Heinrich Heine
Universität
Düsseldorf

Institut für Informatik
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Generierung eines Abhängigkeitsgraphen für Python

Florian Peter Leander Mager

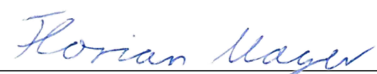
Bachelorarbeit

Beginn der Arbeit: 13. November 2019
Abgabe der Arbeit: 13. Februar 2020
Gutachter: Univ.-Prof. Dr. Michael Leuschel
Dr. John Witulski

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 13. Februar 2020



Florian Peter Leander Mager

Zusammenfassung

Software-Erosion spielt eine große Rolle in Hinblick auf die großen Wartungskosten von Software. Als Erosionsschutz werden automatisierte statische Code-Analysen eingesetzt. Die Axivion Bauhaus Suite stellt diese Analysen bereits für einige Programmiersprachen zur Verfügung. Das Ziel dieser Arbeit war die Generierung eines Abhängigkeitsgraphen im RFG-Format für die Sprache Python, um diese Analysen auf Python-Programmen durchführen zu können. Für die Generierung des Abhängigkeitsgraphen wurde ein Schema für diesen konzipiert und diskutiert. Unter Einhaltung dieses Schemas wurde das Programm py2rfg zur Generierung des RFG implementiert und sein Aufbau beschrieben. Das Ziel der Arbeit wurde erreicht und tote Code-, Zyklen- und Architekturanalysen lassen sich aussagekräftig auf den generierten RFG anwenden.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Vom Bauhaus-Projekt zur Axivion Bauhaus Suite . . . 1

1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **2**

2.1 Python . . . 2

2.1.1 Dynamische Eigenschaften . . . 2

2.1.2 Aufbau eines Programms . . . 3

2.1.3 Namensräume . . . 4

2.1.4 Das Lesen und Schreiben von Variablen . . . 4

2.1.5 Das Importsystem . . . 5

2.2 Resource-Flow-Graph . . . 6

2.2.1 Existierende RFG-Schemata . . . 7

3 Konzeption eines RFG-Schemas für Python **8**

3.1 Python RFG-Schema . . . 8

3.2 Attribute . . . 8

3.2.1 *Linkage*-Attribute . . . 8

3.2.2 *Soure*-Attribute . . . 9

3.2.3 *Element*-Attribute . . . 9

3.2.4 *Metric*-Attribute . . . 10

3.3 Knoten- und Kantentypen . . . 10

3.3.1 *Module*/*Module_Call* und *Package* . . . 11

3.3.2 *Class* . . . 12

3.3.3 *Function* . . . 12

3.3.4 *Method* 12
3.3.5 *Variable* und *Member* 13
3.4 Wieso ist der Knotentyp *Type* nicht im RFG enthalten? 13
3.5 Views 13
3.5.1 *Code-Facts* 13
3.5.2 *File* 14
3.5.3 *Import* 14
3.5.4 *Call* 14
3.5.5 *Entry* 15

4 Generierung des Python-RFG **15**
4.1 Eingabe 15
4.2 Schritte der RFG Generierung 16
4.3 Analyse von Programmen 16
4.3.1 Knoten in *py2rfg* 16
4.3.2 Analyse des Dateisystems 19
4.3.3 Analyse der Module 20
4.3.4 Generierung eindeutiger *Linkage*-Namen 21

5 Auswertung **23**
5.1 Korrektheit 23
5.2 Analysen 23
5.2.1 Zyklenanalyse 23
5.2.2 Tote Code-Analyse 25
5.2.3 Architekturanalyse 26

