empfängerobjektes bei einem dynamisch gebundenen Methodenaufruf wird erst zur Laufzeit bekannt. Deswegen kann man die dynamisch gebundenen Methodenaufrufe mit statischen Mitteln nicht eindeutig auflösen.

Angesichts des Verhaltens der dynamisch gebundenen Methodenaufrufe gegenüber den statischen Aufrufgraphanalyseverfahren wurde eine Klassifizierung der Kanten im Aufrufgraph in stabile und instabile eingeführt. Damit wurden die Kanten markiert, die mit der Verwendung eines aufwändigeren Analyseverfahrens aus dem Aufrufgraph eventuell entfernt werden könnten.

Die Bestimmung der maximalen Größe des Aufrufkellers auf der Grundlage des Aufrufgraphen wurde auf das Problem des längsten Pfades abgebildet. Dabei muss der Aufrufgraph als Hauptvoraussetzung für eine erfolgreiche Berechnung azyklisch sein. Bei der Umsetzung wurde der Bellman-Ford-Algorithmus implementiert.

Für die Beurteilung der Genauigkeit der bestimmten maximalen Größe des Aufrufkellers wurden obere und untere Schranken bezüglich der mit statischen Analyseverfahren erzeugten Aufrufgraphen und der eingeführten Klassifizierung der Kanten ermittelt. Die Differenz zwischen beiden Schranken stellt ein Maß für die Güte der bestimmten Größe des Aufrufkellers dar. Wurden ungleiche obere und untere Schranken berechnet, so kann man versuchen, den Aufrufgraph mit aufwändigeren Analyseverfahren zu verbessern. Hat man gleiche obere und untere Schranken bestimmt, obwohl im Aufrufgraph noch instabile Kanten vorhanden sind, ergibt es kein Sinn, den Aufrufgraph mit aufwändigeren Analyseverfahren zu verbessern.

Das erarbeitete Verfahren wurde mit sechs Java-Card-Anwendungen aus dem Java Card Development Kit 2.2.1 getestet. Für jede Anwendung wurden Aufrufgraphen mit verschiedenen Analyseverfahren erzeugt. Außerdem wurden vier MIDlets aus MIDP betrachtet, um die Anteile der instabilen Kanten in Aufrufgraphen für verschiedene Anwendungsarten zu vergleichen. Dabei sind folgende Erkenntnisse gewonnen worden.

Als Erstes ist zu erwähnen, dass keine Zyklen in den erzeugten Aufrufgraphen für betrachtete Java-Card-Anwendungen entdeckt worden sind, sodass damit die Hauptvoraussetzung für weitere Berechnungen erfüllt wurde. Als Nächstes habe ich bei allen Aufrufgraphen für verschiedene Anwendungen einen sehr geringen Anteil an instabilen Kanten beobachtet, wobei diese mit der Verwendung von aufwändigeren Verfahren in fast allen Fällen vollständig aufgelöst wurden. Damit wurde die

eingeführte Klassifizierung der Kanten erfolgreich eingesetzt und die Voraussetzung zur Ermittlung von guten Ergebnissen bezüglich definierter Güte erfüllt. Die gleichen oberen und unteren Schranken konnten bereits mit nicht ganz aufwändigen Analyseverfahren bestimmt werden. Dabei spielten die eingeführte Klassifizierung der Kanten und die Bestimmung der Pfade eine entscheidende Rolle. Für den Fall, wenn ungleiche obere und untere Schranken berechnet wurden, bietet das Verfahren Mittel, um den Verbesserungsaufwand der instabilen Pfade zu beurteilen und aufwändigere Aufrufgraphanalyseverfahren gezielt anzuwenden.

Im Gegensatz dazu wiesen die für betrachtete MIDlets erzeugten Aufrufgraphen einen erheblich größeren Anteil an instabilen Kanten. Außerdem wurden mehrere Zyklen entdeckt. Untersuchungen haben gezeigt, dass es sich an einigen Stellen um rekursive Aufrufe handeln konnte. Es ist schwer mit statischen Mitteln die Rekursionstiefe abzuschätzen. Deshalb wurden keine weiteren Berechnungen mit dem erarbeiteten Verfahren an den MIDlets durchgeführt.

Ich sehe zwei Richtungen, in denen das entworfene Verfahren erweitert werden könnte. Zum einen ist das die Konzipierung der Auswahlstrategien zur gezielten Verbesserung der instabilen Kanten (siehe Kapitel 5.3). Zum anderen ist das die Fähigkeit der Analyseverfahren bei der Erzeugung von Aufrufgraphen die nativen Methoden zu berücksichtigen. Dadurch kann die im Kapitel 3.5.2 beschrieben Problematik mit Eintrittspunkten gelöst werden. In dieser Arbeit wurden Aufrufgraphanalyseverfahren verwendet, die in der Lage sind, nur Java-Methoden analysieren zu können.

Literaturverzeichnis

Java Card

[Chen] Chen, Zhiqun: *Java Card Technology for Smart Cards. The Java Series.* Addison Wesley, 2000.

[Hansmann] Hansmann, Uwe; Nicklous, Martin S.; Schäck, Thomas; Seliger, Frank: *Smart Card Application Development Using Java.* Springer, 2000.

[Horstmann] Horstmann, Cay S.; Cornell, Gary: *Core Java 2 Band 1 – Grundlagen.* Markt+Technik, 2002.

