UNIVERSITÄT KOBLENZ · LANDAU

Masterarbeit

Objektorientierte High-Level Datenflussanalyse

David Mebus
1. September 2019

Gutachter: Prof. Dr. Jan Jürjens
M. Sc. Sven Peldszus

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universität Koblenz
Universitätsstraße 1
56070 Koblenz
https://rgse.uni-koblenz.de

David Mebus
dmebus@uni-koblenz.de
Matrikelnummer: 211200114
Studiengang: Master Informatik
Prüfungsordnung: MPO2012

Masterarbeit in der AG Softwaretechnik
Thema: Objektorientierte High-Level Datenflussanalyse

Eingereicht: 1. September 2019

Betreuer: Sven Peldszus

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universität Koblenz
Universitätsstraße 1
56070 Koblenz

Ehrenwörtliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-ROM).

Koblenz, den 1. September 2019

David Mebus

Abstrakt

Datenflussmodelle in der Literatur weisen oftmals einen hohen Detailgrad auf, der sich auf die auf den Modellen durchgeführten Datenflussanalysen überträgt und diese somit schwerer verständlich macht. Da ein Datenflussmodell, das von einem Großteil der Implementierungsdetails des modellierten Programms abstrahiert, potenziell leichter verständliche Datenflussanalysen erlaubt, beschäftigt sich die vorliegende Masterarbeit mit der Spezifikation und dem Aufbau eines stark abstrahierten Datenflussmodells und der Durchführung von Datenflussanalysen auf diesem Modell. Das Modell und die darauf arbeitenden Analysen wurden testgetrieben entwickelt, sodass ein breites Spektrum möglicher Datenflussszenarien abgedeckt werden konnte. Als konkrete Datenflussanalyse wurde unter anderem eine statische Sicherheitsprüfung in Form einer Erkennung unzureichender Nutzereingabenbereinigungen durchgeführt. Bisher existiert kein Datenflussmodell auf einer ähnlich hohen Abstraktionsebene. Es handelt sich daher um einen einzigartigen Lösungsentwurf, der Entwicklern die Durchführung von Datenflussanalysen erleichtert, die keine Expertise auf diesem Gebiet haben.

Data flow models in the literature are often very fine-grained, which transfers to the data flow analysis performed on them and thus leads to a decrease in the analysis' understandability. Since a data flow model, which abstracts from the majority of implementation details of the program modeled, allows for potentially easier to understand data flow analyses, this master thesis deals with the specification and construction of a highly abstracted data flow model and the application of data flow analyses on this model. The model and the analyses performed on it have been developed in a test-driven manner, so that a wide range of possible data flow scenarios could be covered. As a concrete data flow analysis, a static security check in the form of a detection of insufficient user input sanitization has been performed. To date, there's no data flow model on a similarly high level of abstraction. The proposed solution is therefore unique and facilitates developers without expertise in data flow analysis to perform such analyses.

Inhaltsverzeichnis

Abbildungsverzeichnis **x**

1 Einleitung **1**

2 Grundlagen **3**
2.1 Durchgehendes Beispiel . . . 3
2.2 Objektorientierte Konzepte . . . 9
2.2.1 Datenkapselung . . . 9
2.2.2 Polymorphie . . . 9
2.2.3 Java-spezifische Umsetzung von Polymorphie . . . 10
2.3 Sicherheitsanforderungen . . . 13
2.4 GRaViTY . . . 13
2.4.1 Modellspezifikation . . . 14
2.4.2 Modellaufbau . . . 16
2.4.3 Triple-Graph-Grammatik (TGG) . . . 18
2.5 Datenfluss . . . 19
2.5.1 Datenflussdarstellung . . . 19
2.5.2 Datenflussanalyse . . . 22

3 Related Work **23**
3.1 Datenflussmodelle . . . 23
3.2 Datenflussanalyse . . . 23

4 Aufbau eines Datenflussmodells **25**
4.1 Spezifikation . . . 25
4.2 Modellaufbau . . . 28

5 Datenflussanalyse **33**
5.1 Extraktion von Datenflüssen . . . 34
5.2 Konsistenzprüfung der Integrity-Eigenschaft . . . 35
5.3 Konsistenzprüfung der Secrecy-Eigenschaft . . . 40

6 Implementierung und Evaluation **41**
6.1 Implementierung . . . 41
6.1.1 Komponentendiagramm der GRaViTY-Erweiterung . . . 41
6.1.2 Hinzugefügte TGG-Regeln . . . 42
6.1.3 Datenflussanalyse mit GReQL . . . 44

6.2 Evaluation 48
6.2.1 Modellaufbau 48
6.2.2 Datenflussanalyse 58

7 Fazit **61**

Literaturverzeichnis **63**

Abbildungsverzeichnis

2.1 Startseite der Check-in-Anwendung einer Airline 4
2.2 Metamodell des von GRaViTY erstellten Programmmodells 15
2.3 Beispielinstanz eines von GRaViTY erstellten Programmmodells . . . 16
2.4 Allgemeiner Ablauf des Programmmodellaufbaus in GRaViTY 17
2.5 Beispiel einer TGG-Regel zum Transformieren von Feldzugriffen . . . 18
2.6 Datenflussgraph für die Definition der Methode *actionPerformed* . . . 20
2.7 Datenflussdiagramm zur Weiterleitung von Nutzereingaben an die Buchungsdatenbank 21

4.1 Metamodell des Datenflussmodells 27
4.2 Beispielinstanz des Datenflussmodells 28
4.3 Ablauf des Datenflussmodellaufbaus 29
4.4 Generierter Dot-Graph für die Definition der Methode *actionPerformed* 30
4.5 Beispiel eines Reduktionsschritts im intraprozeduralen Datenflussmodell 31

5.1 Anfragediagramm für die Extraktion aller Datenflüsse zwischen Signaturen und Memberdefinitionen 34
5.2 Anfragediagramm für die Extraktion aller Datenflüsse zwischen der Signatur des Felds *engine* und anderen Memberdefinitionen 34
5.3 Ergebnis der Anfrage aus Abbildung 5.2 35
5.4 Ausschnitt des Datenflussmodells mit Annotationen 36
5.5 Anfragediagramm des für unzureichende Eingabebereinigung typischen Datenflussmusters 37
5.6 Ergebnis der Anfrage aus Anfragediagramm 5.5 38
5.7 Ausschnitt des Datenflussmodells nach Hinzufügen der Eingabebereinigung 39
5.8 Anfragediagramm für die Konsistenzprüfung der Secrecy-Eigenschaft 40

6.1 Komponentendiagramm der GRaViTY-Erweiterung 42
6.2 TGG-Regel zum Transformieren von Datenflüssen 43
6.3 TGG-Regel zum Transformieren von Datenflussquellen 43
6.4 TGG-Regel zum Transformieren von Datenflusszielen 44
6.5 GReQL-Ausgabe des gefundenen Memberpaars und deren gemeinsamen Datenflusspfads 47
6.6 Ausführungszeiten der einzelnen Schritte des Modellaufbaus in Sekunden 54

6.7 Prozentuale Ausführungszeiten der einzelnen Schritte des Modellaufbaus . . . 54
6.8 Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Codezeilen . . . 55
6.9 Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Klassen . . . 55
6.10 Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Methoden . . . 56
6.11 Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Felder . . . 56
6.12 Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Member . . . 57
6.13 Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl AST-Knoten . . . 57

1 Einleitung

In den Medien wird regelmäßig von der Aufdeckung neuer Sicherheitslücken in zum Teil sicherheitskritischer Software berichtet. Diese werden oftmals nur langsam behoben [GS]. Insbesondere falls der Angriffssicherheit bei der Softwareentwicklung eine eher untergeordnete Rolle zukommt und Schwachstellen erst spät entdeckt werden, ist der Aufwand für deren Behebung deutlich größer als bei einer frühen Erkennung. Diese wird z. B. durch die Berücksichtigung von Sicherheitsfragen bereits in der Entwurfsphase erleichtert [Neu18].

Der Code von Java-Projekten wie Mozilla Rhino[1] kann beispielsweise von einer frühen Berücksichtigung von Angriffssicherheit profitieren. Rhino ist eine quelloffene JavaScript-Implementierung in Java, die als Standard-JavaScript-Engine Teil der Java 6 SE ist. Im Jahr 2011 wurde eine Schwachstelle in Rhino gefunden, die Remote Code Execution ermöglichte [Sch] [Dat]. Diese kann mithilfe der Schwäche CWE-20 (Improper Input Validation)[2] ausgenutzt werden, da ein übergebener String von JavaScript-Code mit unzureichender Prüfung bzw. Bereinigung evaluiert wird. Eine solche Schwäche kann z. B. mit einer Tool-unterstützten Datenflussanalyse vorzeitig identifiziert und durch Hinzufügen einer Eingabebereinigung behoben werden.

Allerdings gestaltet sich die Umsetzung einer solchen Analyse schwierig, da eine geeignete Repräsentation des Datenflusses erforderlich ist. Aufgrund der Fülle an Informationen, die für die eigentliche Analyse irrelevant sind sowie nicht explizit enthaltene Informationen, eignet sich der gesamte abstrakte Syntaxbaum z. B. nicht als Grundlage für effiziente Datenflussanalysen. Daher wird eine Programmrepräsentation benötigt, welche schlank und einfach ist.

Das GRaViTY-Tool von Peldszus et al. [PKLS15] beinhaltet beispielsweise die Erstellung eines Programmmodells aus Java Quellcode, in welchem nur für die Analyse objektorientierter Programme relevante Informationen, wie Klassen, Methoden und Felder sowie ihre Beziehungen zueinander, durch Knoten bzw. Kanten dargestellt werden. Eine solche Repräsentation eines Programms hat unter anderem den Vorteil, dass sich Refactorings einfacher und zum Teil weniger fehleranfällig durchführen lassen als z. B. bei den auf dem gesamten abstrakten Syntaxbaum basierenden Refactorings der Eclipse IDE. Darüber hinaus kann die Repräsentation in Form eines Programmmodells für die statische Analyse eines Programms verwendet werden. Die aktuelle GRaViTY-Version ermöglicht z. B. bereits eine Anti-Pattern-Erkennung auf Grundlage des Quellcodes bzw. des daraus generierten Programmmodells [PKLS16]. Allerdings sind die Möglichkeiten statischer Analyseanwendungen bisher noch durch

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[2] https://cwe.mitre.org/data/definitions/20.html

das Fehlen einer geeigneten Repräsentation des Datenflusses eines Programms beschränkt. Dadurch kann eine statische Sicherheitsanalyse, welche beispielsweise das Vorhandensein von Eingabebereinigungen oder die Geheimhaltung vertraulicher Daten überprüft, das Programmmodell noch nicht optimal nutzen.

Im Rahmen der vorliegenden Arbeit wird daher, implementiert als Erweiterung des GRaViTY-Programmmodells, eine geeignete Repräsentation von Datenfluss entworfen, die für die Durchführung von Datenflussanalysen objektorientierter Programme auf einem hohem Abstraktionsniveau geeignet ist. Auf dem Modell werden anschließend konkrete Datenflussanalysen spezifiziert und praktisch umgesetzt. Die Ausarbeitung ist in 7 Kapitel unterteilt. In Kapitel 1 wird das Thema motiviert sowie ein Überblick über die Inhalte gegeben. Anschließend folgt mit Kapitel 2 ein Grundlagen-Kapitel, welches im Wesentlichen den zum Verständnis der darauffolgenden Ausführungen vorausgesetzten Kenntnisstand im Bezug auf die verwendeten Begrifflichkeiten und Konzepte darstellt. In Kapitel 3 werden zur Einordnung der vorliegenden Arbeit relevante Publikationen zu den beiden Hauptthemen Datenflussmodelle und Datenflussanalyse zusammengefasst. Im darauffolgenden Kapitel 4 wird das im Rahmen der Arbeit entwickelte Datenflussmodell sowie der Ablauf des Modellaufbaus beschrieben. Die auf dem Modell durchführbaren Datenflussanalysen werden in Kapitel 5 anhand drei konkreter Analysen demonstriert. Schließlich werden in Kapitel 6 Details zur Implementierung des vorgestellten Ansatzes sowie die Methodik und die Ergebnisse der Evaluation diskutiert. Das Fazit in Kapitel 7 rundet die Ausarbeitung durch eine Zusammenfassung der Hauptergebnisse sowie eine Einordnung der Arbeit und einen Ausblick ab.

2 Grundlagen

Um eine objektorientierte High-Level Datenflussanalyse für den Einsatz in der Domäne der Softwaresicherheit konzipieren zu können, ist ein grundlegendes Verständnis einiger relevanter Begriffe und Konzepte erforderlich. Zum erleichterten Nachvollziehen der Ausführungen in der Arbeit wird ein durchgehendes Beispiel verwendet, welches in Abschnitt 2.1 eingeführt wird. Anschließend werden in Abschnitt 2.2 für die interprozedurale Datenflussanalyse relevante objektorientierte Konzepte sowie deren Einfluss auf konkrete Analysen vorgestellt. In Abschnitt 2.3 werden die klassischen Sicherheitsanforderungen und ihre Bedeutung für die Datenflussanalyse beschrieben. Der folgende Abschnitt 2.4 führt das GRaViTY-Tool ein, welches als Basis für eine Implementierung des vorgestellten Datenflussmodells sowie einer darauf arbeitenden Datenflussanalyse dient. Abschnitt 2.5 beinhaltet die relevanten Grundlagen zum Thema Datenfluss, wobei sowohl auf etablierte Datenflussmodelle als auch auf die Analyse von Datenfluss eingegangen wird.

2.1 Durchgehendes Beispiel

Eine Airline möchte ihre Browser-basierte Online Check-in Lösung (siehe Abbildung 2.1) auf weitere Geräte portieren. Unter anderem soll mit möglichst geringem Aufwand das selbstständige Check-in an Computer-Terminals in Flughäfen ermöglicht werden. Um den Entwicklungsaufwand zu minimieren, wird der JavaScript-Code der Online Check-in Lösung innerhalb einer auf den Terminals laufenden Java-Anwendung ausgeführt. Für diesen Zweck wird eine von Java zur Verfügung gestellte JavaScript-Engine verwendet.

Zur Weiterleitung an den für das Check-in verantwortlichen JavaScript-Code werden die Eingaben allerdings innerhalb des von der JavaScript-Engine ausgewerteten Codes ungeprüft als Funktionsparameter verwendet. Code-Beispiel 2.1 zeigt, wie eine solche ungeprüfte Verwendung von Nutzereingaben aussehen könnte. Zur Vereinfachung wurden in der Darstellung Imports, Annotationen und Exception-Handling entfernt sowie die Implementierungsdetails der verwendeten externen Funktionalität ausgelassen.

Es wird zunächst ein einfaches Dialogfenster mit Eingabefeldern erstellt (siehe Zeile 22), welches die in Abbildung 2.1 dargestellten Eingabefelder enthält. Anschließend wird der bereits aus der webbasierten Check-in Anwendung vorhandene JavaScript-Code geladen (siehe Zeile 24), damit dieser zur Verarbeitung der von der grafischen Benutzeroberfläche (GUI) erfassten Eingaben verwendet werden kann. Den Feldern wird nun ein ActionListener zugeordnet, welcher beim Bestätigen der Eingabe mit der Enter-Taste diese über die JavaScript-Engine an die

Abbildung 2.1: Startseite der Check-in-Anwendung einer Airline

verarbeitenden JavaScript-Funktionen weiterleitet (Zeile 13). Obwohl an dieser Stelle vor der JavaScript-basierten Weiterverarbeitung der Nutzereingabe die Bereinigungsfunktion *sanitize* aus der geladenen JavaScript-Datei *checkIn.js* aufgerufen wird, enthält der Code hier die sicherheitsrelevante Schwäche CWE-20 (Improper Input Validation): Da keine Bereinigung auf Java-Ebene stattfindet, kann ein Angreifer innerhalb seiner Eingabe die von der Script-Engine durchgeführten JavaScript-Funktionsaufrufe beenden, die dadurch vorgesehene Eingabebereinigung und -verarbeitung umgehen und eigenen JavaScript-Code injizieren. Beispielsweise könnte ein Angreifer beide Funktionsaufrufe beenden, um mittels JavaScript eine Datei im Rootverzeichnis eines Windows-Systems auszulesen. Dies kann mit der in Listing 2.2 gezeigten Eingabe geschehen. Hier werden nach dem String `"Everything beyond here has been injected via the JavaScript engine:"` mit der Zeichenkette `'));` beide JavaScript-Funktionsaufrufe aus Zeile 13 von Listing 2.1 beendet, wodurch die folgenden Eingaben von der Script-Engine als JavaScript-Code interpretiert werden. Im dargestellten Code wird der Pfad einer geheimen Textdatei im Windows-Rootverzeichnis angegeben, welche anschließend geöffnet und vollständig ausgegeben wird.

Java-Annotationen im Quellcode können dabei helfen, solche Sicherheitsprobleme durch eine manuelle oder automatisierte Analyse zu identifizieren. So könnte der Entwickler die Methoden `getName` und `eval` mit der Annotation `@Integrity` versehen haben, um auszudrücken, dass diese Methoden die Sicherheitsanforderung Integrität (siehe Abschnitt 2.3) erfüllen und von mit ihnen interagierenden Methoden mindestens das gleiche Sicherheitsniveau fordern (siehe Listings 2.3 und 2.4). Da die Methode `getText` die Nutzereingabe des zugehörigen Textfeldes zurückgibt,

Listing 2.1: Beispielcode der Check-in Anwendung mit unzureichender Bereinigung von Nutzereingaben

```
1  public class CheckInApp {
2
3     static ScriptEngine engine = new
          ScriptEngineManager().getEngineByName("JavaScript");
4
5     private void addListenersToTextFields(JDialog dialog) {
6        for (Component comp : ((JPanel)
             dialog.getContentPane().getComponent(0)).getComponents()) {
7           if (comp instanceof JTextField) {
8              JTextField inputField = (JTextField) comp;
9              // Defining the actions to be performed, when enter is pressed
                  with the focus set on the input field
10             inputField.addActionListener(new
                  java.awt.event.ActionListener() {
11                public void actionPerformed(java.awt.event.ActionEvent
                     event) {
12                   // Passing name and input of the text field to
                        JavaScript functions from checkin.js and
                        evaluating the script
13                   engine.eval("processInput('" + inputField.getName() +
                        "', sanitizeInput('" + inputField.getText() +
                        "'));");
14                }
15             });
16          }
17       }
18    }
19
20    public static void main(String[] args) {
21       CheckInApp app = new CheckInApp();
22       JDialog dialog = CheckInGUI.createGUI();
23       // Loading the JavaScript code, which handles the check-in process
24       engine.eval("load('checkIn.js');");
25       app.addListenersToTextFields(dialog);
26       dialog.setVisible(true);
27    }
28 }
```

Listing 2.2: Eingabe mit JavaScript-Code zur Ausnutzung der Schwachstelle CWE-20 (Improper Input Validation)

```
Everything beyond here has been injected via the JavaScript engine:'));
var filename = 'C:/Secret.txt';
var reader = new java.io.BufferedReader(
new java.io.InputStreamReader(
new java.io.File(filename).toURI().toURL().openStream()));
while ((line = reader.readLine()) != null) {
print(line);
}
reader.close();//
```

erhält sie vom Entwickler die Annotation `@Tainted` zum Markieren nicht vertrauenswürdiger externer Eingaben (siehe Listing 2.5). Mithilfe einer Datenflussanalyse kann erkannt werden, dass an dieser Stelle die Rückgabe einer mit `@Tainted` annotierten Methode in das Argument einer mit `@Integrity` annotierten Methode fließt. Die statische Überprüfung von Quellcode auf solche Sicherheitsverletzungen wird in den Abschnitten 5.2 und 5.3 näher beschrieben.

Listing 2.3: Beispiel für eine Verwendung der Sicherheitsannotation `@Integrity`

```
@Integrity
public String getName() {...}
```

Listing 2.4: Beispiel für eine Verwendung der Sicherheitsannotation `@Integrity`

```
@Integrity
public Object eval(String script) throws ScriptException;
```

Die Schwäche CWE-20 kann in der beschriebenen Form allerdings nur ausgenutzt werden, wenn Javas SecurityManager nicht aktiv ist. Dieser kann jedoch durch Ausnutzen der Schwachstelle CVE-2011-3544 in der JavaScript-Engine Mozilla Rhino bis einschließlich Java Version 6u27 deaktiviert werden [Sch] [Dat]. Code-Beispiel 2.6 zeigt den auf [Rep11] basierenden Code, der unter anderem die Schwäche CWE-20 dazu verwendet, um die Schwachstelle CVE-2011-3544 auszunutzen. Auch hier wurden zur Vereinfachung Imports, Annotationen und Exception-Handling entfernt.

Bei Darstellung des in diesem Beispiel verwendeten Java Applets wird clientseitig im Webbrowser des Opfers Schadcode ausgeführt. Der Anwendungskontext unterscheidet sich daher insofern vom vorangegangen Beispiel, dass sich der Angreifer auf der Server- statt auf der Client-Seite befindet. Dennoch kann der Exploit ebenfalls im Rahmen einer clientseitigen Code-Injection verwendet werden, um den Security-Manager zu deaktivieren und somit die Ausführung von Schadcode zu ermöglichen.

Listing 2.5: Beispiel für eine Verwendung der Sicherheitsannotation `@Tainted`

```
1 @Tainted
2 public String getText() {...}
```

Zur Ausnutzung der Schwachstelle in Rhino wird ein Java Applet erzeugt, in dessen *init*-Methode eine Instanz der Rhino JavaScript-Engine erstellt wird (Zeile 10). Dieser wird zur Auswertung ein String von JavaScript-Code übergeben (Zeile 16-27), in dem eine *toString*-Methode definiert ist, die den SecurityManager deaktiviert und anschließend mithilfe einer CallBack-Methode eine Payload ausführt, deren Code sich außerhalb der Sandbox befindet. Damit die *toString*-Methode mit erhöhten Rechten ausgeführt werden kann, muss sie einem JavaScript-Fehlerobjekt zugewiesen werden (Zeile 26), welches anschließend mithilfe eines Java-Proxyobjekts in einer *JList* gerendert wird (Zeile 30-32). Das Rendering des Objekts führt schließlich dazu, dass die *toString*-Methode und somit der Schadcode ausgeführt wird.

Wie bei dem vorigen Beispiel, wird auch hier der an die *eval*-Methode übergebene String vor der Auswertung nicht bereinigt. Daher kann durch den Aufruf der *toString*-Methode des JavaScript-Fehlerobjekts der SecurityManager deaktiviert und somit die Ausführung von Schadcode ermöglicht werden.

Listing 2.6: Beispielcode des Rhino Exploits basierend auf [Rep11]

```
public class RhinoExploitExample extends Applet {
  public void init() {
    ScriptEngine se = new
       ScriptEngineManager().getEngineByName("JavaScript");
    // Creating a binding to this applet in the script engine
    Bindings b = se.createBindings();
    b.put("applet", this);

    // Evaluating the JavaScript code, which creates a new error object
       with a malicious toString-method
    Object proxy = se.eval(
         "this.toString = function() {"
              // Disabling the SecurityManager
         + "      java.lang.System.setSecurityManager(null);"
              // Using the callBack-method to run code from outside of
                 the sandbox
         + "      applet.callBack();"
              // Returning a random character for obfuscation
         + "      return String.fromCharCode(97 +
            Math.round(Math.random() * 25));"
         + "};"
         + "e = new Error();"
         + "e.message = this;"
         + "e", b);

    // Adding the resulting error object to a newly created JList
    JList<?> jl = new JList<Object>(new Object[] {proxy});
    // Adding the JList to the applet, thus ensuring its rendering and
       the execution of the error object's toString-method
    this.add(jl);
  }

  public void callBack() {
    // Running code from the Payload class, whose access would be
       prohibited by the SecurityManager
    Payload.main(null);
  }
}
```

2.2 Objektorientierte Konzepte

Die objektorientierte Programmierung ist ein Programmierparadigma, bei dem Objekte als Verbunde von Daten und Verhalten (Methoden) sowie deren Interaktion miteinander im Mittelpunkt stehen. Diese Sichtweise erlaubt das einfache Modellieren von Dingen der realen Welt, wie z. B. Personen, Autos, Akten oder auch elektronische Geräten wie Drucker. Eine Person kann beispielsweise als Objekt mit den Daten *Name*, *Alter* und *Wohnort* sowie mit Methoden zum Setzen und Abrufen der Datenfelder modelliert werden. Andere Objekte können dadurch mithilfe der Methoden z. B. das Alter einer konkreten Person abfragen oder einen geänderten Wohnort registrieren. [GS10]

Das objektorientierte Paradigma beinhaltet einige Konzepte, die für eine Datenflussanalyse objektorientierter Programme relevant sind, da sie beispielsweise einen Verlust der Eindeutigkeit aufgerufener Methoden mit sich bringen (siehe Unterabschnitt 2.2.2). Dieser führt z. B. dazu, dass ohne zusätzliche Laufzeitinformationen für alle potenziellen Aufrufziele die durch den Aufruf auslösbaren Datenflüsse analysiert werden müssen, weil jeder dieser Flüsse theoretisch zur Laufzeit auftreten kann. In den folgenden Unterabschnitten werden die relevanten objektorientierten Konzepte und ihr Bezug zur Datenflussanalyse beschrieben.