6 Fazit **27**
6.1 Ausblick 27

6.1.1 Erweiterung des RFG 27

Abbildungsverzeichnis 28

Tabellenverzeichnis 28

Literatur 30

1 Einleitung

1.1 Motivation

Heutzutage verursachen Wartung und Evolution von Softwaresystemen mehr als 90% der Kosten, die im Software-Lebenszyklus anfallen [Kos03]. Eine große Bedeutung spielt dabei die Software-Erosion. Dieser Begriff wurde von der Axivion GmbH geprägt und bezeichnet den steten und schleichenden Verfall der inneren Software-Struktur [axia].
Die Axivion Bauhaus Suite [axib] ist eine Toolsuite zur automatisierten statischen Code-Analyse zur Bekämpfung von Software-Erosion. Ihre Analysewerkzeuge werden meistens nicht direkt auf dem Programm angewendet sondern auf einem Resource-Flow-Graph (RFG), der aus dem Quelltext generiert wird. Der RFG stellt einen Abhängigkeitsgraphen dar und kann bisher von der Axivion Bauhaus Suite für die Programmiersprachen C, C++, C# und Java generiert werden.
Ziel dieser Arbeit ist die Generierung eines Abhängigkeitsgraphen im RFG-Format für die sehr populäre Programmiersprache Python [tio]. Als Folge sollen Werkzeuge der Axivion Bauhaus Suite, für die der RFG die Analyse Grundlage darstellt, auf Python-Programme anwendbar sein. Zu diesen gehören Werkzeuge zur Architekturprüfung, Erkennung toten Codes, Zyklenerkennung und Metrikanalyse.

1.2 Vom Bauhaus-Projekt zur Axivion Bauhaus Suite

Das Bauhaus-Projekt hat zum Ziel, das Programmverstehen auf Code- und Architekturebene durch analytische Methoden und Werkzeuge zu erleichtern, um damit die Qualität und Effizienz der Wartungsprozesse zu verbessern. Die Forschungsschwerpunkte bilden Software-Redundanz, Software-Qualität, Architektur Rekonstruktion & Konformanzprüfung und Software-Produktlinien. Außerdem bildet es den Rahmen für vielfältige Forschungsaktivitäten im Bereich des Programmverstehens, der Software-Qualität und Wartungsprozessen im Allgemeinen [pro].
1996 wurde das Bauhaus-Projekt als Forschungsprojekt des Instituts für Softwaretechnologie (ISTE) der Universität Stuttgart [stu] in Kooperation mit dem Frauenhofer Institut für experimentelles Software-Engineering ins Leben gerufen. Durch die Berufung des Mitbegründers Rainer Koschke an die Universität Bremen, ist seit 2004 die Arbeitsgruppe Softwaretechnik des Fachbereichs 3 der Universität Bremen [bre] Teil des Bauhaus-Projekts. Die Firma Axivion GmbH [axia] wurde 2006 von Mitarbeitern des Forschungsprojekts als Spin-Off, zur Vermarktung von Forschungsprototypen der Universitäten als marktreife Produkte, gegründet. Heutzutage ist das Bauhaus-Projekt eine Kooperation mit der Universität Bremen und der Firma Axivion GmbH [Kos08].
Obwohl die Axivion Bauhaus Suite [axib] ihren Ursprung im Bauhaus-Projekt hat, ist sie ein eigenständiges Produkt, das kommerziell von Axivion vertrieben wird. Sie dient als Komplettlösung für Software-Erosionsschutz, indem sie eine Toolsuite zur automatisier-

ten statischen Code-Analyse bereitstellt.

1.3 Aufbau der Arbeit

Nach dieser Einleitung teilt sich die Arbeit in sechs weitere Kapitel auf. Im folgenden Kapitel 2 *Grundlagen* werden die für diese Arbeit notwendigen Aspekte der Programmiersprache Python und des Resource-Flow-Graphen vorgestellt. Das anschließende Kapitel 3 *Konzeption eines RFG-Schemas für Python* befasst sich mit den Bestandteilen des zu generierenden RFG und Besonderheiten des Schemas für Python. Kapitel 4 *Generierung des Python-RFG* gibt die Programmspezifikationen an und erläutert die einzelnen Schritte bei der Generierung des RFG. Weiter wird der grundlegende Aufbau des Programms vorgestellt und es wird auf besondere Implementierungsdetails eingegangen. In Kapitel 5 *Auswertung* wird die Korrektheit des generierten RFGs dargelegt. Außerdem wird der RFG in Hinsicht auf die Analysen der Axivion Bauhaus Suite kritisch betrachtet. Das letzte Kapitel 6 *Fazit* fasst den Nutzen des generierten RFG zusammen und gibt einen Ausblick für die Erweiterung des RFG und die weitere Nutzung dieser Arbeit.