[JCRE] Sun Microsystems, Inc.: *Java Card™ Platform, Version 2.2.1, Runtime Environment Specification Virtual Machine Specification.* Sun Microsystems, October, 2003.

[JCVM] Sun Microsystems, Inc.: *Java Card™ Platform, Version 2.2.1, Virtual Machine Specification.* Sun Microsystems, October, 2003.

[Kopp] Kopp, Markus: Die virtuelle Java Maschine. In: *Linux Magazin* (1997), Heft 5.

[Lindholm] Lindholm, Tim und Yellin, Frank: *The Java Virtual Machine Specification, Second Edition. The Java Series.* Addison Wesley, 1999.

[Rankl] Rankl, Wolfgang und Effing, Wolfgang: *Handbuch der Chipkarten, Aufbau – Funktionsweise – Einsatz von Smart Cards.* Hanser, 2002.

[Riggs] Riggs, Roger; Taivalsaari, Antero; van Peursem, Jim etc: *Programming Wireless Devices with the Java 2 Platform, Micro Edition, 2nd Edition.* The Java Series. Addison Wesley, 2003.

[Schmatz] Schmatz, Klaus-Dieter: *Java 2 Micro Edition.* Dpunkt Verlag, 2004.

Aufrufgraph

[Bodden] Bodden, Eric: *A High-level View of Java Applications.* OOPSLA'03, 2003.

[Dahm] Dahm, Markus: *Byte Code Engineering with the BCEL API.* Freie Universität Berlin, Technical Report, 2001.

[Grove] Grove, David, DeFouw, Greg, Dean, Jeffrey und Chambers, Craig: *Call Graph Construction in Object-Oriented Languages.* In Proceeding of the Twelfth Annual Conference on Object-Oriented

Programming Systems, Languages and Applications, 1997.

[Rayside] Rayside, Derek: *Polymorphism is a Problem.* Panel on Reverse Engineering and Architecture at CSMR'02, 2002.

[Souter] Souter, Amie L.; Pollock, Lori L.: *Characterization and Automatic Identification of Type Infeasible Call Chains.* University of Delaware, 2003.

[Thies] Thies, Michael: *Combining Static Analysis of Java Libraries with Dynamic Optimization.* Universität Paderborn, Dissertation, 2001.

[Tip] Tip, Frank; Palsberg, Jens: *Scalable Propagation-Based Call Graph Construction Algorithms.* Appears in proceedings of the ACM Conference of Object-Oriented Programming Systems, Languages and Applications, 2000.

[Wessels] Wessels, Hermann: *Optimierung dynamischer Methodenbindung zum Entfernen ungenutzter Programmelemente durch Kombination statischer Analyseverfahren.* Universität Paderborn, Diplomarbeit, 2004.

Längster Pfad

[Cormen] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.: *Introduction to Algorithms.* MIT Press, Twentieth printing, 1998.

[Garey] Garey, Michael R.; Johnson, David S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company, 1979.

[Lawler] Lawler, Eugene L.: *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

Anhang

A. Detaillierte Auswertungsdaten

Tabelle A.1: Die oberen und unteren Schranken für die Aufrufgraphen mit der Methode `javacard.framework.Dispatcher.main` *als Eintrittsknoten.*

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.
HelloWorld	146	146	146	146	146	146	146	146	146	146
JavaLoyalty	146	146	146	146	146	146	146	146	146	146
Wallet	146	146	146	146	146	146	146	146	146	146
JavaPurse	146	146	146	146	146	146	146	146	146	146
PhotoCard	146	188	188	188	188	188	188	188	188	188
SecurePurse	146	188	146	188	146	188	146	188	146	188

Tabelle A.2: Die oberen und unteren Schranken für die Aufrufgraphen mit der `process`*-Methode der jeweiligen Anwendungen als Eintrittsknoten.*

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.	Untr.	Obr.
HelloWorld	116	116	116	116	116	116	116	116	116	116
JavaLoyalty	114	114	114	114	114	114	114	114	114	114
Wallet	106	106	106	106	106	106	106	106	106	106
JavaPurse	136	136	136	136	136	136	136	136	136	136
PhotoCard	116	178	178	178	178	178	178	178	178	178
SecurePurse	116	178	116	178	116	178	116	178	116	178

Tabelle A.3: Anteilel der stabilen und instabilen Kanten in Aufrufgraphen mit der Methode `javacard.framework.Dispatcher.main` *als Eintrittsknoten.*

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.
HelloWorld	76	0	76	0	76	0	76	0	76	0
JavaLoyalty	79	0	79	0	79	0	79	0	79	0
Wallet	88	4	88	4	88	4	88	4	90	0
JavaPurse	133	2	133	2	133	2	133	2	134	0
PhotoCard	116	2	117	0	117	0	117	0	117	0
SecurePurse	131	7	131	6	131	6	131	6	131	6

Tabelle A.4: Anteile der stabilen und instabilen Kanten in Aufrufgraphen mit der `process`*-Methode der jeweiligen Applet-Klasse als Eintrittsknoten.*

	CHA		RTA		RTA-DefUse		FTA		RTA-FTA	
	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.	St.	Inst.
HelloWorld	30	0	30	0	30	0	30	0	30	0
JavaLoyalty	32	0	32	0	32	0	32	0	32	0
Wallet	40	0	40	0	40	0	40	0	40	0
JavaPurse	87	0	87	0	87	0	87	0	87	0
PhotoCard	71	2	72	0	72	0	72	0	72	0
SecurePurse	86	7	86	6	86	6	86	6	86	6