2.2.1 Datenkapselung

Datenkapselung ist das Aufteilen von Daten und Programmverhalten, das mit diesen Daten arbeitet, auf Felder und Methoden einer Klasse bzw. eines Objekts. Teil dieser Aufteilung ist das Absichern interner Daten und Methoden, die, z. B. aus Sicherheitsgründen, nicht von außerhalb des Objekts erreichbar sein sollen, durch eine Zugriffsbeschränkung. Ein unkontrolliertes direktes Zugreifen auf ein vertraulich zu behandelndes Datenfeld von Außen stellt ein großes Sicherheitsrisiko dar, da es Angreifern das Auslesen und Missbrauchen der Daten erleichtert. Eine Datenflussanalyse könnte solche und weitere unerwünschte Zugriffe auf vertrauliche Daten erkennen (siehe Abschnitt 5.3) und dem Entwickler somit die Möglichkeit geben, das Sicherheitsproblem z. B. durch das Hinzufügen einer Zugriffsbeschränkung zu beheben. Ein kontrollierter externer Zugriff auf interne Datenfelder kann beispielsweise über, durch das Objekt zur Verfügung gestellte, öffentliche Getter- (lesender Zugriff) und Setter-Methoden (schreibender Zugriff) erfolgen. Um gleichzeitig ein direktes unkontrolliertes Zugreifen auf das Feld von außerhalb des Objekts zu verbieten, wird am Anfang der Feldsignatur das Schlüsselwort `private` eingefügt. [GS10]

2.2.2 Polymorphie

In objektorientierten Programmiersprachen wie Java können Variablen und Werte mehr als einen Typ besitzen. Diese Mehrdeutigkeit von Typen wird als Polymorphie bezeichnet. Cardelli und Wegner unterscheiden vier verschiedene Arten von Polymorphie [CW85], von denen zwei für die Analyse von Datenfluss relevant sind. Diese zwei werden im Folgenden zusammengefasst.

Die **einschließende Polymorphie (*inclusion*)** hat ihre Ursache in dem objektorientierten Konzept der Vererbung. Dieses sorgt dafür, dass Objekte zu mehreren verschiedenen Klassen gehören können und der Typ eines gegebenen Objekts somit nicht garantiert eindeutig ist. Gibt beispielsweise eine Methodensignatur einen bestimmten Argumenttyp vor, kann beim Aufruf der Methode nicht nur ein Objekt genau dieser Basisklasse, sondern darüber hinaus auch eine Instanz einer Unterklasse, welche die Zugehörigkeit zur Basisklasse erbt, als Parameter übergeben werden.

Bei der **überladenden Polymorphie (*overloading*)** kann es für einen einzigen Methodenbezeichner mehrere Signaturen mit unterschiedlichen Parametertypen geben, wodurch verschiedene Varianten einer Methode unter dem gleichen Namen zur Verfügung gestellt werden können. So kann beispielsweise eine Ausgabemethode für verschiedene Datentypen unterschiedliche Implementierungen anbieten, sodass z. B. bei Übergabe einer Zeichenkette diese auf andere Weise ausgegeben wird als eine Ganzzahl.

2.2.3 Java-spezifische Umsetzung von Polymorphie

Die in 2.2.2 beschriebenen Konzepte werden in objektorientierten Programmiersprachen teilweise unterschiedlich umgesetzt. Da in dieser Arbeit Java-Programme Gegenstand statischer Analysen sein sollen, wird im Folgenden die Java-spezifische Umsetzung der beschriebenen Polymorphiearten veranschaulicht.

Methodenüberschreibung (*method overriding*)

Das Überschreiben von Methoden (*method overriding*) stellt eine konkrete Art einschließender Polymorphie (*inclusion*) dar. In Java können Methoden, die in einer Oberklasse definiert wurden, in Unterklassen überschrieben werden, um eine spezifische Implementierung angeben zu können, die von dem in der Oberklasse angegebenen allgemeineren Verhalten abweicht. Bei einem Aufruf der Methode auf einem Objekt der Unterklasse, wird, auch wenn der statische Typ der Objektreferenz die Oberklasse ist, die überschreibende Methode der Unterklasse ausgeführt. Dies ist der Fall, weil die Entscheidung über das Ziel des Aufrufs bei Methodenüberschreibung erst dynamisch zur Laufzeit gefällt wird, wenn konkrete Objekte vorliegen, und folglich genauere Informationen über den Typ eines referenzierten Objekts bekannt sind, als zur Übersetzungszeit. Dementsprechend können lediglich die auf konkreten Objekten aufgerufenen Instanzmethoden überschrieben werden. Im Falle von Klassenmethoden tritt bei der erneuten Definition einer bereits in der Oberklasse definierten Methode in einer Unterklasse die sogenannte Methodenverdeckung (*method hiding*) auf. Diese wird im nächsten Paragraphen dieses Unterabschnitts beschrieben. [GJS+19]

In Listing 2.7 ist demonstriert, wie sich die Methodenüberschreibung praktisch auswirkt. Die in Abschnitt 2.1 beschriebene Check-in Anwendung könnte z. B. neben der in Listing 2.1 dargestellten Implementierung mehrere erweiternde Implementierungen besitzen, die entsprechend von der Klasse *CheckInApp* erben. Die Methode *getCurrentAppInstance* gibt ein Objekt dieser Klasse oder einer ihrer Unterklassen zurück. Anschließend wird die in der Basisklasse *CheckInApp* definierte Methode *run* auf dem erhaltenen Objekt aufgerufen, welche in den Unterklassen überschrieben

worden sein könnte, wodurch mehrere Implementierungen zu dieser Methode existieren können. Da erst zur Laufzeit bekannt ist, ob *app* ein Objekt der Basisklasse *CheckInApp* oder einer Unterklasse referenziert, ist der Typ der von *app* referenzierten Instanz und somit auch der Aufruf der *run*-Methode zur Übersetzungszeit mehrdeutig. Hierdurch muss bei einer statischen Datenflussanalyse jeder durch die gegebene Vererbungshierarchie mögliche Datenfluss betrachtet werden. Eine präzise Bestimmung der effektiven Datenflüsse ist allein mit statischen Information bzw. zur Übersetzungszeit nicht möglich, da erst zur Laufzeit konkrete Instanzen und deren Typen bekannt sind.

Listing 2.7: Beispiel für einschließende Polymorphie in Java

```
1 ...
2 CheckInApp app = getCurrentAppInstance();
3 // CheckInApp has multiple subclasses and getCurrentAppInstance() could
     also return an instance of any of these classes
4 app.run();
5 ...
```

Methoden- und Feldverdeckung (*method/field hiding*)

Wird für eine Klassenmethode in einer Unterklasse eine weitere Definition angegeben, hat dies *method hiding* zur Folge, welches ebenfalls als Java-spezifische Variante von einschließender Polymorphie angesehen werden kann. Hierbei wird die Methode der Oberklasse nicht überschrieben und bei einem Methodenaufruf lediglich der statische Typ der für den Aufruf verwendeten Objektreferenz (im Gegensatz zum erst zur Laufzeit bestimmbaren Objekttyp bei *method overriding*) herangezogen, um das Aufrufziel zu bestimmen. Analog tritt bei in Unterklassen neu definierten Feldern einer Oberklasse *field hiding* auf. Auch hierbei wird das Ziel eines Zugriffs rein statisch bestimmt, d. h. der dem Feld bei der Deklaration zugewiesene statische Typ ist ausschlaggebend. [GJS+19]

Bei einer Datenflussanalyse, bei der statische Typinformationen der analysierten Programme verwendet werden können, gibt es keine Mehrdeutigkeit von Aufruf- bzw. Zugriffszielen, wenn *hiding* vorliegt. Entsprechend ist keine Überapproximation der betrachteten Flüsse, die zwangsläufig zu einem Präzisionsverlust führt, notwendig. Liegen bei der Analyse keine statischen Typinformationen vor, muss jede Definition der aufgerufenen Methode bzw. des Feldes, auf das zugegriffen wird, als mögliches Ziel in Betracht gezogen werden und die dadurch potenziell anfallenden Flüsse in die Datenflussanalyse aufgenommen werden.

In Listing 2.8 ist die Behandlung von *field hiding* in Java veranschaulicht. Erneut wird über *getCurrentAppInstance* eine Instanz von *CheckInApp* oder einer Unterklasse abgerufen und der Variable *app* zugewiesen. Anschließend wird auf das Feld *name* des in *app* referenzierten Objekts zugegriffen und der Feldinhalt ausgegeben. Da bei *hiding* lediglich der statische Typ verwendet wird, um zu bestimmen, welches Zugriffsziel ein konkreter Zugriff hat, führt der Beispielcode dazu, dass unabhängig

von der Rückgabe der Methode *getCurrentAppInstance* immer auf das *name*-Feld der Basisklasse *CheckInApp* zugegriffen wird. Das Beispiel verdeutlicht, dass in Java Zugriffe im Kontext von *hiding* ausschließlich mithilfe statischer Typinformationen getätigt werden.

Listing 2.8: Beispiel für einschließende Polymorphie in Java

```
1 ...
2 CheckInApp app = getCurrentAppInstance();
3 System.out.println(app.name);
4 ...
```

Methodenüberladung (*method overloading*)

Das *method overloading* ist die Java-spezifische Variante der überladenden Polymorphie, welche im Allgemeinen wie die in Unterabschnitt 2.2.2 beschriebene überladende Polymorphie funktioniert. Bei einem Methodenaufruf wird zur Übersetzungszeit basierend auf den statischen Typen der übergebenen Argumente bestimmt, welche konkrete Signatur aufgerufen wird. Eine Datenflussanalyse, die auf statische Typinformationen der Felder in den analysierten Programmen zurückgreifen kann, muss entsprechend keine Überapproximation möglicher Flüsse aufstellen, da diese Form von Polymorphie in diesem Fall keine Mehrdeutigkeit in der Programmsemantik mit sich bringt. Stehen jedoch keine statischen Typinformationen zur Verfügung, muss jede Signatur als potenzielles Aufrufziel in Betracht gezogen werden, deren Parameterzahl der Anzahl der an die Methode übergebenen Argumente entspricht. In einer Datenflussanalyse müssten entsprechend die von jedem dieser möglichen Aufrufziele auslösbaren Datenflüsse analysiert werden. [GJS+19]

Listing 2.9: Beispiel für überladende Polymorphie in Java

```
1  void processTicketID(ShortID id) {...}
2
3  void processTicketID(LongID id) {...}
4
5  ...
6
7  ShortID sID = new LongID();
8  processTicketID(sID); // Invokes 1st method signature
9  LongID lID = new LongID();
10 processTicketID(lID); // Invokes 2nd method signature
```

In Listing 2.9 ist veranschaulicht, wie eine Methodenüberladung in Java aussehen kann und wie polymorphe Aufrufe gehandhabt werden. Die Methode *processTicketID* wird für die beiden Parametertypen *ShortID* und *LongID* definiert. Die Klasse

LongID erweitert darüber hinaus die Klasse *ShortID*. Im Beispielcode wird je eine Instanz von *LongID* erstellt, jedoch zunächst mit dem statischen Typ *ShortID* und anschließend mit dem statischen Typ *LongID*. Beide Objektreferenzen werden im Anschluss als Argument an einen Aufruf der Methode *processTicketID* übergeben. Da die aufgerufene Methodensignatur basierend auf dem statischen Typ der übergebenen Argumente identifiziert wird, ist das Aufrufziel des ersten Aufrufs von *processTicketID* die Signatur mit dem Parametertyp *ShortID*, obwohl das übergebene Argument *sID* zur Laufzeit eine Instanz der Klasse *LongID* beinhaltet. Beim zweiten Aufruf lautet der statische Typ *LongID*, weshalb die Signatur mit diesem Argumenttyp aufgerufen wird. Das Beispiel verdeutlicht, dass polymorphe Aufrufe überladener Methoden in Java ausschließlich anhand statischer Typinformationen aufgelöst werden.

2.3 Sicherheitsanforderungen

Die klassischen Anforderungen der IT-Sicherheit lauten *Vertraulichkeit*, *Integrität* und *Verfügbarkeit*. Vertraulichkeit bedeutet, dass geheime Daten vor dem Auslesen durch Unbefugte geschützt werden müssen. Die Integrität von Daten ist gegeben, wenn keine unbefugten Manipulationen daran vorgenommen wurden. Verfügbarkeit ist die Forderung, dass ein System bzw. eine Ressource für seine potenziellen Verwender zur Nutzung verfügbar ist. [VK83]

Für die Datenflussanalyse eines Programms sind die Vertraulichkeit und Integrität der Daten relevant, da diese durch einen unerlaubten Datenfluss verletzt werden können. Ein Angreifer könnte beispielsweise das vom Entwickler als vertraulich gekennzeichnete Feld `PEPPER` (siehe Listing 2.10), durch das Ausnutzen von Schwachstellen auslesen und die dadurch erhaltenen vertraulichen Daten missbrauchen. Wie in Listing 2.4 zu sehen, wurde die Methode *eval* vom Entwickler ebenfalls annotiert. Da es sich hierbei um eine Methode handelt, die Integrität fordert, sollten keine Daten aus nicht vertrauenswürdigen Quellen in die Methode fließen. Durch eine Analyse des Programmdatenflusses könnte die Möglichkeit solcher Angriffe erkannt und verhindert werden, da ausgehende Flüsse aus vertraulichen Feldern in nicht vertrauenswürdige Member (Methoden- und Felddefinitionen) sowie eingehende Flüsse aus nicht vertrauenswürdigen Quellen in Member, die Integrität fordern, gefunden werden könnten.

Listing 2.10: Beispiel für eine Verwendung der Sicherheitsannotation `@Secrecy`

```java
@Secrecy
static final byte[] PEPPER = createRandomByteArray();
```

2.4 GRaViTY

GRaViTY [PKLS15] ist ein Tool, das mehrere den Softwareentwicklungsprozess unterstützende Funktionen zur Verfügung stellt. Beispielsweise enthält das Tool

sowohl eine Anti-Pattern-Erkennung als auch die Möglichkeit zu deren Behebung [PKLS16]. Als Grundlage für die Analysen und Programmtransformationen, die das Tool ausführt, wird zu einem gegebenem Java-Code ein leichtgewichtiges Programmmodell erzeugt. Dieses wird durch eine Reduktion sowie Erweiterung des abstrakten Syntaxbaums erstellt und enthält lediglich die Konstrukte objektorientierter Programme, die für die vorgesehenen Funktionen des Tools von Relevanz sind. Dadurch kann die Komplexität ausgewählter Quellcode-Operationen, wie z. B. Refactoring, im Vergleich zu den in gängigen IDEs eingesetzten Implementierungen erheblich reduziert werden [PKLS15]. In den folgenden Unterabschnitten werden Details zum von GRaViTY aufgebauten Programmmodell (Unterabschnitte 2.4.1 und 2.4.2) sowie der für die Modelltransformation im Rahmen des Modellaufbaus verwendeten *Triple-Graph-Grammatik* (*TGG*) vorgestellt.

2.4.1 Modellspezifikation

Abbildung 2.2 zeigt das Metamodell des von GRaViTY erstellten Programmmodells (im Modell als *TypeGraph* bezeichnet) in einer auf die wesentlichen Modellelemente reduzierten Form. In jedem Programmmodell können beliebig viele Pakete (*TPackage*) modelliert werden, welche wiederum eine beliebige Anzahl von Klassen beinhalten können. Je Klasse werden alle Member sowie zugehörige Signaturen und deren Parameter modelliert. Zu jeder Feldsignatur wird im Modell der statische Typ vorgehalten (siehe Assoziation *type* zwischen *TFieldSignature* und *TAbstractType*). Dieser ist somit in statischen Analyseanwendungen, die mit Instanzen des Metamodells arbeiten, verwendbar. Hierdurch kann beispielsweise eine Mehrdeutigkeit von Aufrufen bzw. Zugriffen durch das in Unterabschnitt 2.2.3 beschriebene *hiding* verhindert werden. Da eine Methode mit einem Bezeichner mehrere unterschiedliche Signaturen haben kann (Überladung), existiert zusätzlich die Klasse *TMethod*, die eine Methode anhand ihres Bezeichners identifiziert. Analog dazu gibt es eine Klasse *TField*, die Felder anhand ihres Bezeichners identifiziert, da ein Feld aufgrund von Hiding ebenfalls mehrere Signaturen haben kann. Wesentlich für das im Rahmen dieser Arbeit angestrebte Hinzufügen interprozeduraler Datenflussinformationen ist das Vorhandensein von Informationen über Zugriffe (*TAccess*) zwischen Membern, da jeder Zugriff zu einem interprozeduralen Datenfluss führen kann. Die abgeleitete konkrete Flussart *TCall* modelliert Methodenaufrufe, während Zugriffe auf Felder über die Basisklasse *TAccess* dargestellt werden. Um im Quellcode z. B. zur Angabe von Sicherheitsinformationen hinzugefügte Java Annotationen (siehe Code-Beispiel 2.11) zu modellieren, besitzt jedes annotierbare Modellelement ein Attribut *tAnnotation*, welches die ihm zugeordneten Annotationen referenziert.

Listing 2.11: Verwendung von Java Annotationen zum Hinzufügen von Sicherheitsinformationen

```
1 @Secrecy
2 int secretNumber = 7;
```

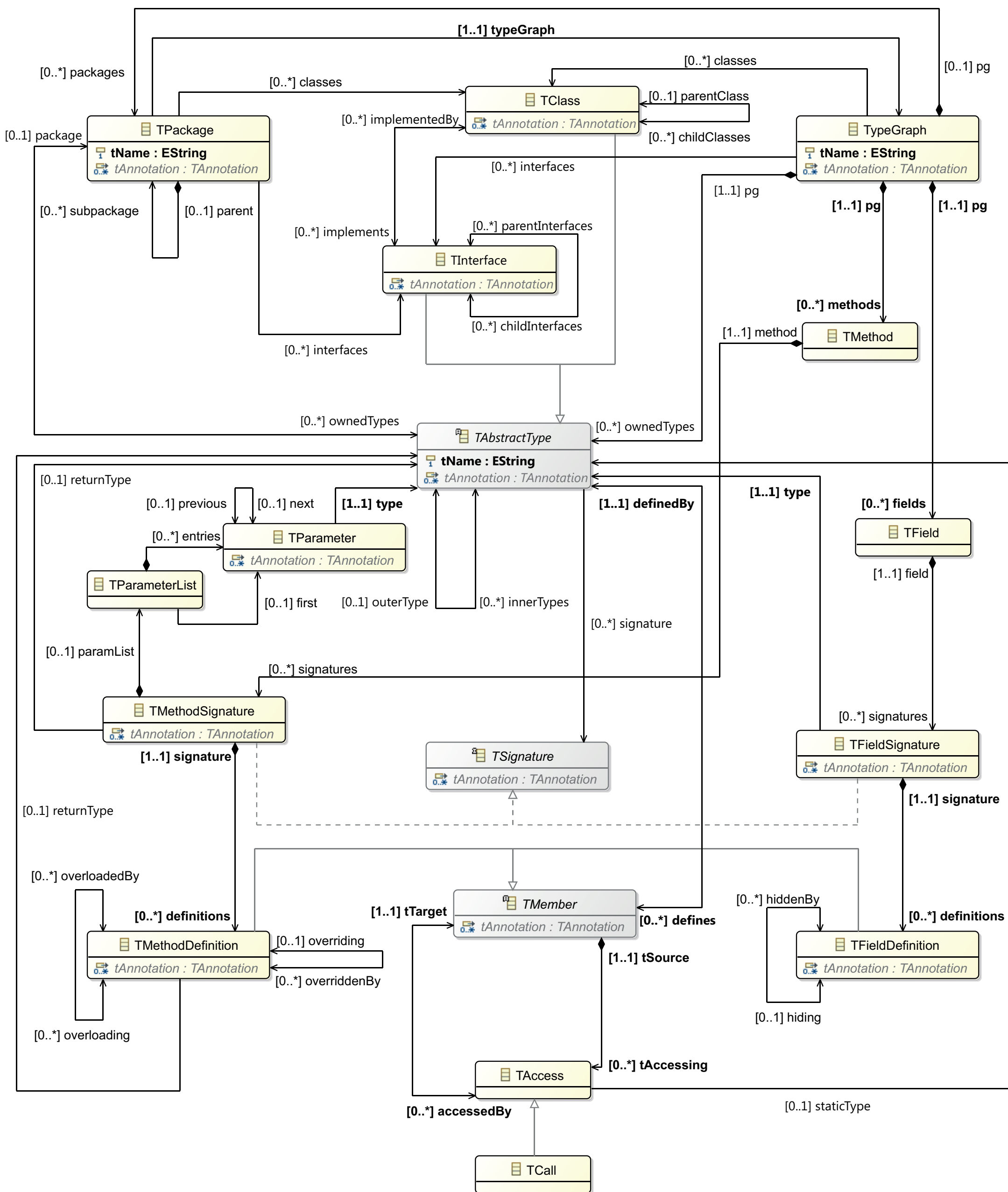

Abbildung 2.2: Metamodell des von GRaViTY erstellten Programmmodells

Abbildung 2.3 zeigt einen Ausschnitt aus dem Modell zu Listing 2.1, in dem eine umgehbare Bereinigung von Nutzereingaben in einer Check-in Anwendung veranschaulicht ist. Es ist der Ausschnitt des Code-Beispiels visualisiert, der die Sicherheitsschwäche beinhaltet (Definition der Methode *actionPerformed*). Die Methode *actionPerformed* wird in einer anonymen Klasse, welche das Interface *ActionListener* implementiert, definiert (siehe Mitte des oberen Teils der Abbildung). Im Methodenkörper werden mehrere Zugriffe auf weitere Member getätigt. So wird zunächst auf das Feld *engine* der Klasse *CheckInApp* zugegriffen (siehe rechte Seite des mittleren Teils der Abbildung), um die referenzierte *ScriptEngine* zu erhalten und deren Methode *eval* aufzurufen (siehe Mitte des unteren Teils der Abbildung). In dem an *eval* übergebenen Argument vom Typ *String*, welches den auszuwertenden JavaScript-Code darstellt, werden die Rückgabewerte der Aufrufe der beiden Methoden *getText* und *getName* aus der Klasse *JTextField* (siehe linker Teil der Abbildung) verwendet. Die dargestellte Modellinstanz wird in Kapitel 4.2 um Datenflüsse sowie konkrete Zugriffsarten für Felder ergänzt und geringfügig angepasst, um die spezifischen Anforderungen eines Datenflussmodells zu erfüllen.

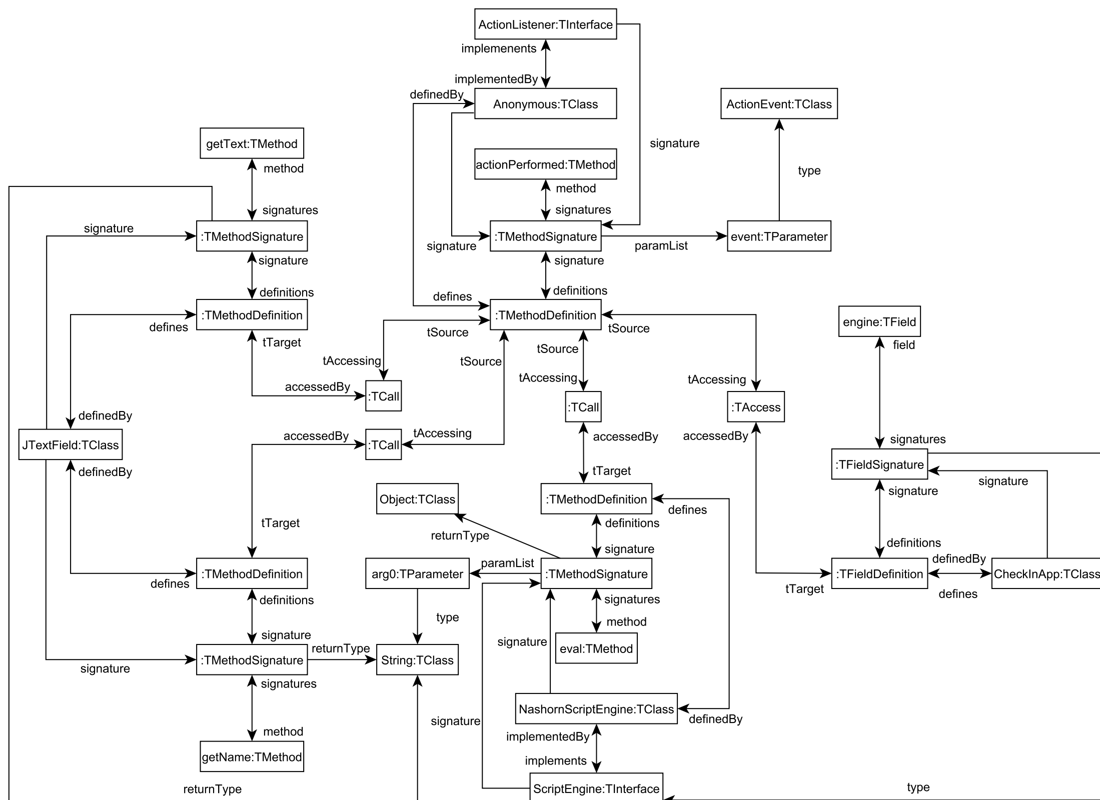

Abbildung 2.3: Beispielinstanz eines von GRaViTY erstellten Programmmodells

2.4.2 Modellaufbau

Der Aufbau der beschriebenen Programmmodellform des GRaViTY-Tools ist in Abbildung 2.4 dargestellt und wird im Folgenden beschrieben. Einige Schritte sind

mit ”vorwärts” oder ”rückwärts” gekennzeichnet, um anzugeben, ob diese zu der Vorwärtstransformation vom Quellmodell (hier: AST) zum Zielmodell (hier: Programmmodell) oder bei der entgegengesetzten Rückwärtstransformation vom ursprünglichen Zielmodell zum ursprünglichen Quellmodell durchgeführt werden. Zum Quellcode eines beliebigen korrekten Java-Programms wird zunächst mithilfe des Software-Modernisierungstools MoDisco[1] der vollständige abstrakte Syntaxbaum (AST) aufgebaut (Schritt 1). Anschließend werden auf dem AST Vorverarbeitungen durchgeführt (Schritt 2), welche diesen erweitern und in eine für nachfolgende Verarbeitungsschritte geeignete Form bringt. Eine anschließende Modelltransformation (Schritt 3) mithilfe von Triple-Graph-Grammatik-Transformationsregeln (siehe Unterabschnitt 2.4.3) sorgt einerseits für eine Reduktion von in Analysen nicht benötigten Details des ASTs und andererseits für das Hinzufügen von aus dem AST ableitbaren impliziten Informationen, die den Analysen wiederum dienlich sind.

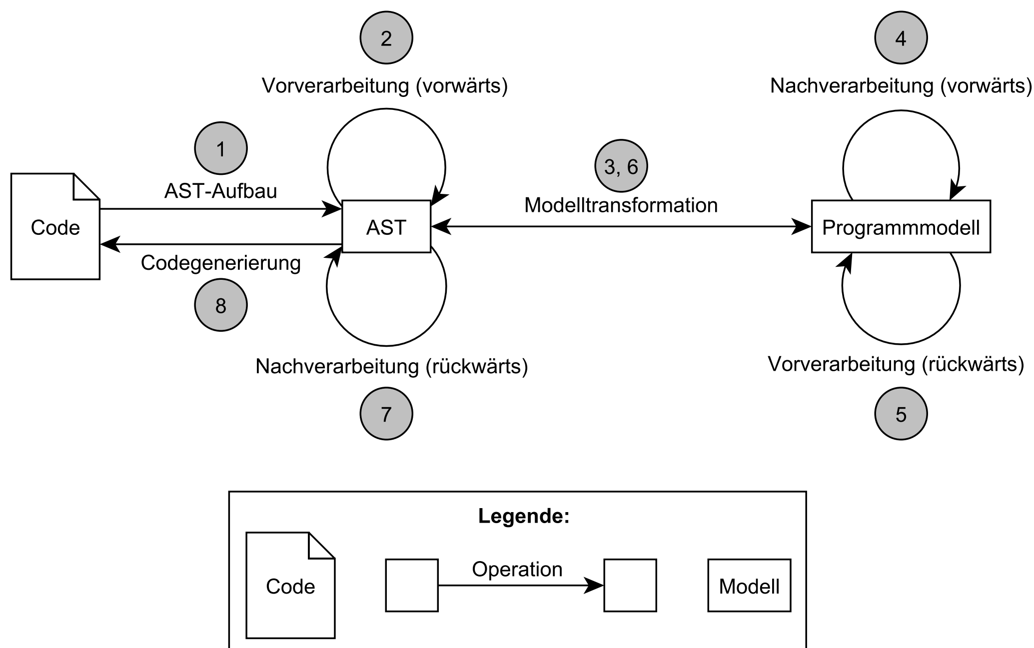

Abbildung 2.4: Allgemeiner Ablauf des Programmmodellaufbaus in GRaViTY

Das hierbei entstehende Programmmodell ist schließlich eine Instanz des in Abbildung 2.2 dargestellten Metamodells. In weiteren Nachverarbeitungen (Schritt 4) können für konkrete Analysen oder Operationen notwendige Detailanpassungen des Modells durchgeführt werden. Änderungen, die anschließend direkt auf dem Programmmodell getätigt werden (z. B. Refactorings, vgl. [PKLS15]), können durch eine Rücktransformation in den AST und schließlich auch in den Quellcode zurückpropagiert werden (Schritt 6). Davor ist es nötig, dem Programmmodell hinzugefügte

[1] https://www.eclipse.org/MoDisco/

Elemente, die kein Pendant im AST haben, zu entfernen. Zu diesem Zweck werden Vorverarbeitungen auf dem Programmmodell durchgeführt (Schritt 5). Nach erfolgter Rücktransformation kann der AST per Nachverarbeitung (Schritt 7) in eine für die Codegenerierung (Schritt 8) geeignete Form gebracht werden, sodass die durch Operationen auf dem Programmmodell entstandenen Änderungen schließlich auch im Code sichtbar werden.

2.4.3 Triple-Graph-Grammatik (TGG)

Die Triple-Graph-Grammatik ist eine Graphgrammatik, in der zwei Graphmodelle über ein drittes Korrespondenzmodell miteinander verbunden sind. Durch die Herstellung einer Korrespondenz zwischen den beiden Graphmodellen ist eine Modelltransformation in beide Richtungen möglich. Ihr Ablauf wird durch die Anwendung von Regeln bestimmt. Eine TGG-Regel definiert immer eine oder mehrere Vor- und Nachbedingungen sowie Korrespondenzen zwischen den in den Bedingungen verwendeten Knoten. Bei der Modelltransformation wird die Regel angewandt, falls die Vorbedingungen erfüllt sind und es wird gemäß der Korrespondenzen an den entsprechenden Stellen im Zielmodell der in den Nachbedingungen spezifizierte Zustand hergestellt. [Sch94] [ABD+15]

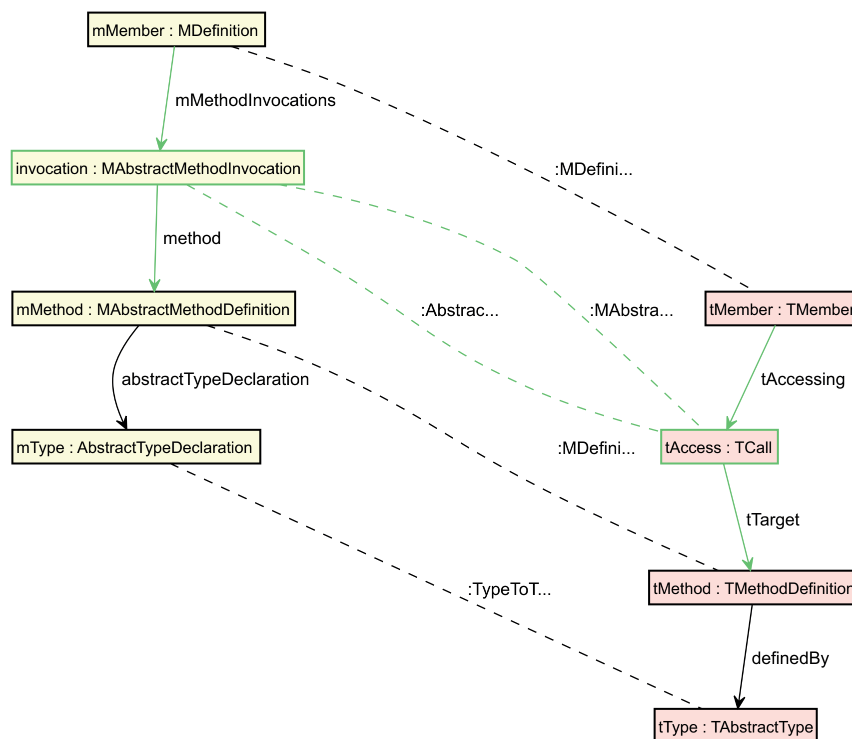

Abbildung 2.5: Beispiel einer TGG-Regel zum Transformieren von Feldzugriffen

In Abbildung 2.5 ist ein Beispiel für eine TGG-Regel des GRaViTY-Tools zur Transformation von Feldzugriffen dargestellt, die im Rahmen dieser Arbeit erweitert wird. Die gelb eingefärbten Knoten auf der linken Seite sowie ihre Verbindungen stellen dabei die zu erfüllende Vorbedingung in Form eines Graphmusters dar. Wird dieses Graphmuster bei der Modelltransformation im Quellgraph identifiziert, so

wird im Zielgraph das Muster der rot eingefärbten Knoten auf der rechten Seite der Regel eingefügt bzw. gematcht, sodass die hierdurch spezifizierte Nachbedingung erfüllt ist. Durch die gestrichelten Kanten zwischen den Knoten beider Seiten geben wird angegeben, welche Regel für die Korrespondenzbeziehung anzuwenden ist. Die obere Korrespondenz trägt z. B. den Bezeichner *MDefinitionToTMember* und gibt an, dass eine *MDefinition* im Quellmodell einen *TMember* als zugehöriges Element im Zielmodell besitzt. Entsprechend wird zur *MDefinition* der Quellseite, bei Anwendung der Regel, im Zielgraph ein *TMember* eingefügt bzw. gematcht. Die grünen Hervorhebungen von Knoten und Kanten bedeuten, dass diese Elemente neu hinzugefügt bzw. angewandt werden. Alle anderen Teile werden lediglich als Kontext der Regelanwendung gematcht und müssen dementsprechend bereits durch vorige Regeln erstellt worden sein.

In GRaViTY werden TGG-Regeln mithilfe des Modelltransformationstools eMoflon[2] eingesetzt, um den vom MoDisco-Tool aufgebauten, in Pre-Processings verarbeiteten AST in das in Unterabschnitt 2.4.1 spezifizierte Programmmodell zu transformieren und umgekehrt. Der TGG-Regelsatz wird in dieser Arbeit um Regeln zur Übersetzung von Datenflüssen und konkreten Feldzugriffen erweitert. Ein Beispiel einer hinzugefügten Regel ist in Abschnitt 6.2 beschrieben.

2.5 Datenfluss

Das in 2.4 vorgestellte Programmmodell von GRaViTY enthält bisher keine Informationen über Datenfluss. Allerdings fließen in nahezu allen Programmen zur Laufzeit Daten, deren Vertraulichkeit und Integrität erforderlich sein kann. Eine Datenflussanalyse zur Prüfung dieser Eigenschaften auf Basis des Programmmodells ist daher in vielen Fällen sinnvoll. Ein einzelner Datenfluss beginnt an einer Quelle und endet in einer Senke. Eine Quelle ist demnach ein Punkt im betrachteten System oder Teilsystem, an dem es von Daten betreten wird, während eine Senke ein Punkt ist, an dem die Daten das System verlassen. [EGH+14]

Zur Modellierung des Datenflusses existieren bereits etablierte Darstellungen, die im folgenden Abschnitt 2.5.1 beschrieben werden. Diese werden beispielsweise beim Datenfluss-basierten Testen zur Bestimmung der Testabdeckung verwendet [Rie13]. Im darauffolgenden Abschnitt 2.5.2 werden grundlegende Formen der Datenflussanalyse beschrieben.

2.5.1 Datenflussdarstellung

Zwei verbreitete Datenflussdarstellungen sind der *Datenflussgraph* (*DFG*) und das *Datenflussdiagramm* (*DFD*). Ein *Datenflussgraph* ist in Abbildung 2.6 für die Methodendefinition von *actionPerformed* aus dem Beispiel in Listing 2.1 dargestellt. Der Datenflussgraph zeichnet sich dadurch aus, dass jede einzelne Operation bzw. Methode sowie alle Variablen, Konstanten und Datenflüsse zwischen diesen enthalten sind (vgl. [DK82]). Somit besitzt diese Darstellungsform eine besonders hohe Granularität. Der dargestellte Beispielgraph umfasst z. B. lediglich eine Codezeile,

[2] https://emoflon.org

da die Methodendefinition von *actionPerformed* nur diese Zeile enthält. Trotzdem sind bereits je neun Knoten und Datenflusskanten enthalten.

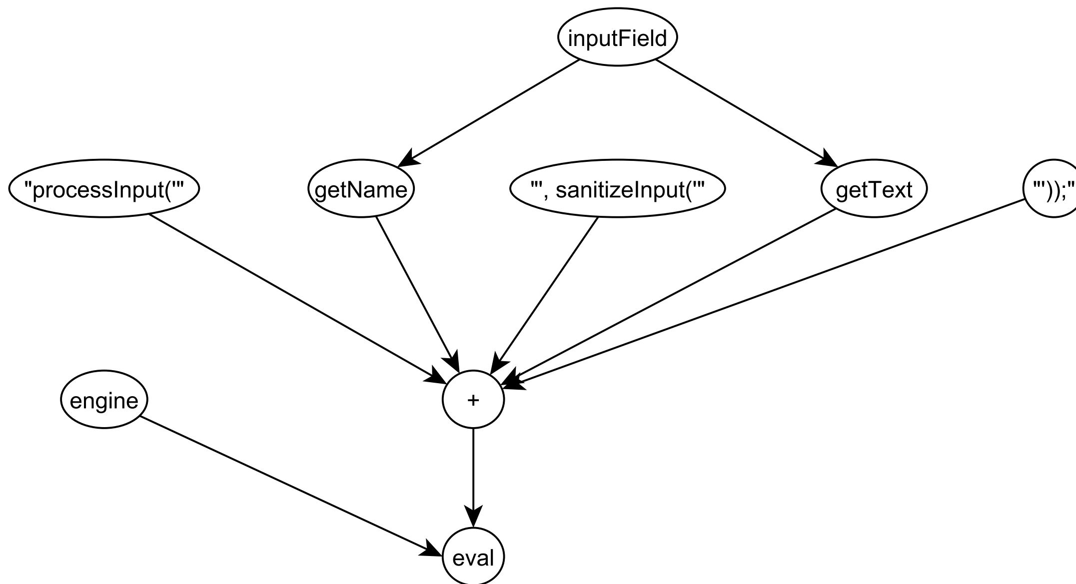

Abbildung 2.6: Datenflussgraph für die Definition der Methode *actionPerformed*

Der Datenflussgraph besitzt eine Granularität, welche für die im Rahmen dieser Arbeit durchgeführten Datenflussanalysen (siehe Kapitel 5) nicht nötig und eher hinderlich ist, da hierdurch die Verständlichkeit des Modells bzw. Graphs sowie dementsprechend auch der darauf durchgeführten Analysen und deren Ergebnisse reduziert wird. Zu der Erkenntnis, dass die Verständlichkeit von Modellen durch den Anstieg der Modellgröße leidet, sind bereits [Stö18] und [MRC07] im Rahmen einer Untersuchung von UML-Diagrammen bzw. Prozessmodellen gekommen. Da eine höhere Granularität gleichzeitig auch einen Anstieg der Modellgröße bedeutet, ist diese Erkenntnis auf Datenflussgraphen übertragbar. Die Datenflussanalysen basieren auf der Identifikation von Datenflussmustern mithilfe einer einfachen Anfragesprache. Die Einfachheit dieser Anfragesprache kann nur gewährleistet werden, wenn die zu definierenden Muster und dementsprechend auch das zu untersuchende Modell ebenfalls einfach aufgebaut ist. Da der Datenflussgraph viele Informationen enthält, die für den Datenfluss zwischen Membern objektorientierter Programme (interprozeduraler Datenfluss) nicht relevant sind, eignet sich dieser nicht als Datenflussmodell für die vorgesehenen Analysen. Insbesondere, da der herkömmliche DFG keinen Kontrollfluss beinhaltet, wäre bei dessen Verwendung als Grundlage für interprozedurale Datenflussanalysen eine wesentliche Information nicht vorhanden. Ein interprozeduraler Datenfluss geht immer auf einen Variablenzugriff oder einen Methodenaufruf zurück. Da Variablenzugriffe und Methodenaufrufe im DFG allerdings nicht explizit enthalten sind, eignet sich dieser auch aus diesem Grund nicht für die vorgesehenen Analysen. In Kombination mit Kontrollfluss eignen sich Datenflussgraphen z. B. für die Modellierung paralleler Systeme und Berechnung [KBB86]. Darüber hinaus wird beim datenflussbasierten Testen eine Kombination aus Daten- und Kontroll-

flussgraph verwendet, um Testfälle auszuwählen, die eine möglichst hohe Abdeckung der Datenflüsse eines Programms oder Programmabschnitts garantieren [Rie13]. Ein Beispiel für die Verwendung von Datenflussgraphen ohne die Hinzunahme von Kontrollfluss wird in [DK82] vorgestellt. Hier werden DFGs zur Modellierung rein funktionaler Programme verwendet.

Während sich der Datenflussgraph durch eine besonders feine Granularität auszeichnet, ist das von DeMarco in [DeM79] vorgestellte *Datenflussdiagramm* eher zur Planung sehr grober Datenflüsse in frühen Phasen des Systementwurfs geeignet. Abbildung 2.7 stellt ein Datenflussdiagramm für die Weiterleitung der Nutzereingaben von der Nutzerschnittstelle aus Abbildung 2.1 zur Buchungsdatenbank dar. Da ein Datenflussdiagramm noch sehr stark von einer konkreten Implementierung abstrahiert, können in diesem eher konzeptionelle Fehler gefunden werden. Beispielsweise scheint an dieser Stelle noch keine Validierung bzw. Bereinigung der vom Nutzer getätigten Formulareingaben vor der Evaluierung und der damit verbundenen Weiterleitung an die Buchungsdatenbank vorgesehen zu sein, da die Formulareingaben direkt vom Knoten *getDataFromUI* zum Knoten *eval* fließen. Da tatsächliche Datenflussanomalien, die Sicherheitsverletzungen darstellen können, jedoch erst basierend auf der Implementierung sicher identifiziert werden können, reicht die Granularität des Datenflussmodells nicht aus. Daher wird mit dem in Abschnitt 4.1 spezifizierten Datenflussmodell bezüglich der Granularität ein Mittelweg zwischen dem feingranularen Datenflussgraphen und dem grobgranularen Datenflussdiagramm umgesetzt.

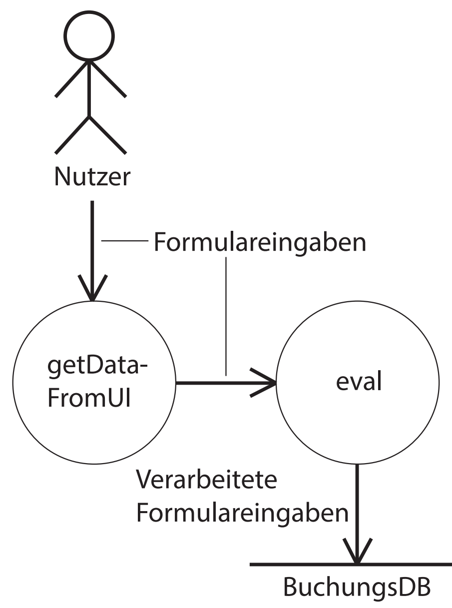

Abbildung 2.7: Datenflussdiagramm zur Weiterleitung von Nutzereingaben an die Buchungsdatenbank

2.5.2 Datenflussanalyse

Das zentrale gemeinsame Konzept aller Datenflussanalysen ist die *Erreichbarkeit* von Programmelementen oder Codeblocks (Sequenz von Anweisungen). Im Allgemeinen ist ein Codeblock von einem Programmelement x, wie z. B. der Definition eines Datenobjekts, *erreichbar*, falls das x in einem Codeblock definiert wurde, der im Kontrollfluss vor dem zu erreichenden Block auftritt, und mindestens einen Pfad zu diesem Block besitzt. Hierbei muss die Einschränkung gelten, dass x auf dem Pfad nicht neu definiert wird. [AC76]

Die allgemeine Definition kann auf die in dieser Arbeit durchgeführte interprozedurale Datenflussanalyse von Java-Programmen übertragen werden. Der Erreichbarkeitsbegriff wird dabei auf Datenflüsse zwischen Membern angewandt. Entsprechend ist ein Member x von einem Member y *erreichbar*, falls mindestens ein Datenflusspfad von x zu y existiert. Die Einschränkung, dass keine Neudefinition auf dem Pfad stattfindet, wird in dieser Arbeit außer Acht gelassen, da nur eine grundsätzliche Erreichbarkeit zwischen Membern, abstrahiert von konkreten Daten, von Interesse ist. Diese ist ausreichend, um statische Sicherheitsanalysen, beispielsweise zur Identifikation von Datenlecks, durchführen zu können.

3 Related Work

Da im Rahmen dieser Masterarbeit ein Datenflussmodell aufgebaut und anschließend eine Datenflussanalyse darauf durchgeführt wird, werden im Folgenden einige relevante Ansätze zu diesen beiden Themen aus der Literatur beschrieben und deren Bedeutung für das Thema der Arbeit herausgestellt.

3.1 Datenflussmodelle

A Formal Definition of Data Flow Graph Models [**KBB86**] In diesem Papier wird eine formale Definition eines Modells für parallele Systeme und Berechnung vorgestellt, welche einen Datenflussgraph darstellt. Dieser hat die Besonderheit, dass zur Vermeidung von Deadlocks neben Datenfluss auch Kontrollfluss modelliert wird und begründet dadurch die Nützlichkeit einer Hinzunahme von Kontrollfluss in Datenflussmodellen, welche bei dem im Rahmen der Arbeit erstellten Modell ebenfalls vorgesehen ist.

Soot Compiler Framework [**LBLH11**] Das Soot Framework dient der Analyse von Java-Programmen. Hierzu werden die Programme in eine leicht zu analysierende und optimierende Form übersetzt, einen *Three-Address Code* namens *Jimple*. Dieser zeichnet sich dadurch aus, dass jedes Statement des Java Bytecodes auf max. 3 Operanden reduziert ist. Diese Repräsentationsform von Java-Programmen ist deutlich detaillierter als die im Rahmen der Arbeit entwickelte Repräsentation, da jedes Statement nach der Übersetzung erhalten bleibt und keine für die Datenflussanalyse irrelevanten Details ausgelassen werden. Soot bietet weiterhin die Möglichkeit, Datenflussanalysen auf dem erzeugten Jimple Code durchzuführen. Hierzu kann z. B. das Tool Heros verwendet werden, welches in Abschnitt 3.2 beschrieben wird.

3.2 Datenflussanalyse

Heros Tool [**Bod**] Heros[1] ist ein Tool, mit dem interprozedurale Datenflussanalyseprobleme gelöst werden können. Es implementiert dafür die Methoden aus [RHS95] bzw. [SRH96] zum Lösen von *Interprocedural, Finite, Distributive Subset* (IFDS) bzw. *Interprocedural, Distributive Environment* (IDE) Problemen. Zur Verwendung des Tools muss ein interprozeduraler Kontrollflussgraph sowie eine Definition eines IFDS bzw. IDE Problems gegeben sein. Aufgrund des hiermit verbundenen Mehraufwands und der Notwendigkeit von Kenntnissen auf dem Gebiet der Datenflussanalyse, wird eine Verwendung des Tools für die in dieser Arbeit durchgeführten

[1] https://github.com/Sable/heros

Analysen nicht in Betracht gezogen.

Precise interprocedural dataflow analysis via graph reachability [RHS95] Die in diesem Papier vorgestellte Methode ermöglicht das Lösen von *Interprocedural, Finite, Distributive Subset)* (IFDS) Problemen zum Zwecke einer Datenflussanalyse. Ein solches Problem wird dafür auf ein Grapherreichbarkeitsproblem reduziert, um bewährte Methoden zur Lösung jener Probleme nutzen zu können. Da hierdurch sehr präzise Datenflussanalysen möglich sind, wird die Implementierung des Ansatzes in Form des Tools Heros in einigen Analysetools wie z. B. FlowDroid [ARF+14] und FlowTwist [LHBM14], die ein hohes Maß an Präzision fordern, verwendet.

FlowTwist [LHBM14] FlowTwist ist ein Tool zur IFDS-basierten Datenflussanalyse von Java-Programmen ausgehend von potenziell verwundbaren Calls bis zur API (*inside-out analysis*). Die zugrundeliegende Programmrepräsentation ist der von Soot erstellte Jimple-Bytecode bzw. ein daraus erstellter interprozeduraler Kontrollflussgraph (ICFG). Heros wird als IFDS-Implementierung für die Datenflussanalyse verwendet. Durch die Verfolgung des Datenflusses von innen nach außen konnte laut den Autoren eine verbesserte Skalierbarkeit im Vergleich zu einer herkömmlichen Vorwärtsanalyse (von außen nach innen) erreicht werden. Da für die in dieser Arbeit erstellte automatisierte Identifikation zuvor nicht bekannter Sicherheitsprobleme im Quellcode eine Vorwärtsanalyse notwendig ist, kann FlowTwist jedoch nicht verwendet werden.

SuSi Tool [ARB13] SuSi ist ein Tool zur Analyse von Quellen und Senken von Android Apps. Die untersuchten Quellen und Senken werden mithilfe von Machine Learning vollautomatisch identifiziert und klassifiziert. Da der Ansatz auf machinelles Lernen setzt und in der anzufertigenden Arbeit keine Verwendung von Machine Learning vorgesehen ist, kann SuSi im Rahmen dieser Arbeit nicht verwendet werden.

GUILeak [WQH+18] GUILeak ist ein Ansatz zum Tracing vertraulicher Daten in Android Apps. Verletzungen der Datenschutzbestimmungen der Apps werden auf diese Weise automatisch erkannt. Der Ansatz ist methodisch interessant, da die durchgeführte Analyse wie auch die in Abschnitt 5.3 vorgestellte Datenflussanalyse die Wahrung der Vertraulichkeit von Daten zum Ziel hat. Jedoch wurden bei der Lösungsfindung zu bewältigender Teilprobleme, wie z. B. der Referenzanalyse polymorpher Aufrufe, Android-spezifische Ansätze erstellt, die auf den in dieser Arbeit erstellten Ansatz, welcher die Datenflüsse in Java-Programmen untersucht, nicht direkt übertragbar sind.

4 Aufbau eines Datenflussmodells

Eines der beiden Hauptziele der vorliegenden Masterarbeit ist die Entwicklung eines Datenflussmodells, welches von den Implementierungsdetails objektorientierter Programme abstrahiert und stattdessen den daraus resultierenden Datenaustausch zwischen Membern in den Fokus rückt. Es sollen lediglich interprozedurale Flüsse sowie lokale Datenflüsse, die sich interprozedural auswirken (z. B. Parameter zu Rückgabewert) modelliert werden. Hieraus ergibt sich eine Modellform, auf der sich im Rahmen statischer Analysen einfache Datenflussmuster spezifizieren und finden lassen (siehe Kapitel 5), für deren Erstellung keine tiefgreifenden Kenntnisse auf dem Gebiet der Datenflussanalyse erforderlich sind.

Das im Rahmen der Masterarbeit entwickelte Datenflussmodell wird in Abschnitt 4.1 vorgestellt. Zu diesem Zweck wird zunächst beschrieben, warum das Modell auf dem Programmmodell von GRaViTY basiert und wie eine Darstellungsform gefunden wurde, die mächtig genug ist, um alle elementaren Flüsse objektorientierter Programme abbilden zu können. Anschließend wird das Metamodell sowie, zur Erleichterung des Verständnisses des Modells, eine Beispielinstanz dessen beschrieben. In Abschnitt 4.2 wird der Ablauf des Aufbaus des in 4.1 spezifizierten Datenflussmodells erläutert. Hierfür wird zunächst der allgemeine Ablauf des Programmmodellaufbaus des GRaViTY-Tools und anschließend der konkrete Ablauf des Datenflussmodellaufbaus beschrieben.

4.1 Spezifikation

Das für diese Arbeit spezifizierte Datenflussmodell basiert auf dem in Abschnitt 2.4 beschriebenen Programmmodell des GRaViTY-Tools. Dieses Modell erwies sich aus drei Gründen als geeignete Grundlage für das Hinzufügen von Datenflüssen objektorientierter Programme. Erstens soll das Zielmodell interprozedurale Datenflüsse enthalten. Da das Programmmodell von GRaViTY bereits Zugriffe zwischen Membern (Kontrollflüsse) enthält und solche Interaktionen auf Member-Ebene interprozedurale Datenflüsse mit sich bringen können, ist eine wesentliche Grundlage für das Ableiten und Einfügen von Datenflüssen bereits vorhanden. Somit entsteht eine Modellform, die auf Member-Ebene sowohl Kontroll- als auch Datenfluss enthält. Zweitens hat das Programmmodell bereits die gewünschte Granularität, da von einzelnen Statements abstrahiert wird, um die Essenz objektorientierter Programme, ohne einen für die vorgesehenen statischen Analysen überflüssigen Detailreichtum, erfassen zu können. Anders als z. B. bei einem feingranularen Datenflussgraphen, in dem jedes Statement von einem eigenen Knoten repräsentiert wird, können auf dem Programmmodell und folglich auch auf dem darauf basierenden Datenflussmodell

entsprechend einfach aufgebaute Datenflussanalysen durchgeführt werden. Da die Analysen basierend auf Anfragen des Nutzers ablaufen sollen, ist der einfache Aufbau des angefragten Modells, welcher gleichzeitig auch eine Vereinfachung der eingesetzten Anfragesprache ermöglicht, für die Bedienbarkeit von großem Vorteil. So erlaubt die Verwendung einer abstrakten objektorientierten Modellform beispielsweise die Suche nach unerwünschten Datenflussmustern durch die Spezifikation von Anfragen in ebendiesem abstrakten objektorientierten Stil. Hierdurch wird Entwicklern ohne Vorkenntnisse in der Datenflussanalyse, durch die Nutzung einer ihnen vertrauten Notationsform, dennoch das Durchführen feingranularer Datenflussanalysen ermöglicht. Der dritte Grund für die Eignung des Programmmodells für die Erweiterung durch Datenflüsse ist die Modellierung statischer Typinformationen, wodurch diese in statischen Analysen verwendet werden können. Dadurch können Zugriffe und die von ihnen hervorgerufenen Datenflüsse feingranularer untersucht werden als bei der sonst z. B. im Kontext von Methodenüberschreibung notwendigen Betrachtung aller potenzieller Typen eines Felds (vgl. Unterabschnitt 2.2.3).

Zum Finden eines für die Modellierung von Datenfluss geeigneten Modells wurden elementare Flüsse objektorientierter Programme identifiziert. Das zu erstellende Datenflussmodell sollte alle identifizierten Flüsse einheitlich darstellen können. Hierzu wurden zunächst für alle neun gefundenen elementaren Datenflüsse Code- und Modellbeispiele erstellt, welche in einem Git-Repository[1] gesammelt wurden. Durch eine Vereinheitlichung der Modellbeispiele wurde eine Datenflussmodelldarstellung gefunden, die aussagekräftig genug ist, um alle elementaren Datenflüsse objektorientierter Programme darzustellen.

Die resultierende Modellspezifikation ist in Abbildung 4.1 als Metamodell dargestellt. Für den Datenfluss unwesentliche Modellelemente wurden ausgeblendet. Das Modell beinhaltet aus dem Programmmodell von GRaViTY als zentrale Informationen Member (Methoden- und Felddefinitionen) und Zugriffe zwischen diesen. Erweiterungen des Programmmodells sind mit einem grünen Rahmen hervorgehoben. Die zusätzlichen konkreten Spezialisierungen von Zugriffen *TRead*, *TWrite* und *TReadWrite* geben an, um welche Art von Feldzugriff es sich handelt. Darüber hinaus sind auch hier Signaturen von Feldern und Methoden (einschließlich Parameter) Teil des Modells. Die zusätzlichen Datenflussinformationen werden über die Klassen *TAbstractFlowElement* und *TFlow* modelliert. Ersteres ist eine Oberklasse aller Modellelemente, die an einem Datenfluss, entweder als Quelle oder als Ziel, beteiligt sein können. Letzteres sind die Datenflüsse selbst, welche immer genau eine Quelle und genau ein Ziel besitzen. Darüber hinaus besitzt jeder Fluss einen *flowOwner*, welcher die Zugehörigkeit des jeweiligen Flusses zu einem Flusselement angibt. Hierdurch entsteht in dem Datenflussmodell eine Koppelung von Kontroll- und Datenfluss, welche insofern hilfreich ist, dass die Ursache eines Flusses über das Auslesen des *flowOwners* in Erfahrung gebracht werden kann. Die modellierten Datenflüsse sind größtenteils rein interprozeduraler Natur, können allerdings auch intraprozedural sein, da die effektiven intraprozeduralen Flüsse einer Methode, wie z. B. ein Fluss von einem Parameter einer Methode in ihren Rückgabewert, sich potenziell auch interprozedural auswirken.

[1] https://github.com/GRaViTY-Tool/dataflow-examples

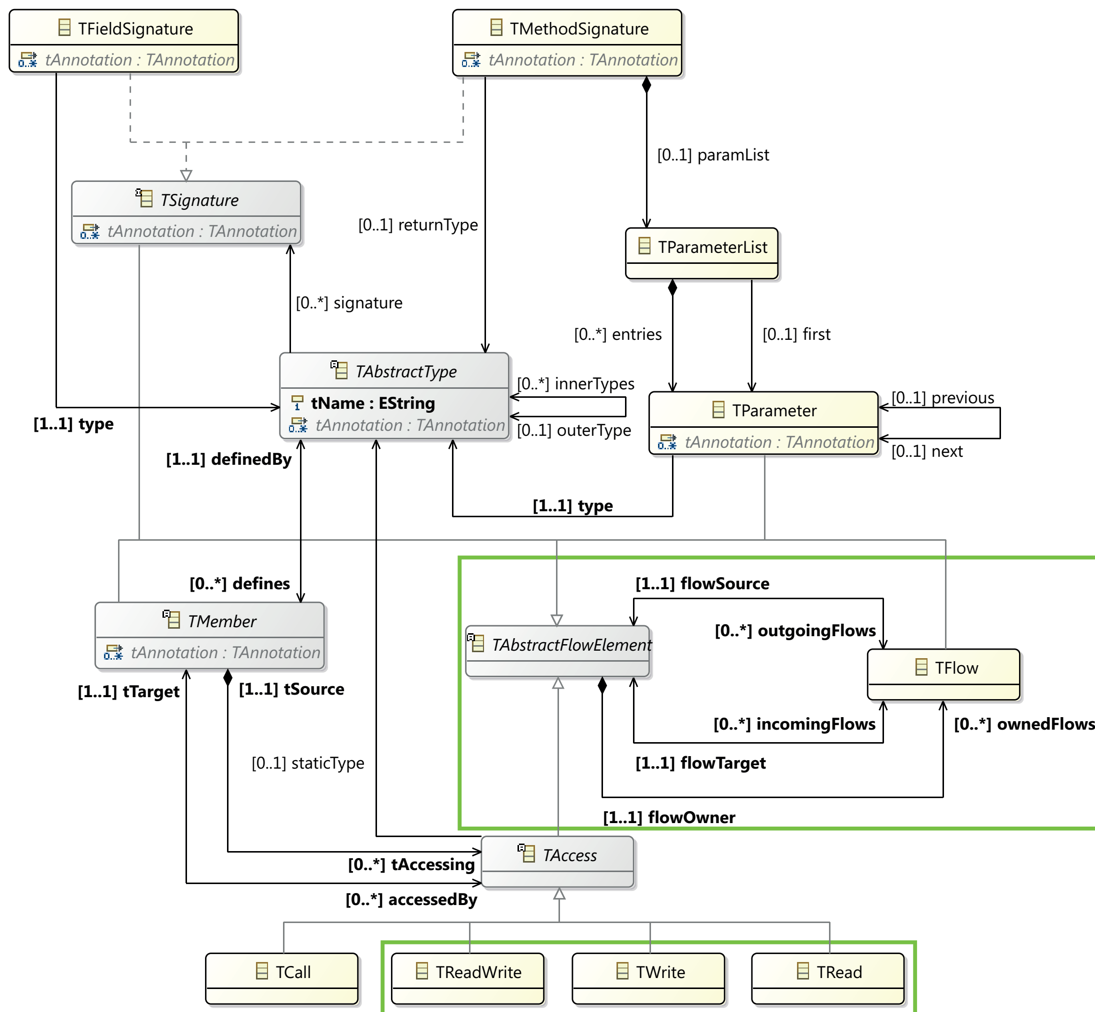

Abbildung 4.1: Metamodell des Datenflussmodells

Abbildung 4.2 veranschaulicht, wie die in Abschnitt 2.4 beschriebene Modellinstanz des Programmmodells (Abbildung 2.3) als um Datenflüsse erweiterte gültige Instanz des Datenflussmodells aussehen würde. Die der Programmmodellinstanz hinzugefügten Modellelemente sind in grün hervorgehoben. Der Übersichtlichkeit halber wurde der Datenfluss zwischen der *TMethodSignature* der Methode *getName* und dem *TParameter arg0* der *TMethodSignature* der Methode *eval* sowie die Rollenbezeichner der Flusskanten zwischen der *TFieldSignature* vom *TField engine* zum *TCall* der *TMethodDefinition eval* ausgelassen. Der Zugriff auf die *TFieldDefinition* des Felds *engine*, welcher zuvor im Programmmodell noch vom allgemeinen Typ *TAccess* war, wird im Datenflussmodell konkretisiert und ist daher vom Typ *TRead*. Durch den Zugriff wird ein Datenfluss vom Feld *engine* zum Aufruf der Methode *eval* mit zwei Teilflüssen ausgelöst. Beim ersten Teilfluss fließt der Feldinhalt aus der *TFieldSignature* des Feldes in den lesenden Zugriff. Anschließend fließt der Feldinhalt aus dem Zugriff zum Methodenaufruf der Methode *eval*. Der dargestellte Fluss zwischen der *TMethodSignature* der Methode *getText* und dem Parameter *arg0*, welcher durch den Aufruf der Methode *getText* ausgelöst wird (siehe *flowOwner*-

Kante), stellt einen im Kontext der Sicherheitsanalyse besonders relevanten Fluss dar. Die Methode *getText* gibt eine Nutzereingabe zurück und sollte daher nicht unbereinigt in die Methode *eval* fließen. Dies ist allerdings der Fall, weshalb an dieser Stelle eine Sicherheitslücke vorliegt. Auf dem Modell kann nun dank dem hohen Abstraktionsgrad eine einfache und leicht verständliche Datenflussanalyse mithilfe einer Graphanfrage durchgeführt werden, um die Schwachstelle zu finden. Diese wird in Abschnitt 5.2 vorgestellt.

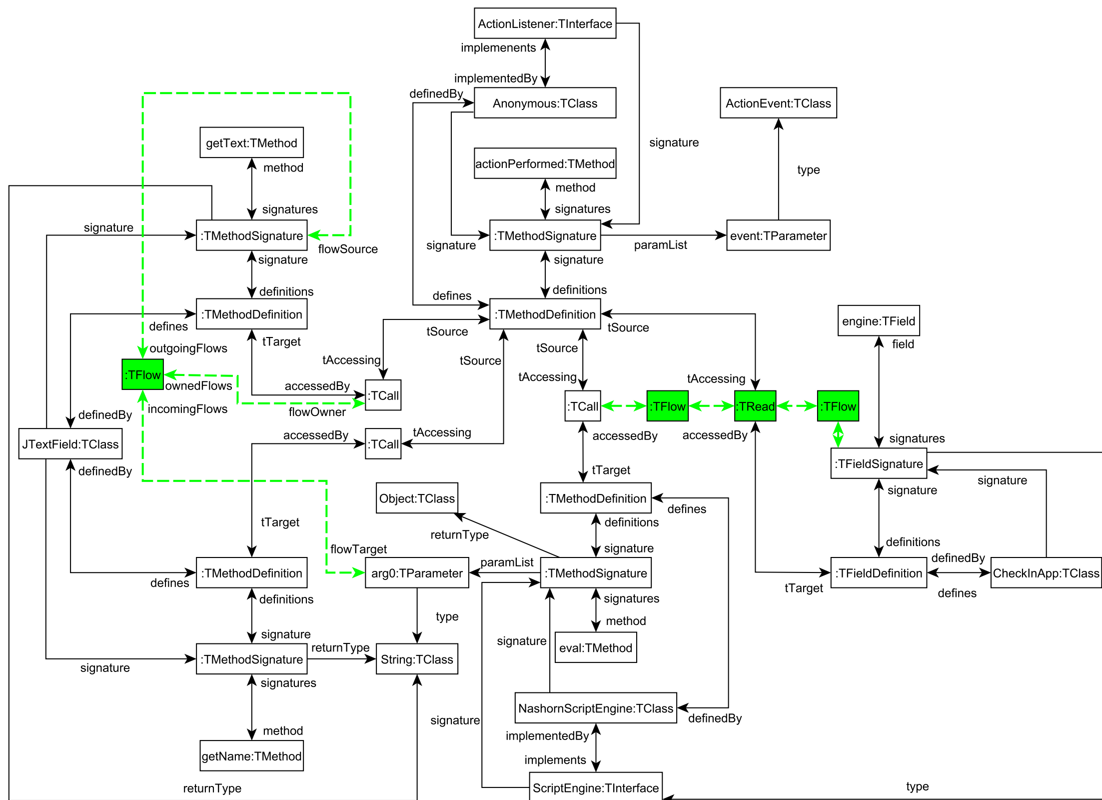

Abbildung 4.2: Beispielinstanz des Datenflussmodells

4.2 Modellaufbau

Zu dem Metamodell des Datenflussmodells (Abbildung 4.1) werden von dem im Rahmen der Arbeit entwickelten Tool, durch eine Erweiterung des zuvor in Abschnitt 2.4.2 beschriebenen allgemeinen Modellaufbaus aus Abbildung 2.4, Modellinstanzen beliebiger Java-Programme erstellt. Diese enthalten alle in dem Java-Programm möglichen interprozeduralen Datenflüsse mit Ausnahme von durch Exceptions ausgelösten Flüssen. Abbildung 4.3 zeigt eine Übersicht des Ablaufs des Datenflussmodellaufbaus. Schritt 1 des allgemeinen Modellaufbaus wird in gleicher Weise auch als erster Schritt des Datenflussmodellaufbaus durchgeführt, um einen vollständigen AST zu erhalten. In einem darauffolgenden Vorverarbeitungsschritt (Schritt 2) wird anschließend ein vollständiges intraprozedurales Datenflussmodell für jede im Quellcode gefundene Memberdefinition aufgebaut. Rudimentäre interprozedurale

Datenflüsse sind ebenfalls bereits enthalten. Ein Beispiel eines solchen Modells ist in Abbildung 4.4 zu sehen. Es ist das Modell der Methode *actionPerformed* aus dem Code-Beispiel 2.1 des durchgehenden Beispiels dargestellt. Die gezeigte Visualisierung des Graphen wurde mithilfe eines Java-Wrappers für Graphviz[2] als Dot-Graph generiert und diente lediglich als Feedback während der Entwicklung. Der Datenflussgraph enthält alle in der Methode rekursiv enthaltenen AST-Elemente (über schwarze gerichtete Kanten erreichbar) sowie die relevanten Elemente der Member, auf die *actionPerformed* zugreift. Die intra- und (noch unvollständigen) interprozeduralen Datenflüsse sind als grau gestrichelte Kanten eingezeichnet.

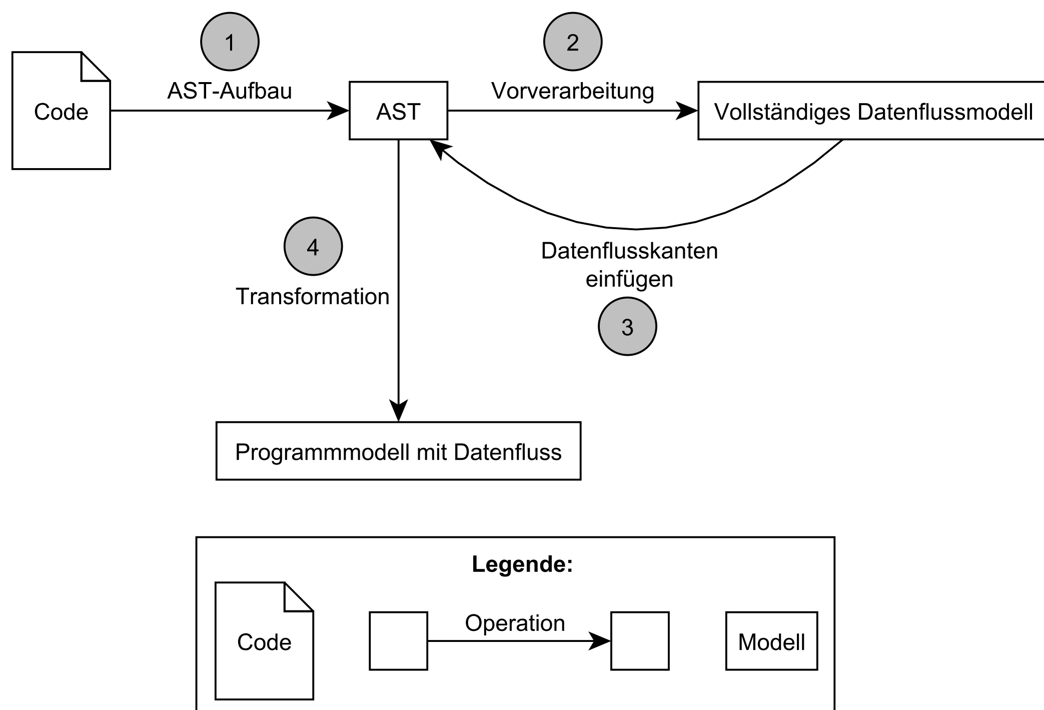

Abbildung 4.3: Ablauf des Datenflussmodellaufbaus

In Schritt 3 werden durch Reduktion der gegebenen intraprozeduralen Modelle detaillierte interprozedurale Datenflüsse abgeleitet und im AST eingefügt. Zu diesem Zweck werden alle Modellelemente, die nicht auf einer Whitelist stehen, reduziert. Die Whitelist besteht aus den folgenden Elementtypen:

- *VariableDeclarationFragment*, wobei nur solche behalten werden, deren Container eine *MFieldDefinition* ist, da es sich hierbei um Felder handelt.

- *ReturnStatement* sowie *IfStatement*, *WhileStatement*, *DoStatement*, *ForStatement*, *EnhancedForStatement* und *SwitchStatement* da durch Rückgabewerte direkt bzw. durch den beobachtbaren Kontrollfluss der Kontrollflussstrukturen indirekt Daten von einer Methode nach außen fließen.

[2]https://github.com/nidi3/graphviz-java

- *MSingleVariableAccess*, wobei nur Zugriffe auf Felder und Parameter behalten werden, da sich lokale Variablen nicht direkt auf interprozeduralen Datenfluss auswirken.
- *AbstractMethodInvocation*, da dieser Typ jede Art von Methodenaufruf umfasst.
- *SingleVariableDeclaration*, da es sich hierbei um Parameter handelt und diese ebenfalls interprozedurale Relevanz besitzen.
- *MAbstractMethodDefinition* (optional), damit in den reduzierten Graphen ersichtlich bleibt, welche Methodendefinitionen vorkommen.

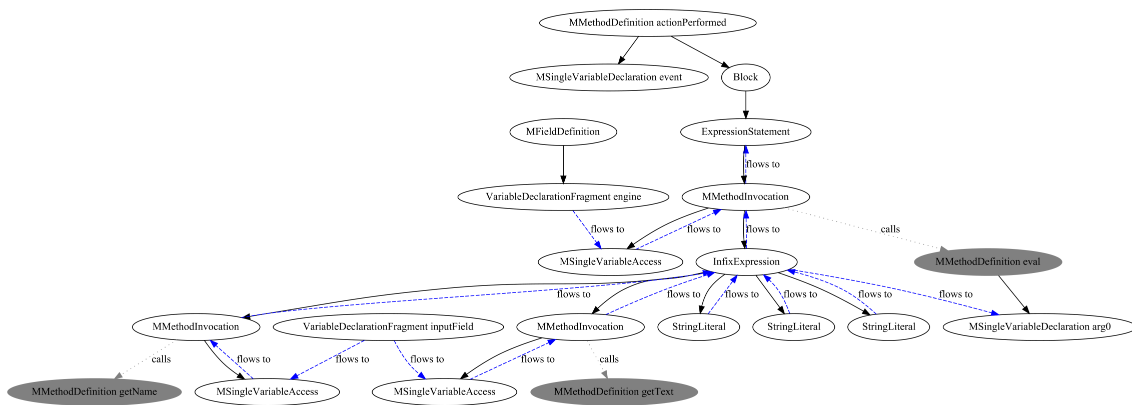

Abbildung 4.4: Generierter Dot-Graph für die Definition der Methode *actionPerformed*

Abbildung 4.5 zeigt, wie die vollständigen Datenflussmodelle reduziert werden, um effektive interprozedurale Flüsse bzw. solche, die sich interprozedural auswirken (z. B. Parameter in Rückgabewert), zu erhalten. Der Knoten *InfixExpression* wird zusammen mit seinen Flüssen (rot-gepunktete Kanten) entfernt, da er keine Relevanz für das interprozedurale Datenflussmodell besitzt. Hierbei werden die eingehenden Flüsse auf die Zielknoten der gelöschten ausgehenden Flüsse umgeleitet, wodurch die zwei grün-gestrichelten Flusskanten entstehen. Die entfernte Flusskante zwischen *InfixExpression* und *MMethodInvocation* wird jedoch nicht auf diese Weise umgeleitet, da in diesem Fall ein Fluss in einen Parameter (*arg0*) vorliegt und der allgemeinere Fluss in den Methodenaufruf an dieser Stelle lediglich zum Setzen der Zugehörigkeit (*flowOwner*) des Flusses im interprozeduralen Datenflussmodell herangezogen wird.

Schließlich wird das Modell, wie auch im allgemeinen Modellaufbau, mithilfe von TGG-Regeln auf die für die vorgesehenen Analysen relevanten Elemente reduziert und mit zuvor nur implizit vorhandenen Informationen angereichert (z. B. Zugriffsarten) (Schritt 4). Der TGG-Regelsatz wurde erweitert, um die Datenflüsse in das Programmmodell zu übernehmen zu können. Das resultierende Modell ist eine Instanz des durch das Metamodell in Abbildung 4.1 spezifizierten interprozeduralen Datenflussmodells. Der Modellaufbau kann zukünftig noch durch das Herstellen einer Korrespondenz zwischen dem Quellcode bzw. seiner Repräsentation als AST

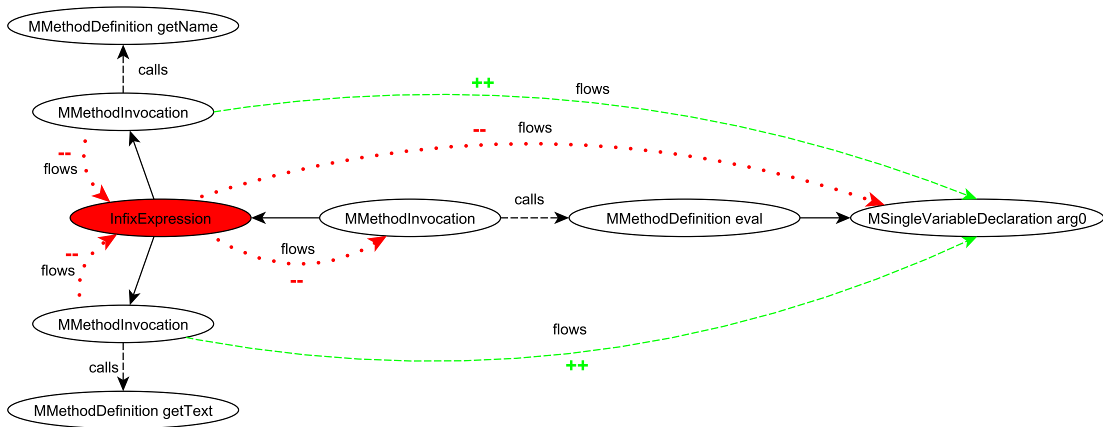

Abbildung 4.5: Beispiel eines Reduktionsschritts im intraprozeduralen Datenflussmodell

und dem interprozeduralen Datenflussmodell optimiert werden, da hierdurch eine inkrementelle Ko-Evolution der beiden Modelle, auf eine ähnliche Weise wie durch Peldszus et al. [PKLS15] beschrieben, ermöglicht wird.

5 Datenflussanalyse

Sicherheitsprobleme wie die durch Listing 2.1 demonstrierte unzureichende Eingabebereinigung werden durch unerlaubten Datenfluss verursacht. Im genannten Beispiel fließen Nutzereingaben direkt in die *eval*-Methode der JavaScript-Engine, ohne zuvor in eine Bereinigungsmethode zu fließen. Hierdurch wird das Injizieren von JavaScript-Code durch den Nutzer ermöglicht, welcher nicht neutralisiert wird und somit von der Script-Engine ausgeführt wird. Ein Angreifer hat somit die Möglichkeit, durch die Eingabe von JavaScript-Code in die z. B. auf Check-in-Automaten an Flughäfen laufende Java-Anwendung, auf dem Hostsystem potenziell unerwünschte Aktionen durchzuführen. Eine statische Analyse des Datenflusses eines Programms ist in der Lage, solche Sicherheitsprobleme zu identifizieren. So können Entwickler die durch unerlaubten Datenfluss auftretenden Sicherheitsprobleme beheben, bevor die Software zum Einsatz kommt.

Um durch unerlaubte Datenflüsse hervorgerufene Sicherheitsprobleme statisch erkennen zu können, werden deren Datenflussmuster auf dem im Rahmen dieser Arbeit erstellten Datenflussmodell (siehe Abschnitt 4.1) identifiziert. Ein Datenflussmuster ist beispielsweise in Abbildung 2.7 als Datenflussdiagramm dargestellt, welches die Weiterleitung von Nutzereingaben an die Methode *eval* aus Listing 2.1 veranschaulicht. Die Datenflüsse, welche im Datenflussdiagramm durch Kanten dargestellt werden, fließen im Wesentlichen zwischen den beiden Aktionen *getDataFromUI* und *eval* (sowie dem Akteur *Nutzer* und dem Speicher *BuchungsDB*). Als Datenflussmuster wird im Folgenden jede Abfolge von Flüssen zusammen mit einer Spezifikation der Flussquellen sowie -ziele bezeichnet. Im Datenflussdiagrammbeispiel ist ein Muster zu erkennen, bei dem von einem nicht vertrauenswürdigen Knoten *getDataFromUI*, in den ungefiltert Nutzereingaben fließen, ein direkter Datenfluss in die Aktion *eval* stattfindet. Da keine Bereinigung der Nutzereingaben erkennbar ist, stellt das Beispiel eine Instanz des zuvor beschriebenen Sicherheitsproblems der unzureichenden Eingabebereinigung dar. Im Folgenden soll daher unter anderem zur Erkennung dieses Sicherheitsproblems ein Datenflussmuster spezifiziert werden. Es werden Anfragen auf der zu untersuchenden Instanz des GRaViTY-Datenflussmodells spezifiziert, die das zu erkennende Muster enthalten und deren Auswertung somit die Modellbestandteile liefert, auf die das gesuchte Datenflussmuster passt. Beispielsweise kann es für Entwickler von Interesse sein, wo Daten von einer konkreten Signatur aus hinfließen. Da Aufrufe aufgrund von Polymorphie auf Signaturen und nicht auf Definitionen stattfinden, sind die Quellen der hierdurch hervorgerufenen Datenflüsse ebenfalls Signaturen und ein Datenfluss startet immer bei einer Signatur.

In diesem Kapitel wird die Datenflussanalyse mithilfe von in Anfragen codierten Datenflussmustern anhand dreier Beispielanwendungen beschrieben. Hierbei wird zunächst die zuvor erwähnte Extraktion aller von einer Signatur ausgehenden Datenflüsse beschrieben. Anschließend wird im Rahmen einer Konsistenzprüfung der Integrity-Eigenschaft konkreter Member die Identifikation des Sicherheitsproblems unzureichender Eingabebereinigung (siehe Listing 2.1) demonstriert. Schließlich wird eine Konsistenzprüfung der Secrecy-Eigenschaft konkreter Member durchgeführt, um Datenlecks vertraulicher Daten zu identifizieren.

5.1 Extraktion von Datenflüssen

Die Extraktion aller von einer konkreten Signatur ausgehenden Datenflüsse ist für Entwickler sinnvoll, die wissen möchten, an welchen Stellen im Programm ausgewählte Daten auftauchen. Beispielsweise kann es hilfreich sein, eine vollständige Ausgabe aller Member zu erhalten, in die Daten eines Members, der vertrauliche Daten enthält, hinfließen. Die Ausgabe kann z. B. als einfache Auflistung von Quelle-Ziel-Paaren der gefundenen Datenflüsse dargestellt werden. Eine weitere Möglichkeit ist die Visualisierung als Baum mit dem Quellknoten als Wurzel und allen über Datenflüsse erreichbaren Programmelementen als Blätter bzw. Knoten.

Mit der in Abbildung 5.1 als Diagramm dargestellten Anfrage, in der Modellelemente zur erleichterten Verständlichkeit gemäß Objektdiagrammkonvention als Objektknoten modelliert sind, können alle Paare von Signaturen und Membern gefunden werden, die über mindestens einen Datenfluss miteinander verbunden sind. Der gestrichelte Rahmen mit Sternsymbol gibt hierbei an, dass diese Pfadbeschreibung beliebig oft auftreten kann, damit das Muster gefunden wird. Durch Einschränkung der Quellsignatur oder -definition auf eine konkrete Signatur bzw. Definition können nur von hier ausgehende, anstelle aller Flüsse extrahiert werden.

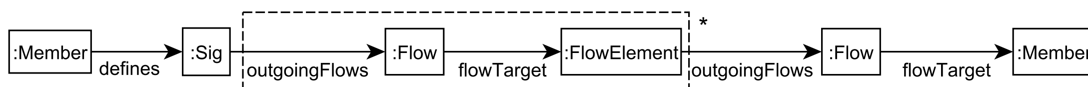

Abbildung 5.1: Anfragediagramm für die Extraktion aller Datenflüsse zwischen Signaturen und Memberdefinitionen

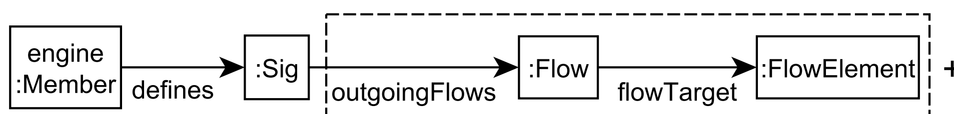

Abbildung 5.2: Anfragediagramm für die Extraktion aller Datenflüsse zwischen der Signatur des Felds *engine* und anderen Memberdefinitionen

Es kann z. B. die in Diagramm 5.2 gezeigte Anfrage spezifiziert werden, die als Flussquelle nur die zur Definition des Feldes *engine* gehörige Signatur zulässt.

Zusätzlich wurde die Anfrage so angepasst, dass nicht nur Flüsse mit Membern als Flussziel gefunden werden, sondern jede Form erlaubter Flussziele, sodass z. B. auch Flüsse in Methodenaufrufe identifiziert werden können. Das Ergebnis der Anfrage ist in Abbildung 5.3 als Ausschnitt der Instanz des Datenflussmodells aus Abbildung 4.2 visualisiert. Da das Feld *engine* im Beispielcode 2.1 ausschließlich in den Aufruf (*TCall*) der Methode *eval* sowie in seinen eigenen lesenden Zugriff (*TRead*) fließt, sind lediglich diese beiden Flüsse hervorgehoben. Der erste Fluss ausgehend von der Feldsignatur des Felds *engine* zum lesenden Zugriff *TRead* dieses Feldes ist in blau mit gestrichelt-gepunkteter Linie hervorgehoben. Da der Datenfluss von *TRead* weiter in den *TCall* der Methode *eval* läuft, wird auch der hieraus resultierende zusammengesetzte Flusspfad, welcher den ersten Fluss enthält, gefunden. Dieser ist rot gepunktet hervorgehoben.

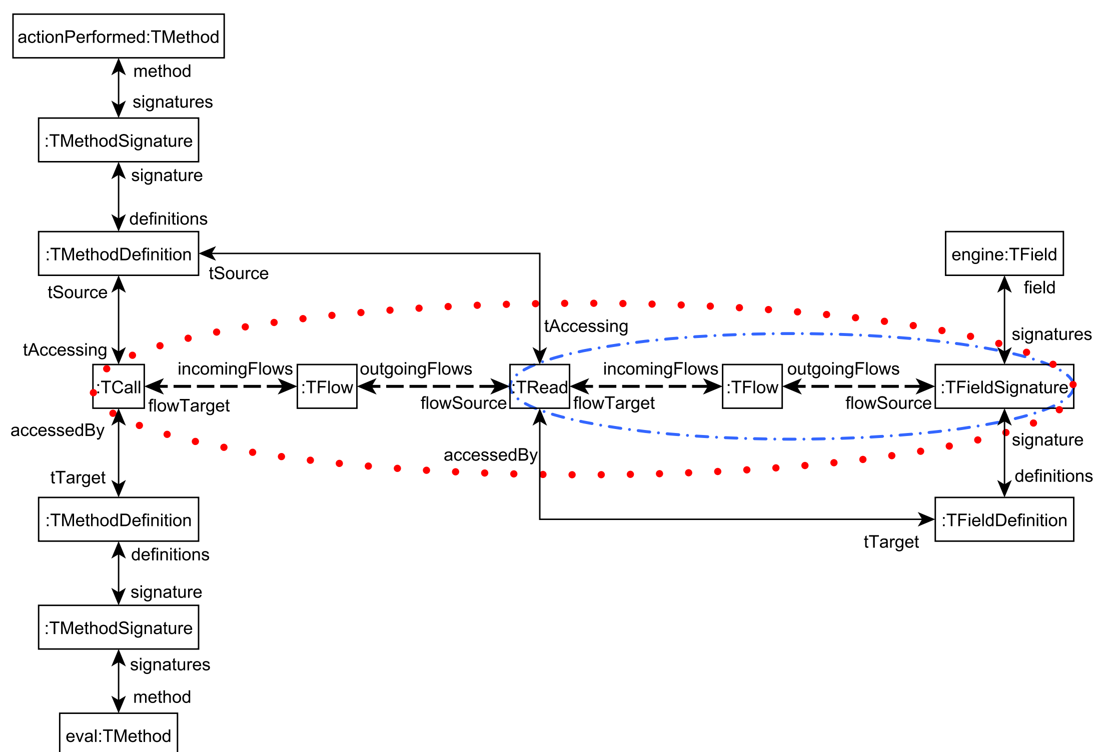

Abbildung 5.3: Ergebnis der Anfrage aus Abbildung 5.2

5.2 Konsistenzprüfung der Integrity-Eigenschaft

Mit den zuvor gezeigten Anfragen konnten die Flüsse aller bzw. konkreter Elemente des Datenflussmodells extrahiert werden, um diese manuell zu betrachten. Eine automatisierte statische Sicherheitsanalyse stellen die Anfragen allerdings noch nicht dar. Für die Entwicklung sicherheitskritischer Software ist die Möglichkeit einer automatisierten Erkennung von Schwachstellen von großem Nutzen. Es wäre möglich, dass ein Programm die in Listing 2.1 demonstrierte Schwachstelle unzureichender Eingabebereinigung beinhaltet, welche im Kontext eines komplexen Softwa-

reprojekts schnell übersehen werden könnte. Eine automatisierte Erkennung dieser Art von Schwachstellen hilft daher, mehr Vertrauen in die Qualität der entwickelten Software zu gewinnen. Daher wird im Folgenden eine Anfrage zur Erkennung unzureichender Eingabebereinigung vorgestellt.

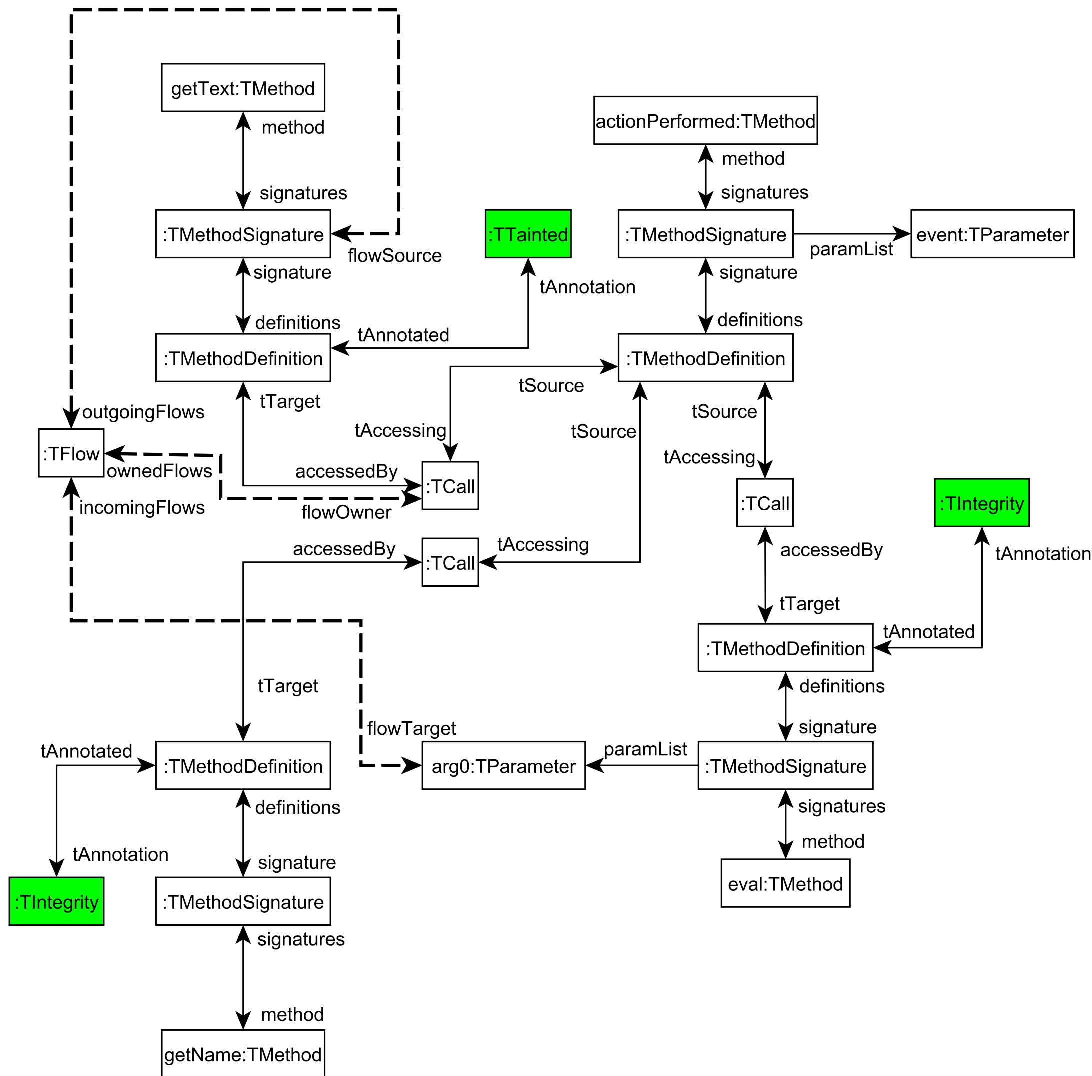

Abbildung 5.4: Ausschnitt des Datenflussmodells mit Annotationen

Der diesbezüglich relevante Ausschnitt des Datenflussmodells ist in Abbildung 5.4 dargestellt. Dieser enthält lediglich den kritischen Datenfluss, welcher für die Verletzung der Integrity-Eigenschaft und somit für ein Sicherheitsproblem sorgt, sowie drei Sicherheitsannotationen (grüne Hervorhebung), die das gegebene bzw. geforderte Sicherheitsniveau eines Members angeben. Es wird angenommen, dass diese Annotationen im Code bereits vorhanden sind. Die Memberdefinitionen des Programms sind somit mit Sicherheitsannotationen entsprechend der Listings 2.3, 2.4 und 2.5 versehen. Die verwendeten Annotationen geben an, dass die von der Methode *getText* kommenden Daten nicht vertrauenswürdig sind (im Modell: *TTainted*)

und, dass die Methoden *getName* sowie *eval* die Integrität (im Modell: *TIntegrity*) der von ihnen zurückgegebenen bzw. mit ihnen in Berührung kommenden Daten garantieren bzw. fordern (vgl. Abschnitt 2.1).

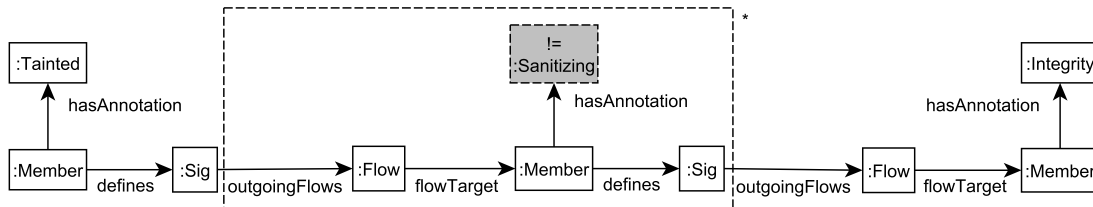

Abbildung 5.5: Anfragediagramm des für unzureichende Eingabebereinigung typischen Datenflussmusters

Bei der datenflussbasierten Sicherheitsanalyse werden in der Anfrage Einschränkungen für die Flüsse zwischen Membern mit bestimmten Annotationen codiert. Hierdurch kann die Konsistenz bzw. die Einhaltung der gegebenen Einschränkungen der vorhandenen Sicherheitsannotationen überprüft und Inkonsistenzen bzw. Verletzungen der Einschränkungen ausgegeben werden. Es wird angenommen, dass `@Integrity` ein höheres Sicherheitsniveau als `@Tainted`, welches für unbereinigte Nutzereingaben verwendet wird, darstellt. Außerdem wird die Annotation `@Sanitizing` für Methoden eingeführt, die eine Bereinigung von Nutzereingaben durchführen und somit das Sicherheitsniveau der Eingabedaten von `@Tainted` auf `@Integrity` heben. Gesucht ist demnach eine Anfrage, die alle Flüsse findet, bei denen Daten von mit `@Tainted` annotierten Membern zu mit `@Integrity` annotierten Membern fließen, ohne dass auf diesem Pfad ein Fluss in einen mit `@Sanitizing` annotierten Member vorkommt. Die in Abbildung 5.5 dargestellte Anfrage erfüllt diese Anforderungen. Der mit gestrichelter Kontur eingerahmte Teil der Anfrage ist mit einem Stern versehen, da er optional ist. Dieser Teil stellt einen Fluss in einen Member dar, welcher **nicht** mit der Annotation `@Sanitizing` versehen ist, da ansonsten mit dem anschließenden Fluss in einen mit `@Integrity` annotierten Member keine Verletzung der Integrity-Eigenschaft mehr vorliegt. Tritt das eingerahmte Teilmuster nicht auf, liegt ebenfalls eine Violation vor, da ein direkter Fluss eines mit `@Tainted` annotierten Members in einen mit `@Integrity` annotierten Member vorliegt. Die Auswertung der Anfrage bewirkt dementsprechend die Identifikation aller Flüsse von mit `@Tainted` annotierten Membern in mit `@Integrity` annotierte Member über beliebig lange Datenflusspfade, ohne dass die Daten auf dem Pfad durch einen mit `@Sanitizing` annotierten Member fließen.

Das Ergebnis der Anfrage ist in Abbildung 5.6 visualisiert. Es ist der Fluss zusammen mit Quelle und Ziel durch eine blaue gestrichelt-gepunktete Linie hervorgehoben, der eine Verletzung der Integrity-Eigenschaft darstellt. Der dargestellte Fluss beginnt bei der Methode *getText*, welche mit `@Tainted` annotiert ist, da sie die Nutzereingabe eines Eingabefeldes zurückgibt. Das unmittelbar folgende Flussziel ist bereits die Methode *eval*, welche mit `@Integrity` annotiert ist. Da zwischen Quelle und Ziel keine Methode zur Eingabebereinigung mit der Annotation

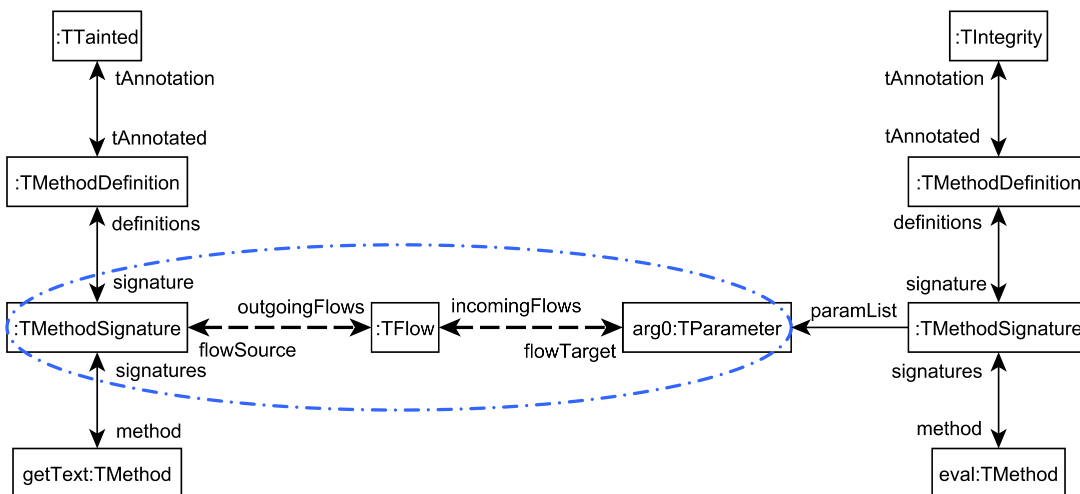

Abbildung 5.6: Ergebnis der Anfrage aus Anfragediagramm 5.5

`@Sanitizing` aufgerufen wird, ist das dargestellte Flussmuster eine Verletzung der Integrity-Eigenschaft.

Listing 5.1: Aufruf der *eval*-Methode nach voriger Bereinigung der Nutzereingabe

```
1 eval(getName() + sanitize(getText()));
```

Da das im Rahmen der Arbeit aufgebaute Datenflussmodell sowohl Daten- als auch Kontrollfluss in Form von Zugriffen zwischen Membern enthält, kann zu jedem Datenfluss das entsprechende Kontrollflusselement abgefragt werden, um die Ursache für den jeweiligen Fluss zu erfahren. Der für einen Fluss verantwortliche Kontrollflussknoten ist über die *flowOwner*-Kante im Datenflussmodell referenziert. Beispielsweise wird der durch die Anfrage identifizierte unerwünschte Datenfluss vom Aufruf der Methode *getText* durch die Methode *actionPerformed* verursacht (siehe Abbildung 5.4). An dieser Stelle kann dementsprechend auch die Behebung der Integrity-Verletzung ansetzen. Abbildung 5.7 zeigt den für die Behebung relevanten Ausschnitt des Datenflussmodells, welcher um eine Umleitung des problematischen Flusses in eine *sanitize*-Methode erweitert wurde. Der Fluss des Methodenaufrufs *getText* wird hierbei in die Methode *sanitize* umgeleitet, indem diese ebenfalls von *actionPerformed* aufgerufen wird und die Rückgabe von *getText* als Parameter annimmt. Da die Methode *sanitize* mit `@Sanitizing` annotiert ist (siehe Knoten *TSanitizing*), kann anschließend deren Rückgabe in den Parameter *arg0* der Methode *eval* fließen, ohne die Integrity-Eigenschaft zu verletzen. Die Umsetzung der Behebung im Quellcode ist in Listing 5.1 dargestellt.

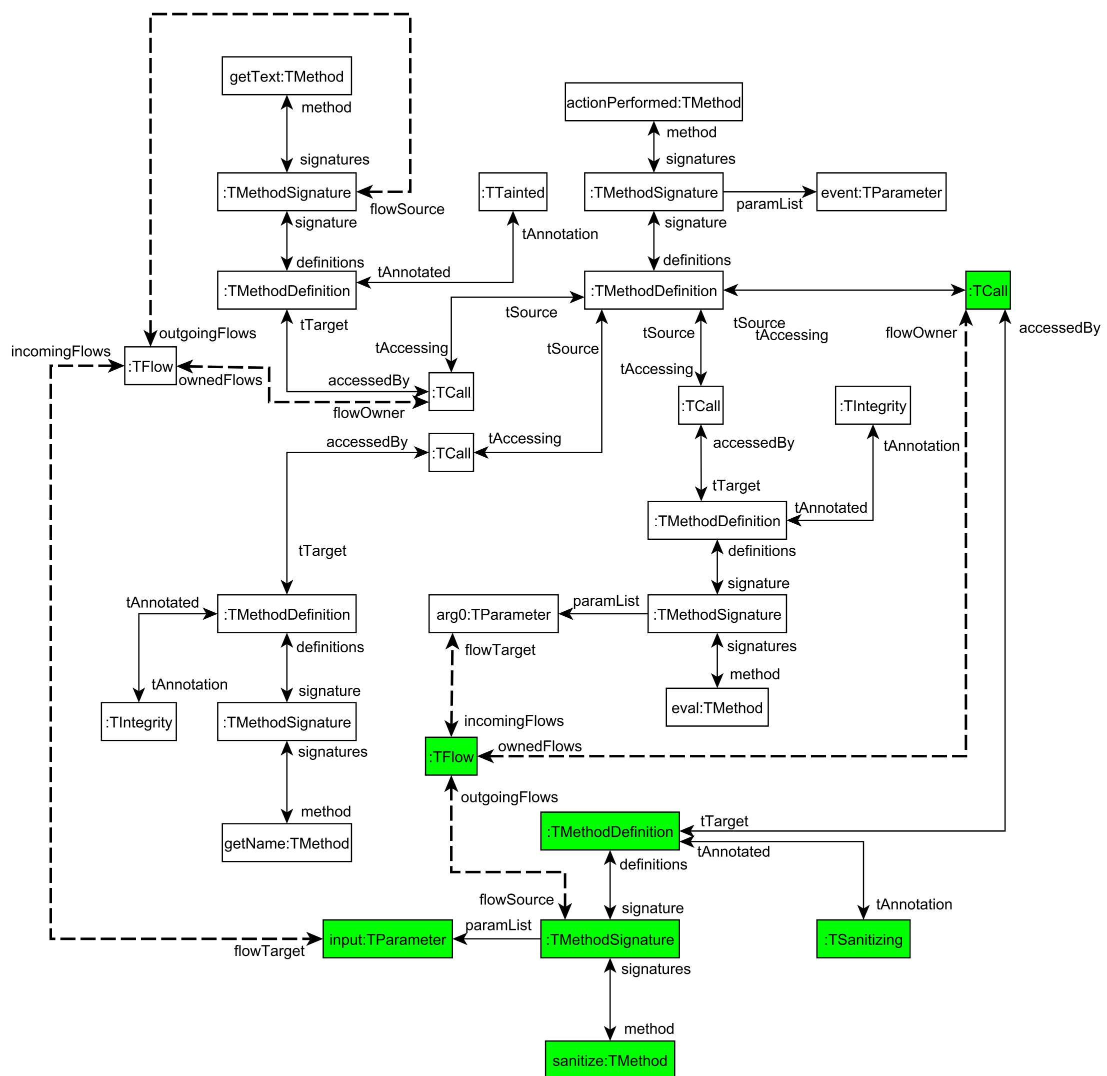

Abbildung 5.7: Ausschnitt des Datenflussmodells nach Hinzufügen der Eingabebereinigung

5.3 Konsistenzprüfung der Secrecy-Eigenschaft

In Systemen, in denen vertraulich zu behandelnde Daten vorkommen, ist eine automatisierte Identifikation von Datenlecks, welche die vertraulichen Daten offenlegen, sinnvoll. Beispielsweise könnte der Quellcode eines Projekts gegeben sein, in dem alle vertraulichen Member sowie alle Member, welche die Sicherheitsanforderung der Vertraulichkeit aufrechterhalten, mit der Sicherheitsannotation `@Secrecy` annotiert sind. Um unerwünschte Flüsse vertraulicher Daten in nicht annotierte Member zu identifizieren, kann das Muster, welches den Datenfluss eines Members mit Annotation `@Secrecy` in einen Member ohne diese Annotation darstellt, in einer Anfrage spezifiziert werden. Die Auswertung der Anfrage führt zur Ausgabe aller Flüsse, auf die das angegebene Muster passt, und somit zu allen Verletzungen der Konsistenz der Secrecy-Annotationen. Grafisch kann eine solche Anfrage mit dem in Abbildung 5.8 gezeigten Anfragediagramm ausgedrückt werden. Der gestrichelt eingerahmte Bereich kann, da er mit einem Stern versehen ist, gar nicht bis beliebig oft vorkommen. Er stellt mögliche Zwischenflüsse in Member dar, die mit `@Secrecy` annotiert sind und dadurch nicht gegen die Secrecy-Eigenschaft verstoßen. Sobald der untersuchte Fluss jedoch in einen Member fließt, der nicht mit `@Secrecy` annotiert wurde, trifft auch der letzte Teil der Anfrage zu und es liegt eine Inkonsistenz der Secrecy-Annotationen auf dem gefundenen Pfad vor. Alle auf diesem Weg gefundenen Pfade können auf die gleiche Weise wie Abschnitt 5.2 ausgegeben werden. Da auf der in Abbildung 4.2 dargestellten Datenflussmodellinstanz keine vertraulichen Daten fließen, ist die Ausgabe der Treffer in diesem Fall leer.

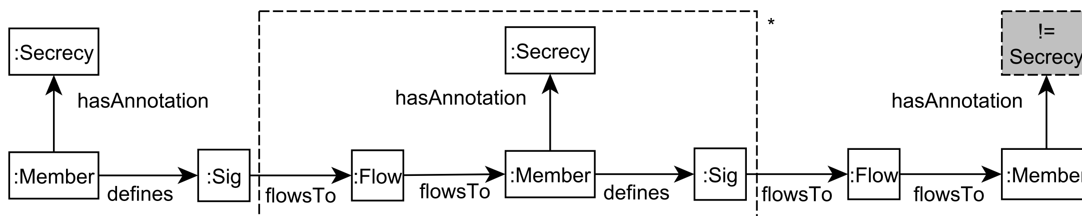

Abbildung 5.8: Anfragediagramm für die Konsistenzprüfung der Secrecy-Eigenschaft

6 Implementierung und Evaluation

In den bisherigen Kapiteln wurde insbesondere der methodische Aspekt des Datenflussmodellaufbaus sowie der darauf durchgeführten Datenflussanalysen beleuchtet. Durch die Implementierung des vorgestellten Ansatzes und dessen anschließender Evaluation konnte gezeigt werden, dass er praktisch umsetzbar ist und funktioniert. Dieses Kapitel behandelt Implementierungsdetails sowie die Ergebnisse der Evaluation des entwickelten Ansatzes, um tiefere Einblicke in dessen Umsetzung und Anwendbarkeit zu gewähren. Zunächst wird in Abschnitt 6.1 der Aufbau des erstellten Tools, welches das in Abschnitt 2.4 vorgestellte GRaViTY-Tool erweitert, anhand eines vereinfachten Komponentendiagramms erläutert und weitere Details der Implementierung beschrieben. Anschließend werden in Abschnitt 6.2 die Ergebnisse der Evaluation, sowohl des Modellaufbaus als auch der Datenflussanalyse, dargelegt und diskutiert.

6.1 Implementierung

Die Implementierung der im Rahmen der Arbeit erstellten methodischen Ansätze dient dazu, deren praktische Umsetzbarkeit zu evaluieren. Durch eine erfolgreiche Implementierung wird zum einen gezeigt, dass das in Abschnitt 4.1 spezifizierte Datenflussmodell gemäß dem in Abschnitt 4.2 beschriebenen Modellaufbau für beliebige korrekte Java-Programme aufgebaut werden kann. Zum anderen werden die in Kapitel 2.5.2 beschriebenen Datenflussanalysen unter Verwendung von Anfragen, welche zu erkennende Datenflussmuster enthalten, ebenfalls praktisch umgesetzt.

Dieser Abschnitt enthält Details zu den genannten Teilen der Implementierung. Da diese basierend auf dem in Abschnitt 2.4 beschriebenen GRaViTY-Tool durchgeführt wurde, werden im Wesentlichen die Teile der Implementierung beschrieben, welche das Tool erweitern. In Unterabschnitt 6.1.1 wird anhand eines Komponentendiagramms beschrieben, wie sich die neuen Implementierungsteile in den bestehenden Kern des Tools einfügen. Anschließend wird in Unterabschnitt 6.1.2 beschrieben, wie die Modelltransformation, welche Triple-Graph-Grammatik nutzt, erweitert wurde. Unterabschnitt 6.1.3 beschäftigt sich schließlich damit, wie die in Kapitel 2.5.2 beschriebenen Anfragen zur Erkennung von Datenflussmustern mit dem Tool GReQL umgesetzt wurden.

6.1.1 Komponentendiagramm der GRaViTY-Erweiterung

In Abbildung 6.1 sind die wesentlichen Komponenten der im Rahmen der Arbeit implementierten GRaViTY-Erweiterung sowie ihre Beziehungen zueinander dargestellt. In der Komponente *org.gravity.eclipse* sind die Schnittstellen zur Kommuni-

kation mit dem Eclipse-Plug-in implementiert. Sie verwendet die Schnittstellen *IPG-Converter* und *IPGConverterFactory* sowie das von der Komponente *org.gravity.typegraph.basic* zur Verfügung gestellte *Typegraph* Metamodell zum Durchführen der TGG-basierten Modelltransformation. Die UI-Komponente *org.gravity.eclipse.ui* nutzt die gleichen Interfaces, um das Anstoßen der Modellerstellung auch über die Benutzerschnittstelle durchführbar zu machen. Die für die Modelltransformation verantwortliche Komponente *org.gravity.tgg.modisco* verwendet die Schnittstellen *GravityModiscoProjectDiscoverer* und *MGravityModel* von *org.gravity.modisco*, um den zu transformierenden AST aufzubauen. Sie wurde durch kleinere Änderungen und Hinzufügungen der TGG-Regeln erweitert. Der im Rahmen der Arbeit implementierte Aufbau interprozeduraler Datenflüsse befindet sich in der Komponente *org.gravity.modisco.dataflow*. Sie nutzt das von *org.gravity.modisco* zur Verfügung gestellte *MGravityModel*, welches dem AST entspricht, und fügt diesem Datenflüsse hinzu. Damit die Datenflussverarbeitung von *org.gravity.modisco* aufgerufen werden kann, implementiert sie, wie auch die bereits vorhandenen Pre-Processing-Komponenten, das Interface *IModiscoProcessor*.

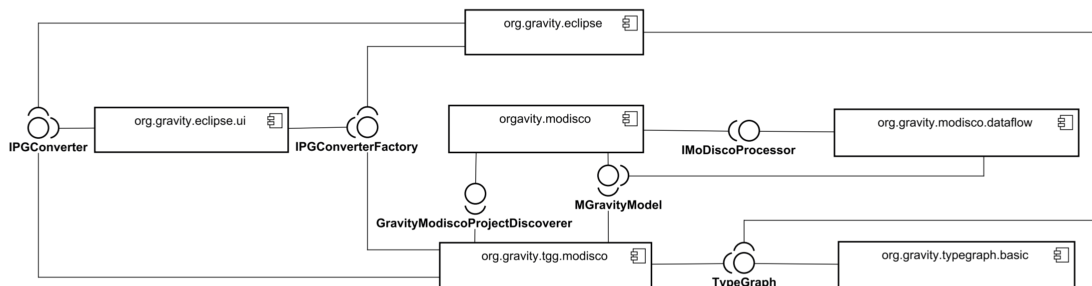

Abbildung 6.1: Komponentendiagramm der GRaViTY-Erweiterung

6.1.2 Hinzugefügte TGG-Regeln

Die Komponente *org.gravity.tgg.modisco* aus Abbildung 6.1 definiert einen TGG-Regelsatz für die Transformation des von MoDisco und den Pre-Processings aufgebauten modifizierten AST zum GRaViTY-Programmmodell. Damit die von der Komponente *org.gravity.modisco.dataflow* im MoDisco-Modell eingefügten Datenflüsse ebenfalls transformiert werden können, wurden im Rahmen der Arbeit weitere TGG-Regeln hinzugefügt. Beispielsweise werden mit der in Abbildung 6.2 visualisierten Regel Flussknoten übersetzt. Bei der Regelanwendung werden zunächst die Elemente *mOwner* vom Typ *MAbstractFlowElement* und *mFlow* vom Typ *MFlow*, die über das Attribut *ownedFlows* des Typs *MAbstractFlowElement* in Beziehung stehen, im zu übersetzenden MoDisco-Modell identifiziert. Sobald dieses Muster gefunden wurde, wird der noch nicht in TGG-Repräsentationsform vorliegende Knoten *mFlow* sowie die Kante *ownedFlows* (grüne Hervorhebung) in der TGG-Repräsentation des MoDisco-Modells erstellt. Anschließend werden die MoDisco-Modellelemente gemäß der Korrespondenzregeln (gestrichelte Kanten) in die zugehörigen Datenflussmodellelemente *tOwner* und *tFlow* sowie die Kante *ownedFlows* transformiert.

Durch Anwendung der Regel wird sichergestellt, dass alle Flussknoten vom Typ *MFlow*, die in dem dargestellten Kontext eines *flowOwners* vom Typ *MAbstract-FlowElement* vorkommen, in einen Flussknoten vom Typ *TFlow* in einem äquivalenten Kontext transformiert werden. Die Flussquelle und das Flussziel werden in je einer weiteren TGG-Regel separat übersetzt. Diese sind in den Abbildungen 6.3 und 6.4 dargestellt.

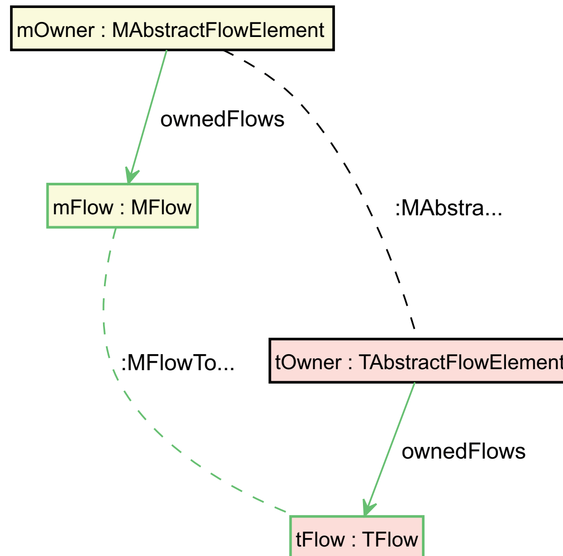

Abbildung 6.2: TGG-Regel zum Transformieren von Datenflüssen

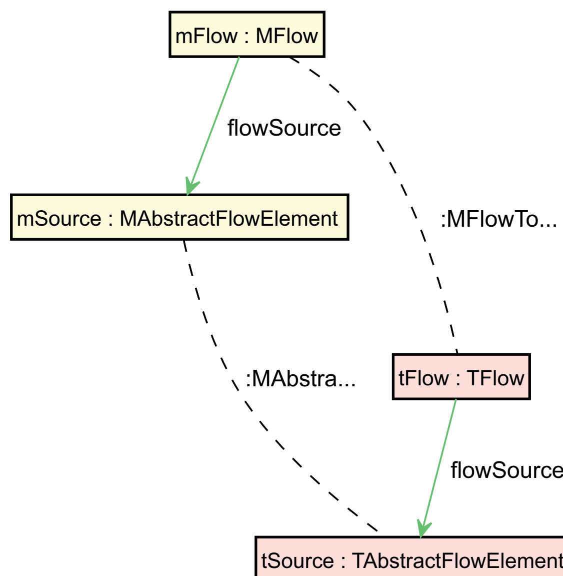

Abbildung 6.3: TGG-Regel zum Transformieren von Datenflussquellen

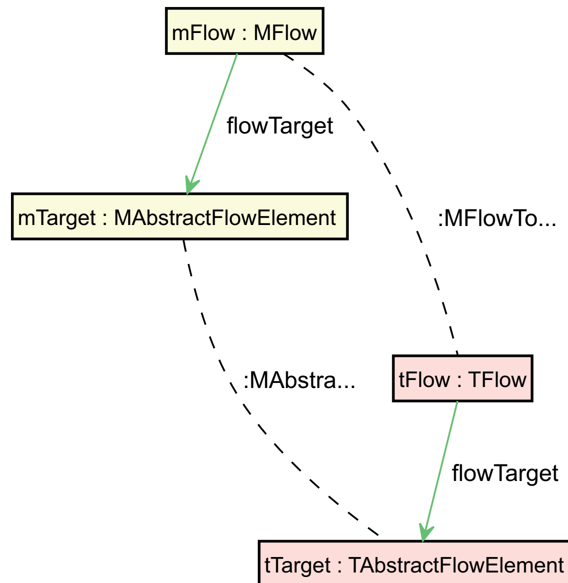

Abbildung 6.4: TGG-Regel zum Transformieren von Datenflusszielen

6.1.3 Datenflussanalyse mit GReQL

Die Mustererkennung wird mithilfe des Tools GReQL, welches Teil des Java-Graph-Labors JGraLab[1] ist, durchgeführt. Da GReQL auf sogenannten *TGraphen* arbeitet, wird zunächst das aufgebaute Datenflussmodell in diese Graphrepräsentation konvertiert. Das Tool *Ecore2Tg*[2] ermöglicht diese Umwandlung. Für die Analyse werden die in Kapitel 5 als Diagramme spezifizierten Anfragen als GReQL-Ausdrücke umgesetzt. Ein GReQL-Ausdruck ist eine deklarative Anfrage, deren Auswertung auf einer Graphinstanz entsprechend die durch die Anfrage spezifizierten Elemente oder Subgraphen zurückgibt. Um eine einfache Darstellung zu ermöglichen, welche den Kern einer Anfrage wiedergibt, abstrahieren die Anfragediagramme von ausgewählten Details des zugrundeliegenden Modells, wodurch ein Anfragediagramm im Allgemeinen nicht 1:1 auf eine GReQL-Anfrage abgebildet werden kann. Beispielsweise sind etwaige Methodenparameter in den Anfragediagrammen ausgelassen. Listing 6.1 zeigt eine GReQL-Anfrage, welche die in Anfragediagramm 5.5 spezifizierte Anfrage zur Identifikation von unzureichender Eingabebereinigung praktisch umsetzt. Die Anfrage besteht im Wesentlichen aus zwei Ausdrücken. Einem sogenannten *from-with-report*-Ausdruck (*FWR*-Ausdruck) sowie einem *Where*-Ausdruck. Der *FWR*-Ausdruck gibt an, welche Modellelemente (*from*-Teil) unter welchen Bedingungen (*with*-Teil) gefunden werden sollen und welche Ausgabe hieraus erzeugt werden soll (*report*-Teil). Im *Where*-Ausdruck werden Variablen für die Verwendung im FWR-Ausdruck definiert. Die Definition einer Variable kann hierbei wiederum, wie im Beispiel zu sehen, selbst einen FWR-Ausdruck enthalten. Das Beispiel beinhaltet im *Where*-Ausdruck die Definitionen der Variablen *tainted-*

[1] http://jgralab.github.io
[2] https://github.com/jgralab/ecore2tg

Members, *integrityMembers* und *sanitizingMembers*, welche jeweils die Mengen von *TMember*-Knoten zurückgeben, die mit *Tainted*, *Integrity* bzw. *Sanitizing* annotiert sind. Im *FWR*-Ausdruck werden alle Memberpaare angefragt, die über den im *With*-Teil spezifizierten Pfad in Verbindung stehen. Für diesen Pfad sind unter Verwendung des logischen *or*-Operators zwei Alternativen angegeben. Die Pfadbeschreibung beginnt in beiden Fällen bei dem Member m, welcher mit *Tainted* annotiert ist. Aus dessen Signatur fließen Daten in einen Member d, der mit *Integrity* annotiert ist. Der hierdurch gegebene Datenflusspfad kann optional (Pfadbeschreibungsteil in eckigen Klammern) auch in einen Parameter der zu d gehörenden Signatur statt in d selbst fließen. Dies ist der Fall, wenn d eine Methodendefinition ist. Andernfalls ist d eine Felddefinition. Bei der ersten Pfadalternative, auf der linken Seite des *or*-Operators, treten auf dem Pfad zwischen m und d, abgesehen von einem potenziellen Fluss in einen Parameter von d, keine Flüsse in Methodenparameter auf. Dieser Pfad muss gematcht werden, da ohne Zwischenflüsse in Parameter, und somit in potenziell bereinigende Methoden, keine Bereinigung der Daten von m stattfinden kann. Die zweite Alternative, auf der rechten Seite des *or*-Operators, sieht einen Zwischenfluss in einen Parameter vor. Der Pfad muss allerdings nur gefunden werden, falls die Signatur des Parameters von einem Member implementiert wird, der **kein** Element von *sanitizingMembers* ist, da ansonsten eine Eingabebereinigung stattfindet. Im Gesamten findet die Anfrage somit alle über einen Datenflusspfad verbundenen Memberpaare m und d, für die gilt, dass m mit *Tainted* sowie d mit *Integrity* annotiert ist, und sich auf dem Pfad zwischen m und d kein Fluss in eine mit der Annotation *Sanitizing* markierte Bereinigungsmethode befindet.

Während die Ausgabe von Memberpaaren, welche Quelle und Ziel der unerwünschten Datenflüsse darstellen, in manchen Fällen durch Zuhilfenahme des Quellcodes noch ausreichend sein kann, um den Weg der Daten nachzuvollziehen, ist es bei längeren Datenflüssen deutlich hilfreicher, den gesamten Datenflusspfad zu kennen. Unter Zuhilfenahme der Bibliotheksfunktion *path* von GReQL können die gefundenen Pfade zwischen den Variablen m und d ausgeben werden. Die Ausgabe entspricht einer Auflistung aller Knoten und Kanten des Pfades in der Reihenfolge, in der sie auf dem Weg von m nach d passiert werden. Der in Abbildung 6.5 gezeigte Screenshot zeigt die GReQL-Ausgabe, die neben dem Memberpaar auch den Pfad zwischen den Membern beinhaltet. Die Knoten v26 und v41 entsprechen den Membern m und d. Jede Kante ist mit dem Buchstaben e und einem Index versehen. Ein negativer Index bedeutet, dass die Kantenbezeichnung ausgehend vom Zielknoten gewählt ist. Beispielsweise ist die Kante mit der Bezeichnung *SignatureContainsDefinitions2* eine Kante der *TMethodSignature* v24. Von oben nach unten gelesen ergibt die Auflistung von Knoten und Kanten den Datenflusspfad zwischen m und d. Der Entwickler kann so nachvollziehen, auf welchem Weg die unbereinigten Nutzereingaben in die mit *Integrity* annotierte Evaluierungsmethode fließen und die Schwachstelle durch das Einfügen einer Bereinigungsmethode (vgl. Listing 5.1) beheben.

Listing 6.1: GReQL-Anfrage zum Identifizieren unzureichender Eingabebereinigungen

```
1  from m: taintedMembers,
2       p: V{TParameter},
3       s2: V{TSignature},
4       s: sanitizingMembers,
5       d: integrityMembers
6  with
7    m <--{signature} (<--{outgoingFlows}-->{flowTarget})+
         [<--{TParameterListContainsEntries}
         <--{TMethodSignatureContainsParamList}
         -->{SignatureContainsDefinitions2}] d
8    or
9    m <--{signature} (<--{outgoingFlows}-->{flowTarget})+ p
         <--{TParameterListContainsEntries}
         <--{TMethodSignatureContainsParamList} s2
         (<--{outgoingFlows}-->{flowTarget})+
10     [<--{TParameterListContainsEntries}
           <--{TMethodSignatureContainsParamList}
           -->{SignatureContainsDefinitions2}] d
11   and not (
12     s2 --> s
13   )
14 reportSet m, d
15 end
16 where
17    taintedMembers := from m: V{TMember}, ty:
          V{annotations.TAnnotationType}
18    with (m <--{tAnnotation}<--{type} ty) and (ty.tName = "Tainted")
19    reportSet m end,
20
21    integrityMembers := from m: V{TMember}, ty:
          V{annotations.TAnnotationType}
22    with (m <--{tAnnotation}<--{type} ty) and (ty.tName = "Integrity")
23    reportSet m end,
24
25    sanitizingMembers := from m: V{TMember}, ty:
          V{annotations.TAnnotationType}
26    with (m <--{tAnnotation}<--{type} ty) and (ty.tName = "Sanitizing")
27    reportSet m end
```

```
Graph id: .xmiGraph

Result size: 1
```

v26: TMethodDefinition	v41: TMethodDefinition	v26: TMethodDefinition e-101: SignatureContainsDefinitions2 v24: TMethodSignature e-57: OutgoingFlowsLinksToFlowSource v16: TFlow e58: IncomingFlowsLinksToFlowTarget v15: TCall e-6: OutgoingFlowsLinksToFlowSource v21: TFlow e7: IncomingFlowsLinksToFlowTarget v40: TParameter e-14: TParameterListContainsEntries v39: TParameterList e-33: TMethodSignatureContainsParamList v38: TMethodSignature e29: SignatureContainsDefinitions2 v41: TMethodDefinition

Abbildung 6.5: GReQL-Ausgabe des gefundenen Memberpaars und deren gemeinsamen Datenflusspfads

6.2 Evaluation

Mit der Implementierung wurde bereits gezeigt, dass sowohl der Modellaufbau als auch die Datenflussanalyse auf dem aufgebauten Modell praktisch umsetzbar sind. In diesem Abschnitt wird ergänzend die Anwendbarkeit auf reale Java-Projekte evaluiert. Im Fokus stehen dabei die Laufzeit des Modellaufbaus sowie die Präzision der Datenflussanalyse. Da im Modellaufbau ein Modell des gesamten ASTs aller Quelldateien aufgebaut und verarbeitet wird, wächst dessen Ausführungszeit mit der Projektgröße an. Die im Rahmen der Arbeit implementierte Erweiterung des Modellaufbaus um das Ableiten, Reduzieren sowie Einfügen von Datenflüssen soll den Faktor dieses Anstiegs nicht wesentlich erhöhen. Eine weitere Anforderung ist es, dass die Datenflussanalyse auf dem Modell möglichst präzise durchgeführt werden kann. Es sollten durch das Auswerten der jeweiligen Anfragen alle in der Anfrage spezifizierten Datenflussmuster und gleichzeitig möglichst wenige vermeintliche Matches (falsche Positive) gefunden werden. Im Rahmen der Evaluation wurden dementsprechend zwei Forschungsfragen beantwortet:

RQ1 Wie gut skaliert der Modellaufbau bezüglich der Laufzeit?

RQ2 Wie präzise ist die Datenflussanalyse?

Zur Beantwortung dieser Fragen wurden Messungen und Tests auf einem Notebook mit Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz Prozessor (2 Kerne) und 16 GB Arbeitsspeicher durchgeführt. Die folgenden Unterabschnitte 6.2.1 und 6.2.2 beschreiben, wie die Evaluation der Forschungsfragen *RQ1* bzw. *RQ2* jeweils durchgeführt wurde und welche Ergebnisse dabei beobachtet wurden.

6.2.1 Modellaufbau

Zur Evaluation des Modellaufbaus wurden quelloffene Java-Projekte ansteigender Größe verwendet. Da der Modellaufbau ein für die Datenflussanalyse nötiger Schritt ist, der dem Nutzer ansonsten nur geringen Mehrwert bringt, sollte dessen Laufzeit nicht zu hoch sein. Daher wurden die Ausführungszeiten des Modellaufbaus der gegebenen Java-Projekte erfasst und deren Entwicklung in Abhängigkeit von sechs Modelleigenschaften untersucht, die als Metriken für die Modellgröße verwendet werden können. Die untersuchten Eigenschaften sowie die je Projekt berechneten Werte sind in Tabelle 6.1 aufgeführt. Die Metrik *TLOC* gibt die Gesamtzahl von Codezeilen an (*Total lines of code*). Das zur Berechnung der Metrik verwendete Eclipse Plug-in *Metrics2*[3] erfasst zu dessen Berechnung alle nicht-leeren Zeilen einer Übersetzungseinheit, die keine Kommentarzeilen sind. Die Gesamtanzahl der Klassen, Methoden, Felder und Member (Methoden und Felder), werden jeweils von dem Tool gezählt. Zur Erfassung der Anzahl AST-Knoten wurde ein eigener Zählmechanismus verwendet. Die Ausführungszeiten wurden durch das Hinzufügen von Zeitstempeln im Code gemessen. So wurde die Zeit für den Aufbau des reinen, unverarbeiteten ASTs durch MoDisco, die Laufzeit des Datenfluss-Processings sowie

[3] http://metrics2.sourceforge.net/

die kumulierte Ausführungszeit der restlichen Processings gemessen. Die Messung des Datenfluss-Processings wurde in zwei Teile aufgeteilt. Erstens wurde der Datenflussaufbau gemessen, der einer Behandlung aller AST-Elemente einschließlich dem Speichern der von ihnen ausgelösten intraprozeduralen Datenflüsse entspricht. Als Zweites wurde die Reduktion dieser Flüsse auf die effektiven interprozeduralen Datenflüsse zwischen Membern durchgeführt.

Projekt	TLOC	Klassen	Methoden	Felder	Member gesamt	AST-Knoten
1 JavaSolitaire	1197	25	108	60	168	5834
2 QuickUML	2667	21	171	156	327	11685
3 JSciCalc	5437	127	475	116	591	19831
4 JUnit	5780	116	674	135	809	21013
5 JSSE OpenJDK 8	20892	214	1656	562	2218	69409
6 Gantt	21228	297	1758	1049	2807	89780
7 Nutch	21473	307	1526	696	2222	86277
8 Lucene	25472	296	1841	891	2732	96301
9 log4j	31429	419	2640	643	3283	116815
10 JHotDraw	32434	368	3340	682	4022	129987
11 jEdit	49829	543	2856	1518	4363	172988
12 PMD	53214	936	4971	1506	6477	180965
13 JTransforms	71348	38	664	290	954	464814
14 iTrust	77501	958	6095	2779	8874	340318
15 JabRef	77836	932	4659	2820	7479	317018
16 Xerces	102052	804	7735	2365	10100	374030
17 ArgoUML	135529	1507	11703	2156	13859	453206
18 jfreechart	137917	1027	10946	2190	13136	557392
19 Tomcat	177013	1624	15161	5257	20418	636263
20 Azureus	201527	2267	14518	4785	19303	633878
21 SvnKit	228236	1933	13665	5846	19511	1014737

Tabelle 6.1: Die Zeiten für den Aufbau des modifizierten MoDisco-Modells (AST) mit Datenflüssen

Da die Modelltransformation mittels Triple-Graph-Grammatik im Rahmen dieser Arbeit nur geringfügig erweitert wurde (siehe Unterabschnitt 6.1.2), sind keine signifikanten Anstiege der Ausführungszeit des Modellaufbaus aufgrund dieser Änderungen zu erwarten. Das dem Modellaufbau hinzugefügte Datenfluss-Processing ist deutlich umfangreicher. Entsprechend ist davon auszugehen, dass dessen Einfluss auf die Laufzeit des Modellaufbaus größer als bei der Modelltransformation ist. Daher beschränkt sich die Evaluation des Modellaufbaus auf die Erstellung des AST-basierten Datenflussmodells (vgl. Schritte 1-3 in Abbildung 4.3) ohne dessen Transformation (Schritt 4 in Abbildung 4.3). Der Schwerpunkt bei der Beantwortung der Forschungsfrage *RQ1*, zur Skalierbarkeit des Modellaufbaus, liegt dementspre-

chend auf der Untersuchung des Einflusses des hinzugefügten Datenfluss-Processings auf die Ausführungszeit des Modellaufbaus. Hierbei sollen etwaige Korrelationen zwischen den erfassten Projekteigenschaften aus Tabelle 6.1 und den gemessenen Zeiten identifiziert werden, sodass mögliche Gründe für Laufzeitanstiege ersichtlich werden. Das erwartete Ergebnis dieser Untersuchung ist, dass durch die Datenfluss-Erweiterung im Allgemeinen ein möglichst geringer Anstieg der Laufzeit zustande kommt. Darüber hinaus ist es erforderlich, dass für Ausnahmefälle mit einer unverhältnismäßig hohen Ausführungszeit potenzielle Gründe abgeleitet und diskutiert werden können. Die erfassten Projektmetriken sind in Tabelle 6.1 dargestellt. Eine Auflistung der gemessenen Zeiten in Sekunden ist in Tabelle 6.2 dargestellt. In den Diagrammen 6.8, 6.9, 6.10, 6.11, 6.12 und 6.13, die zur Verbesserung des Leseflusses separat nach diesem Abschnitt dargestellt sind, werden die Zeiten des Datenfluss-Processings mit den Projekteigenschaften aus Tabelle 6.1 in Beziehung gesetzt, um eine Korrelation zwischen etwaigen Anomalien der Laufzeitentwicklung und dem jeweiligen Wert der Projekteigenschaft leichter identifizieren zu können. Schließlich werden alle Zeiten aus Tabelle 6.2 je Projekt, sowohl absolut als auch prozentual, in den Balkendiagrammen 6.6 und 6.7 visualisiert, um den Einfluss der Ausführungszeiten des Datenfluss-Processings auf die Gesamtlaufzeit des Modellaufbaus besser beurteilen zu können.

Die allgemeine Entwicklung der gemessenen Zeiten des Datenfluss-Processings (DF-Aufbau und Reduktion) ist wie erwartet ein allmählicher Anstieg bei ansteigender Größe des jeweiligen Metrikwerts. Daher kann grundsätzlich davon ausgegangen werden, dass das Datenfluss-Processing losgelöst vom restlichen Modellaufbau eine gute Skalierbarkeit besitzt. In zwei Fällen wurde jedoch eine Laufzeit gemessen, die in starkem Widerspruch zu dieser Entwicklung steht. Daher wird im Folgenden ein Erklärungsversuch für diese beiden Ausnahmefälle auf Grundlage der gegebenen Diagramme unternommen.

Der Erste dieser Extremwerte wurde für die Reduktion der Datenflüsse des Modells zum Projekt *JTransforms* gemessen. Die gemessenen 18,599 Sekunden stellen nahezu eine Verzehnfachung des bis zu diesem Punkt in der Auflistung größten gemessenen Zeitwerts (1,945 von *jEdit*) dar. Gleichzeitig handelt es sich insgesamt um den größten gemessenen Wert für die Reduktion und das, obwohl das Projekt *JTransforms* mit nur 71348 Codezeilen über 150000 Codezeilen weniger als das größte untersuchte Projekt umfasst. Entsprechend kann der rapide Anstieg der Ausführungszeit in diesem Fall nicht aufgrund eines Anstiegs der Metrik *TLOC* zustande gekommen sein. Allerdings ist auffällig, dass das Projekt mit nur 38 Klassen und 954 Membern (davon 664 Methoden und 290 Felder) im Vergleich zu anderen Projekten mit einer ähnlichen Größe in Codezeilen sehr niedrige Werte in diesen Kategorien aufweist. Gleichzeitig ist die Anzahl der AST-Knoten bei *JTransforms* signifikant größer als bei Projekten mit ähnlich vielen Codezeilen. Eine mögliche Erklärung ist, dass das Projekt viele Ausdrücke pro Codezeile enthält. Diese These wird durch mehrere Indizien unterstützt. Jeder Programmbefehl kann mehrere Ausdrücke, wie z. B. einen Methodenaufruf, eine Variablenreferenzierung oder die Anwendung eines Operators, enthalten. Diese können sich ebenfalls aus mehreren Teilausdrücken zusammensetzen. Beispielsweise ist die Addition zweier Zahlen ein Ausdruck, dessen

Projekt	AST-Aufbau	Datenfluss-Processing		Restliche Processings
		DF-Aufbau	Reduktion	
1 JavaSolitaire	1.997	0.032	0.062	0.453
2 QuickUML	1.474	0.03	0.084	0.393
3 JSciCalc	3.869	0.031	0.047	0.921
4 JUnit	2.388	0.05	0.056	0.900
5 JSSE OpenJDK 8	2.377	0.065	0.106	1.633
6 Gantt	6.741	0.122	0.325	2.268
7 Nutch	5.793	0.143	1.118	5.732
8 Lucene	5.476	0.187	0.203	2.272
9 log4j	8.501	0.128	0.629	4.208
10 JHotDraw	9.555	0.354	1.493	5.433
11 jEdit	10.859	0.356	1.945	5.282
12 PMD	31.049	0.171	0.577	4.063
13 JTransforms	6.904	0.967	18.599	6.895
14 iTrust	13.512	0.323	0.811	6.706
15 JabRef	14.725	0.207	0.466	7.146
16 Xerces	15.059	0.337	1.107	6.339
17 ArgoUML	24.82	0.374	0.655	11.472
18 jfreechart	20.96	0.568	3.052	10.305
19 Tomcat	22.784	0.974	1.915	21.044
20 Azureus	16.895	0.608	12.059	13.533
21 SvnKit	53.961	0.619	3.606	18.442

Tabelle 6.2: Zeiten in Sekunden für den Aufbau des modifizierten MoDisco-Modells (AST) mit Datenflüssen

linker und rechter Operand ebenfalls Ausdrücke sind. Sowohl Befehle als auch Ausdrücke werden als AST-Knoten erfasst, jedoch treten je Programmzeile potenziell mehr Ausdrücke auf, da Befehle im Gegensatz zu Ausdrücken in der Regel keine Unterteilung in Unterbefehle besitzen. Daher steigt die Anzahl der Codezeilen nicht immer proportional zur Anzahl von Ausdrücken an. Entsprechend spiegelt sich eine große Menge an Ausdrücken nicht zwingend in vielen Codezeilen wider. Bei einer hohen Anzahl an Ausdrücken pro Codezeile steigt demnach nur die Anzahl der AST-Knoten, nicht jedoch die Anzahl der Codezeilen, an. Die geringe Anzahl an Klassen und Membern im Vergleich zur Anzahl Codezeilen und AST-Knoten, lässt darüber hinaus vermuten, dass die einzelnen Klassen und Member in *JTransforms* sehr viel Funktionalität beinhalten und dementsprechend eine schwache Kohäsion besitzen. Obwohl dies, verglichen mit den anderen untersuchten Projekten, durchaus ungewöhnlich ist, lässt sich nach Meinung des Autors keine Verbindung zwischen diesem Umstand und der These vieler Ausdrücke pro Codezeile herstellen. Darüber hinaus kann die Anzahl AST-Knoten nicht alleine ausschlaggebend sein, da Projekte mit ähnlichen Werten bei dieser Metrik, wie z. B. ArgoUML, deutlich geringere Ausführungszeiten der Reduktion aufweisen. Die These, dass ein ungewöhnlich hohes Maß an Ausdrücken für die Anomalie verantwortlich ist, wird hierdurch indirekt unterstützt, da ersichtlich wird, dass nicht nur die hohe Anzahl an AST-Knoten ausschlaggebend sein kann, und somit eine weitere Eigenschaft von *JTransforms* eine Rolle spielen muss. Unterstützend für die These ist, dass bei der Reduktion von Datenflüssen all jene Flüsse erhalten bleiben und auf dem AST als neue Modellelemente eingefügt werden, die durch interprozedurale Zugriffe verursacht werden. Diese Zugriffe sind ausschließlich Ausdrücke. Da jeder Zugriff im Rahmen der Reduktion beim Ableiten der effektiven Flüsse mehreren Verarbeitungsschritten unterzogen wird, sollte ein erhöhtes Vorkommen von Zugriffen definitiv zu einem Anstieg der Ausführungszeit führen. Die erhöhte Laufzeit des Modellaufbaus kann demnach auf eine hohe Anzahl von Ausdrücken bzw. Zugriffen zurückzuführen sein. Da diese Größe nicht erfasst wurde, kann mit den vorhandenen Messwerten kein Beleg gefunden werden.

Um die aktualisierte These, dass eine hohe Anzahl von Zugriffen (als Konkretisierung von Ausdrücken) für die lange Laufzeit verantwortlich sein könnte, dennoch überprüfen zu können, wurden die zu *JTransforms* und *ArgoUML* erstellten Modelle manuell inspiziert. Da bei dem Modellaufbau von *ArgoUML* trotz der ähnlichen Knotenzahl eine deutlich geringere Ausführungszeit gemessen wurde, liegt die Vermutung nahe, dass in diesem Projekt deutlich weniger Zugriffe existieren müssen. Tabelle 6.3 zeigt die Anzahlen der in den beiden Projekten enthaltenen Zugriffe. Mit nur knapp 40 000 zusätzlichen Zugriffen in *JTransforms* fällt die Differenz nicht in der erwarteten Höhe aus. Entsprechend liegt für die Gültigkeit der Ausgangsthese kein starker Indikator vor.

Um schließlich dennoch eine Aussage über eine wahrscheinliche Ursache für die Laufzeitanomalie von *JTransforms* treffen zu können, wurde eine feingranulare Messung der Ausführungszeit der Reduktion durchgeführt, um den Teil des Codes zu identifizieren, der für die Verzögerung verantwortlich ist. Der hierdurch identifizierte Codeteil, welcher tatsächlich ca. 90 % der Ausführungszeit der Reduktion ausmachte,

Zugriffsart	JTransforms	ArgoUML
MSingleVariableAccess	166137	84870
MMethodInvocation	22837	57657
MClassInstanceCreation	1152	7218
MSuperConstructorInvocation	0	845
MSuperMethodInvocation	0	772
MConstructorInvocation	0	155
Alle	190126	151517

Tabelle 6.3: Übersicht der in den Modellen der Projekte *JTransforms* und *ArgoUML* enthaltenen Zugriffe

enthält eine Iteration über alle Flussknoten mit einer Prüfung des Typs des zugehörigen AST-Elements. Ein Flussknoten ist ein Objekt, das die ausgehenden und eingehenden Datenflüsse eines AST-Elements speichert. Für jeden AST-Elementtyp, der nicht auf der in Abschnitt 4.3 angegebenen Whitelist steht, wird eine Löschung des zugehörigen Flussknotens sowie aller mit ihm verbundenen Flussreferenzen durchgeführt. Dies wird mit einer verschachtelten For-Schleife erreicht, welche offenbar ineffizient implementiert ist, sodass die Ausführungszeit bei Modellen mit sehr vielen zu löschenden Flussknoten und -kanten sehr stark ansteigt.

Zur Bestätigung des Verdachts, dass das häufigere Durchlaufen einer ineffizienten For-Schleife für den ungewöhnlichen Laufzeitanstieg verantwortlich ist, wurde die feingranulare Laufzeitmessung ebenfalls auf dem Projekt mit dem zweiten Extremwert (*Azureus*) durchgeführt. Auch hier hat der Codeausschnitt mit der verschachtelten For-Schleife ca. 90 % der Laufzeit der Reduktion eingenommen. Der geäußerte Verdacht wird somit als bestätigt angesehen.

Die Balkendiagramme 6.6 und 6.7 veranschaulichen deutlich, dass das Datenfluss-Processing im Normalfall nur einen geringen Teil ausmacht und die kumulative Ausführungszeit aller Processings nicht dominiert. Im Allgemeinen zeigt der implementierte Modellaufbau dementsprechend eine gute Skalierbarkeit. An den Beispielen von *JTransforms* und *Azureus* ist jedoch zu erkennen, dass es in der Implementierung der Reduktion von Datenflüssen innerhalb des Datenfluss-Processings noch Raum zur Optimierung gibt. Immerhin macht die Ausführungszeit der Reduktion beim Projekt *Azureus* in etwa die Hälfte der Laufzeit aller Processings aus und beim Projekt *JTransforms* sogar in etwa die Hälfte der gesamten Ausführungszeit des Modellaufbaus. Eine Optimierung an der identifizierten Stelle ist folglich sinnvoll, um die Anwendbarkeit auf reale Projekte in Zukunft zu verbessern.

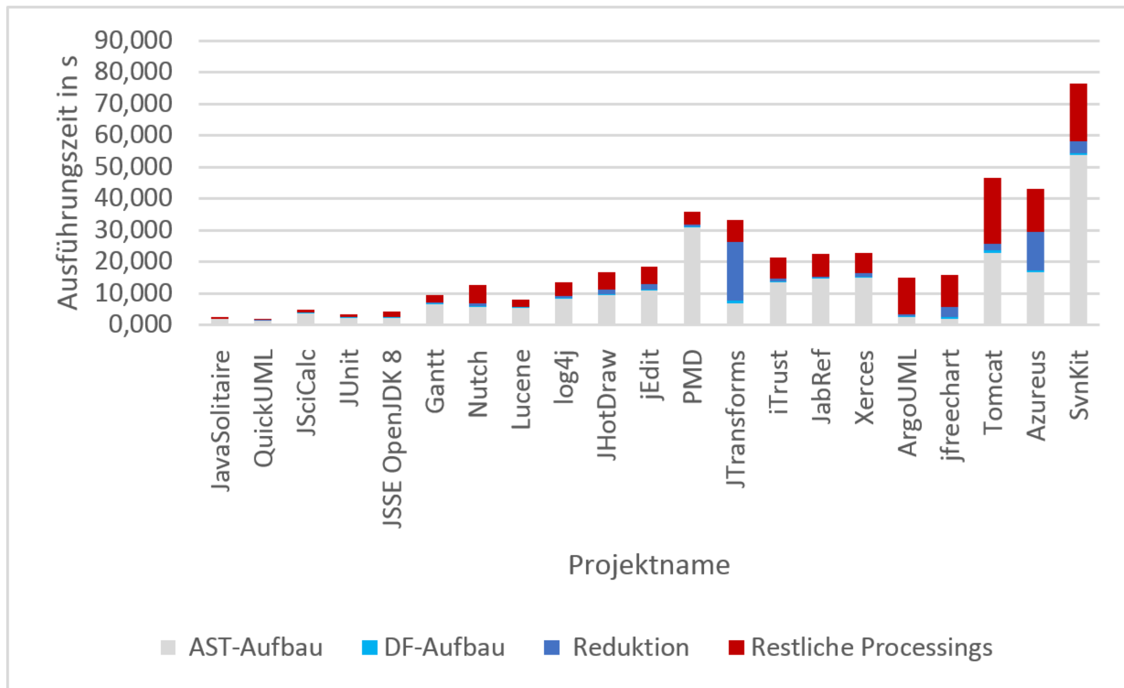

Abbildung 6.6: Ausführungszeiten der einzelnen Schritte des Modellaufbaus in Sekunden

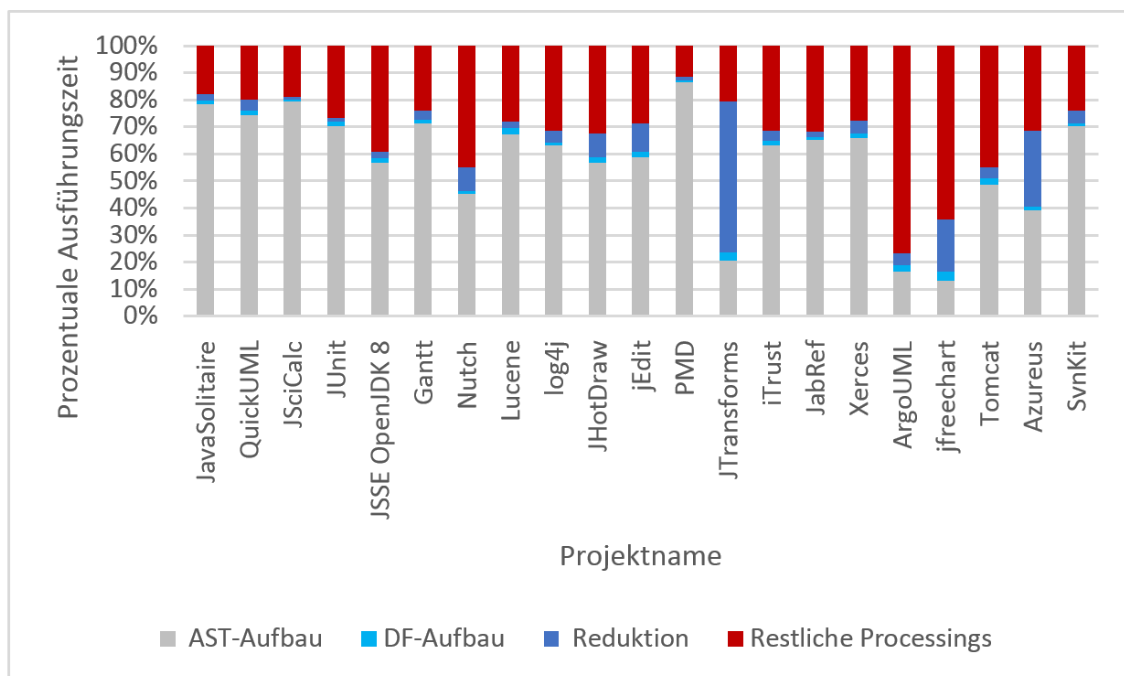

Abbildung 6.7: Prozentuale Ausführungszeiten der einzelnen Schritte des Modellaufbaus

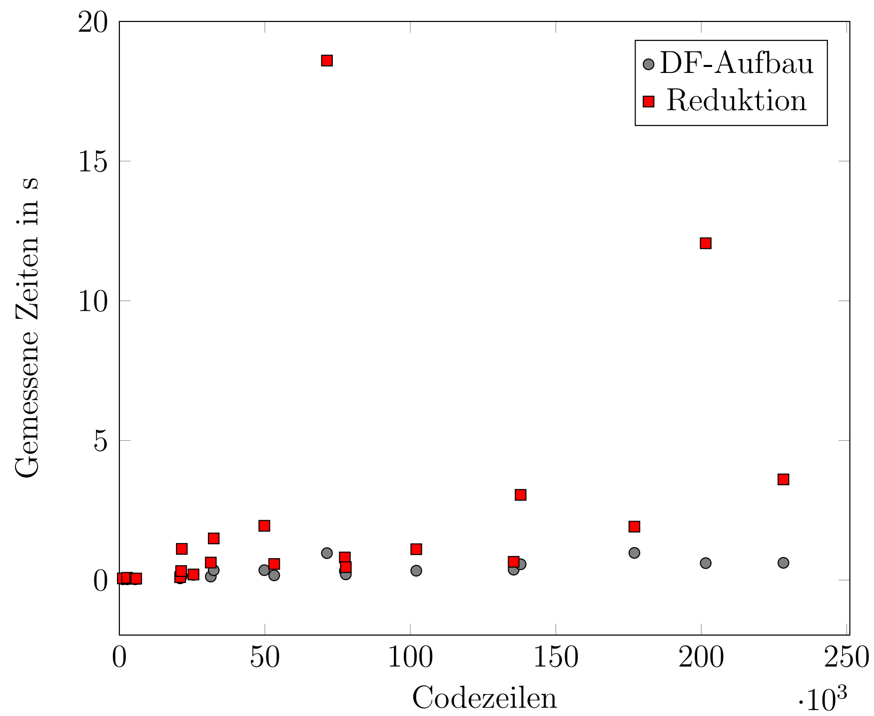

Abbildung 6.8: Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Codezeilen

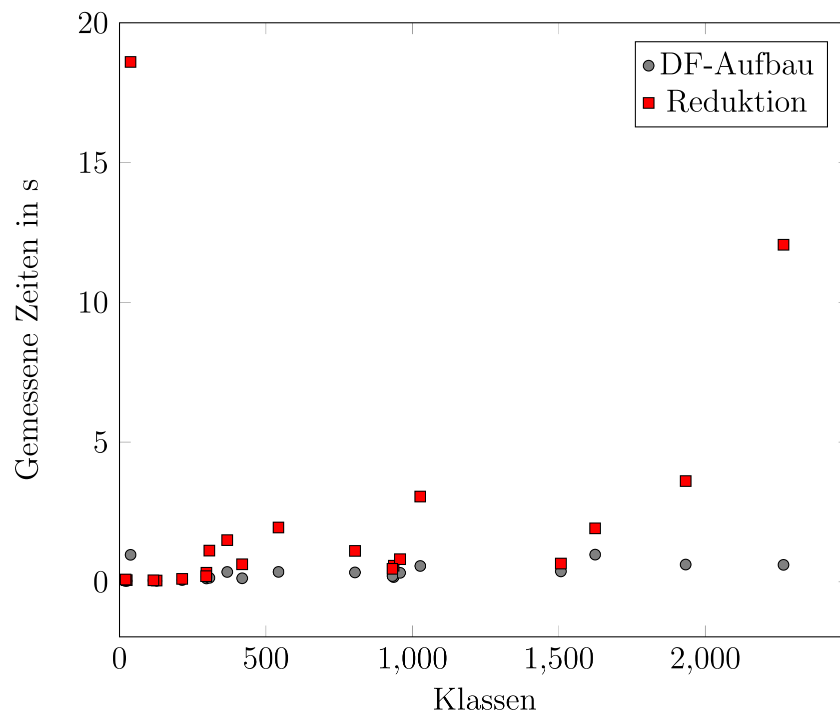

Abbildung 6.9: Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Klassen

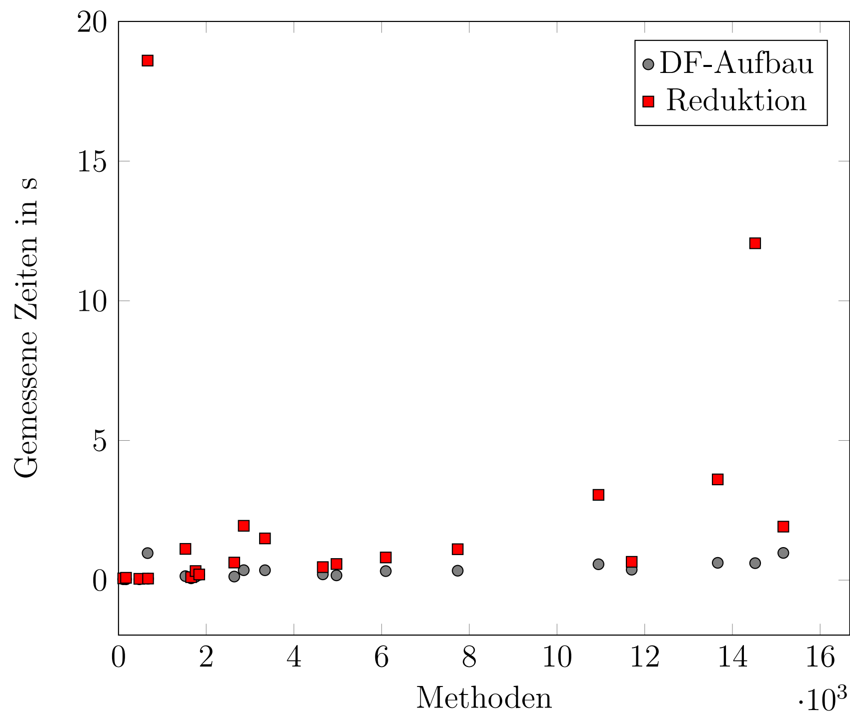

Abbildung 6.10: Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Methoden

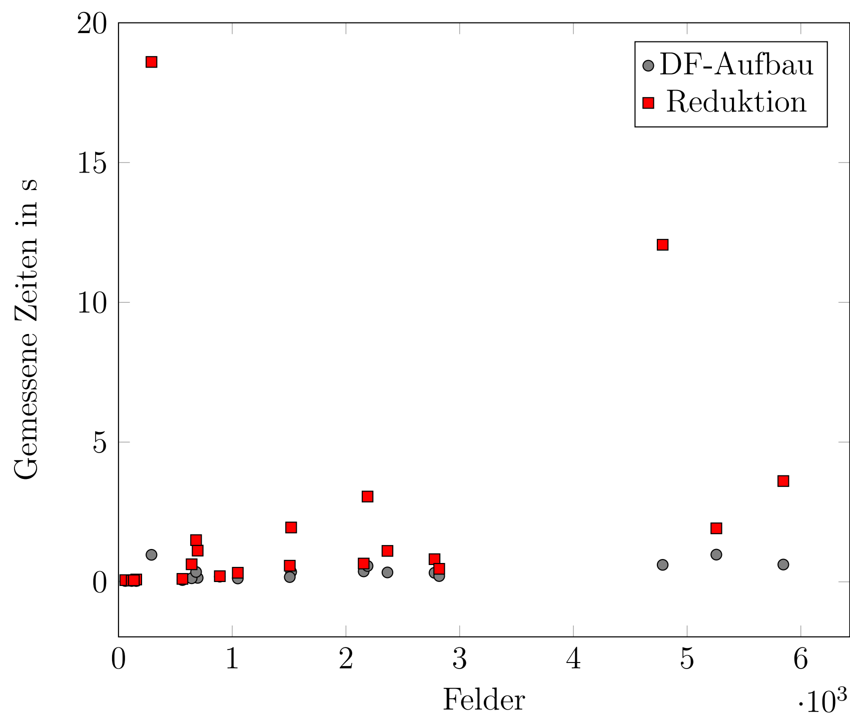

Abbildung 6.11: Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Felder

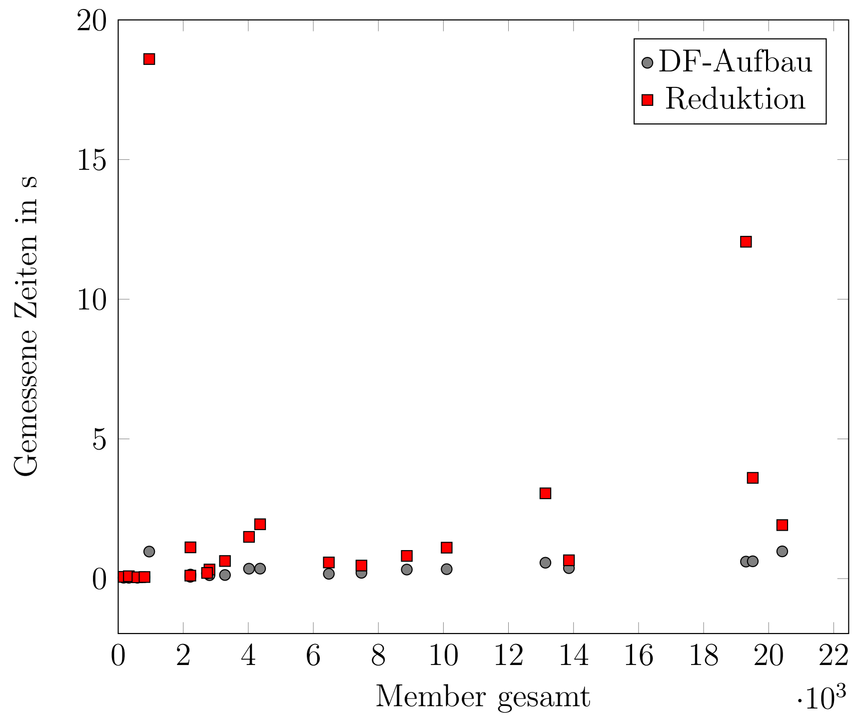

Abbildung 6.12: Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl Member

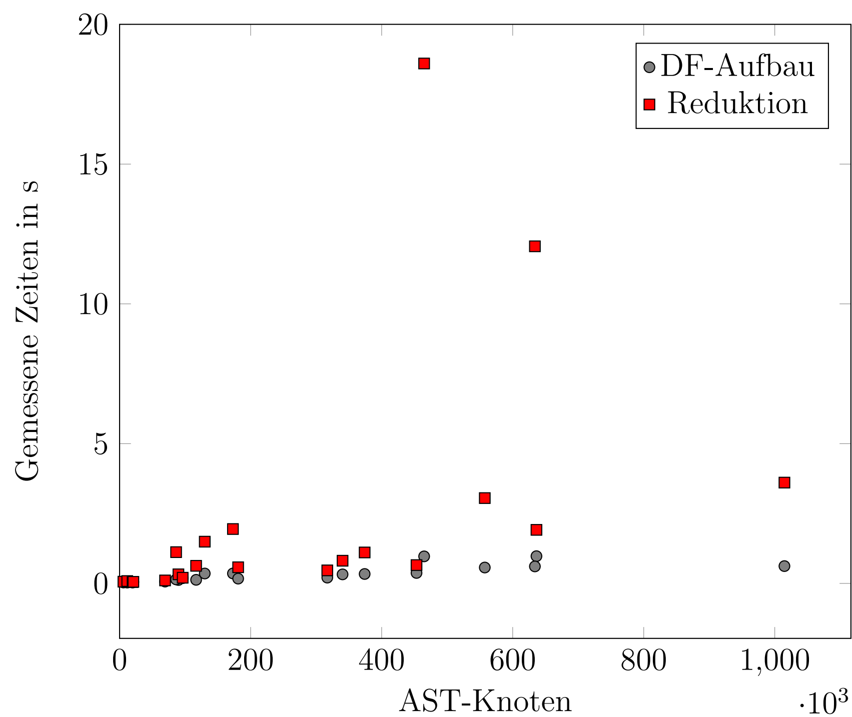

Abbildung 6.13: Ausführungszeiten von Datenflussaufbau und Reduktion in Abhängigkeit von der Projektgröße in Anzahl AST-Knoten

6.2.2 Datenflussanalyse

Zur Evaluation der in den Abschnitten 5.2 und 5.3 vorgestellten Konsistenzprüfungen für Sicherheitsannotationen wurden die Beispiele für Inkonsistenzen (Violations) aus dem für diese Arbeit erstellen Repository mit Datenflussbeispielen[4] verwendet. Diese decken elementare Datenflüsse zwischen mit den Sicherheitsannotationen `@Secrecy` und `@Integrity` annotierten Membern ab, die eine Sicherheitsverletzung darstellen. Beispielsweise wird der Fluss eines vertraulichen und mit `@Secrecy` annotierten Felds in ein Feld ohne `@Secrecy`-Annotation dargestellt. Durch das Anwenden einer Konsistenzprüfung mithilfe einer GReQL-Anfrage zur Identifikation solcher Inkonsistenzen auf alle Violation-Beispiele konnten alle in den Beispielen eingebauten Violations gefunden werden. Darüber hinaus wurden keine vermeintlichen Violations (False Positives) gefunden. Die Berechnung der in [DG06] diskutierten Qualitätsmaße für Suche *Precision* und *Recall* für die Datenflussbeispiele (siehe Gleichungen 6.1 und 6.2) liefert entsprechend in beiden Fällen den optimalen Wert 1. *Precision* gibt den Anteil korrekt gefundener Treffer an der Gesamtzahl von Treffern und *Recall* den Anteil korrekt gefundener Treffer an allen auffindbaren Treffern an.

$$precision = \frac{|Tatsächliche\ Violations \cap Gefundene\ Violations|}{|Gefundene\ Violations|} = \frac{7}{7} = 1 \quad (6.1)$$

$$recall = \frac{|Tatsächliche\ Violations \cap Gefundene\ Violations|}{|Tatsächliche\ Violations|} = \frac{7}{7} = 1 \quad (6.2)$$

Da die Beispiele bei Weitem nicht die Komplexität realer Java-Projekte widerspiegeln, sind zukünftige Tests im Rahmen echter Anwendungen notwendig. Dennoch zeigen die erfolgreichen Tests der Violation-Beispiele die grundsätzliche Anwendbarkeit des vorgestellten Ansatzes zur Identifikation unerwünschter Datenflussmuster in Java-Programmen.

Darüber hinaus erwies sich der vorgestellte Ansatz als geeignet, um das Sicherheitsproblem einer fehlenden Eingabebereinigung aus dem durchgehenden Beispiel (siehe Listing 2.1) zu identifizieren (siehe Abschnitt 5.2). Hierdurch hat sich gezeigt, dass auch komplexere Datenflussmuster mit mehreren Annotationen gefunden werden können. Eine realistische Problemstellung konnte somit bereits gelöst werden und eine weiterführende Anwendung des Ansatzes kann zukünftig dazu beitragen, die Angriffssicherheit komplexer Softwareprojekte durch ein erhöhtes Maß an Automatisierung von Sicherheitsprüfungen zu erhöhen.

Dennoch ist die Verwendung von GReQL-Anfragen für die Datenflussanalyse noch nicht optimal. Zum einen sind die Anfragen in GReQL rein textueller Form. Die erhoffte Einfachheit der Datenflussanalyseanfragen lässt sich allerdings besser durch eine grafische Notation, ähnlich der Anfragediagramme aus Kapitel 5, erreichen. Zum anderen konnten die in den Diagrammen dargestellten Anfragen nicht 1:1 in GReQL-Anfragen übertragen werden, sodass die in GReQL implementierten

[4] https://github.com/GRaViTY-Tool/dataflow-examples

Anfragen sich als komplexer herausstellten (siehe Listing 6.1). Dies ist insbesondere der Tatsache geschuldet, dass das zugrundeliegende Datenflussmodell nur die effektiven Flüsse enthält und keine zusätzlichen Informationen beinhaltet, welche die Analysen vereinfachen würden. Beispielsweise können bei Methodenaufrufen mit Argumentübergabe, zusätzlich zu Flüssen in einen oder mehrere Parameter, Flüsse in alle Member eingefügt werden, welche die zu den Parametern gehörige Signatur implementieren. Hierdurch würde sich die GReQL-Anfrage erheblich verkürzen, da nicht mehr entlang mehrerer Kanten von Parametern zu Signaturen bzw. Membern navigiert werden müsste. Die Umsetzung einer grafischen Notation würde dementsprechend ebenfalls von dieser Änderung profitieren, da auch hier keine Unterscheidung zwischen Flüssen in die Parameter von Membern und sonstigen Flüssen in Member getroffen werden müsste. Folglich wäre die grafische Notation näher an der Notation der Anfragediagramme aus Kapitel 5. Da die vorherige Erstellung des Datenflussmodells vor dem Testen konkreter Anfragen erforderlich war, wurden die genannten Optimierungsmöglichkeiten erst am Ende der Bearbeitungszeit offengelegt. Durch das Einfügen zusätzlicher Informationen im Modellaufbau kann in Zukunft jedoch bereits die Verwendung von GReQL erheblich vereinfacht werden. Anschließend kann die Erstellung einer geeigneten grafischen Anfragesprache Gegenstand weiterer Arbeiten sein und somit die Anwendbarkeit der Datenflussanalyse weiter erhöhen.

7 Fazit

Im Rahmen dieser Masterarbeit wurde ein Datenflussmodell spezifiziert und aufgebaut, auf dem eine Datenflussanalyse auf einem hohen Abstraktionsniveau zur Identifikation von Sicherheitsproblemen objektorientierter Programme durchgeführt wurde. Auf diesem Weg konnte das Problem einer unzureichenden Eingabebereinigung insofern gelöst werden, dass eine Datenflussanalyse zur Erkennung derartiger Sicherheitsprobleme umgesetzt wurde. Die Analyse fand dabei mit vom Tool GReQL ausgewerteten Anfragen statt, in denen zu identifizierende unzulässige Datenflussmuster auf einem hohen Abstraktionslevel spezifiziert wurden. Für den hohen Abstraktionsgrad ist die im Rahmen der Arbeit neu entwickelte Datenflussmodellform verantwortlich. Diese abstrahiert zur Komplexitätsreduzierung von den Programmdetails auf Anweisungsebene und legt den Fokus auf Zugriffe und die dadurch ausgelösten Datenflüsse zwischen Membern objektorientierter Programme. Es können in GReQL-Anfragen somit vergleichsweise einfache Datenflussmuster spezifiziert werden, die keine für interprozedurale Analysen irrelevanten Zwischenschritte, wie z. B. Flüsse in lokale Variablen, beinhalten. Hierdurch ist es dem Anwender möglich, sich auf effektive interprozedurale Datenflüsse zwischen Membern zu konzentrieren, ohne auf Fachwissen aus der Datenflussdomäne angewiesen zu sein.

Es wurde eine neue Form eines Datenflussmodells entwickelt, die bezüglich ihrer Granularität einen Mittelweg zwischen den bewährten Darstellungen des Datenflussgraphen und des Datenflussdiagramms darstellt. Eine ähnlich komprimierte und dennoch in ihrer Mächtigkeit für jede interprozedurale Datenflussart ausreichende Modellform ist dem Autor nicht bekannt. Darüber hinaus können, dank dieser Modellform, Datenflussanalysen auf einem hohen Abstraktionsniveau durchgeführt werden, wodurch auch Entwicklern objektorientierter Software ohne Expertise in der Datenflussanalyse diese hilfreiche Form statischer Analyse zugänglich gemacht wird. Ein ähnlicher Ansatz für Datenflussanalysen, der keine spezifischen Kenntnisse des Themengebiets voraussetzt, ist dem Autor ebenfalls nicht bekannt. Entsprechend stellt die vorliegende Arbeit die Präsentation sowohl einer neuartigen Datenflussmodellform als auch eines bisher ungesehenen Datenflussanalyseansatzes dar. Die Evaluation der Ergebnisse hat darüber hinaus erste Hinweise darauf gegeben, dass der Ansatz in der Praxis durchaus Anwendung finden kann. So zeichnet sich der Aufbau des Datenflussmodells durch eine hohe Skalierbarkeit aus und die konkrete Datenflussanalyse in Form einer Konsistenzprüfung von Sicherheitsannotationen wurde erfolgreich mithilfe minimaler Beispiele von Sicherheitsverletzungen sowie einem realistischen Beispiel der Schwäche CWE-20 getestet.

Da die Evaluation der Konsistenzprüfung sich bislang auf Tests mit sehr rudimentären Beispielen beschränkt hat, sind zukünftige Tests mit realen Projekten not-

wendig, um die Anwendbarkeit des Ansatzes weiter zu evaluieren und potenziellen Optimierungsbedarf offenzulegen. Weiterhin bietet der Ansatz einige Möglichkeiten, in Zukunft vielfältige Erweiterungen und Optimierungen umzusetzen. Beispielsweise kann, basierend auf der für die Modelltransformation eingesetzten Triple-Graph-Grammatik, eine inkrementelle Ko-Evolution von Datenflussmodell und Quellcode ermöglicht werden. Hierdurch könnte die Ausführungszeit eines wiederholten Modellaufbaus nach Änderungen im Quellcode des zu analysierenden Projekts möglicherweise signifikant verringert werden. Dies wäre insbesondere für die Anwendbarkeit auf großen Projekten hilfreich, da ein einzelner Modellaufbau hier mehrere Minuten dauern kann. Darüber hinaus kann als weitere Optimierung z. B. eine Referenzanalyse implementiert werden, um die momentan zur Abdeckung polymorpher Aufrufe aufgestellte Überapproximation von Datenflüssen zu minimieren. Hierdurch sollte die Identifikation vermeintlich schädlicher Datenflüsse bei der Konsistenzprüfung verringert werden können. Die Formulierung von Anfragen für die Datenflussanalyse kann des Weiteren durch die Umsetzung einer grafischen Anfragesprache erleichtert werden. In dieser Arbeit wurden bereits mehrere Beispiele einer grafischen Notationsform in Form von Anfragediagrammen vorgestellt, welche als Vorlage für eine grafische Anfragesprache dienen können. Der vorgestellte Ansatz schafft somit die Grundlage für kommende spannende Arbeiten, die dazu beitragen können, dass das Potenzial einer Datenflussanalyse objektorientierter Programme auf einem hohen Abstraktionsniveau ausgeschöpft wird.

Literaturverzeichnis

[ABD+15] Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. An introduction to metamodelling and graph transformations with emoflon. Hier abrufbar: https://emoflon.org/eclipse-plugin/release/handbook/part4.pdf, 2015. Letzter Abruf: 31.08.2019.

[AC76] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[Bod] Eric Bodden. Heros ifds/ide solver. Hier abrufbar: https://github.com/Sable/heros. Letzter Abruf: 31.08.2019.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[Dat] National Vulnerability Database. Cve-2011-3544 detail.

[DeM79] Tom DeMarco. Structure analysis and system specification. In *Pioneers and Their Contributions to Software Engineering*, pages 255–288. Springer, 1979.

[DG06] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[DK82] Alan L Davis and Robert M Keller. Data flow program graphs. 1982.

[EGH+14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[GJS+19] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The java language specification, java se 12 edition. Hier abrufbar: https://docs.oracle.com/javase/specs/jls/se12/html/index.html, 2019. Letzter Abruf: 31.08.2019.

[GS] Brian Gorenc and Fritz Sands. Hacker machine interface. Hier abrufbar: https://documents.trendmicro.com/assets/wp/wp-hacker-machine-interface.pdf. Letzter Abruf: 31.08.2019.

[GS10] H.P. Gumm and M. Sommer. *Einführung in die Informatik*. Oldenbourg Verlag, 9th edition, 2010.

[KBB86] Krishna M. Kavi, Bill P. Buckles, and U. Narayan Bhat. A formal definition of data flow graph models. *IEEE Transactions on computers*, (11):940–948, 1986.

[LBLH11] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[LHBM14] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–108. ACM, 2014.

[MRC07] Jan Mendling, Hajo A Reijers, and Jorge Cardoso. What makes process models understandable? In *International Conference on Business Process Management*, pages 48–63. Springer, 2007.

[Neu18] Alexander Neumann. State of software security report: Devsecops bietet mehr sicherheit und effizienz. Hier abrufbar: https://www.heise.de/developer/meldung/State-of-Software-Security-Report-DevSecOps-bietet-mehr-Sicherheit-und-Effizienz-4257190.html, Dezember 2018. Letzter Abruf: 31.08.2019.

[PKLS15] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. Incremental co-evolution of java programs based on bidirectional graph transformation. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 138–151, New York, NY, USA, 2015. ACM.

[PKLS16] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 578–589, 2016.

[Rep11] Metasploit GitHub Repository. Oracle java applet rhino script engine remote code execution. Hier abrufbar: https://github.com/rapid7/metasploit-framework/blob/master/external/source/exploits/CVE-2011-3544/Exploit.java, 2011. Letzter Abruf: 31.08.2019.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[Rie13] Eike Hagen Riedemann. *Testmethoden für sequentielle und nebenläufige Software-Systeme*. Springer-Verlag, 2013.

[Sch] Michael Schierl. Cve-2011-3544 / zdi-11-305 – oracle java applet rhino script engine remote code execution. Hier abrufbar: http://schierlm.users.sourceforge.net/CVE-2011-3544.html. Letzter Abruf: 31.08.2019.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[Stö18] Harald Störrle. On the impact of size to the understanding of uml diagrams. *Software & Systems Modeling*, 17(1):115–134, 2018.

[VK83] Victor L Voydock and Stephen T Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys (CSUR)*, 15(2):135–171, 1983.

[WQH+18] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *Proceedings of the 40th International Conference on Software Engineering*, pages 37–47. ACM, 2018.