2 Grundlagen

2.1 Python

Python [pyt] ist eine imperative, objektorientierte, dynamische Programmiersprache, die von Guido van Rossum entwickelt wurde und kostenlos zur Verfügung steht. Alle Versionen sind Open Source. Die erste Version 0.9.0 erschien im Jahr 1991.

2.1.1 Dynamische Eigenschaften

Für diese Arbeit ist es von großer Bedeutung, dass Python nicht wie C, C++, C# und Java, die bisher von der Axivion Bauhaus Suite unterstützt werden, statisch sondern dynamisch ist. Dazu gehört, dass Variablen in Python keinen Typ besitzen, sondern die Objekte auf die sie referenzieren. Weiter wird die Einsatzfähigkeit eines Objekts nicht durch seinen Typ bestimmt sondern durch seine Methoden und Attribute. Ein Beispiel hierzu ist in Listing 1 zu sehen. Diese Art der Typisierung wird Duck-Typing genannt, das eine Anwendung des Ententests darstellt (*If it walks, swims and quaks like a duck - it is a duck*). Außerdem lassen sich in Python Objekte zur Laufzeit modifizieren. Im Listing 2 wird einer Klasse zur Laufzeit ein Attribut hinzugefügt.

```
>>> class Ente:
...     def fliegen(self):
...         print("Ente␣fliegt.")
...
>>> class Eule:
...     def fliegen(self):
...         print("Eule␣fliegt.")
...
>>> class Pinguin:
...     def schwimmen(self):
...         print("Pinguin␣schwimmt.")
...
>>> for vogel in Ente(), Eule(), Pinguin():
...     vogel.fliegen()
...
Ente fliegt.
Eule fliegt.
AttributeError: 'Pinguin' object has no attribute 'fliegen'
```

Listing 1: Beispiel zu Duck-Typing

```
>>> class Foo:
...     pass
...
>>> Foo.var
AttributeError: type object 'Foo' has no attribute var
>>> Foo.var = "Hallo␣Welt!"
>>> Foo.var
'Hallo␣Welt!'
```

Listing 2: Attribut wird zur Laufzeit hinzufügt

2.1.2 Aufbau eines Programms

Python-Programme sind aufgebaut aus Codeblöcken. Diese sind Code, der als Einheit ausgeführt wird. Für Python-Programme stellen Codeblöcke dabei Module, Funktionsrümpfe und Klassendefinitionen dar.

2.1.3 Namensräume

Codeblöcke besitzen einen Namensraum, der Variablen speichert. Diese stellen eine Bindung von einem Bezeichner zu einem Objekt dar. Wird eine Variable in einem Codeblock definiert, so ist sie eine lokale Variable des Blocks. Lokale Variablen eines Moduls sind zudem globale Variablen. Namensräume ermöglichen es, Variablen mit dem gleichen Namen in verschiedenen Codeblöcken zu nutzen, ohne dass diese sich beeinflussen. Neben den Namensräumen von Codeblöcken existiert noch ein eingebauter Namensraum, der eingebaute Funktionen wie *print()*, *type()* oder *isinstance()* überall zur Verfügung stellt. Namensräume besitzen einen Gültigkeitsbereich, in dem direkt auf die Variablen im Namensraum zugegriffen werden kann. Tabelle 1 gibt einen Überblick über die einzelnen Gültigkeitsbereiche.

Zugehöriges Objekt	Gültigkeitsbereich	Bezeichnung
Interpreter	Gesamtes Programm	Eingebauter Namensraum
Modul	Ganzes Modul	Globaler Namensraum
Funktion	Funktion und alle inneren Blöcke	Lokaler Namensraum
Klasse	Klasse	Lokaler Namensraum

Tabelle 1: Namensräume

2.1.4 Das Lesen und Schreiben von Variablen

Im Gegensatz zu anderen Programmiersprachen wie Java unterscheiden sich in Python das Schreiben und Lesen von Variablen.
Für das Lesen einer Variablen in einem Codeblock wird zuerst der Namensraum des aktuellen Blocks nach der Variablen durchsucht. Falls diese dort nicht gefunden wird, erfolgt die Suche in den Namensräumen aller umgebenden Funktionen, beginnend mit der direkt umgebenden Funktion. Anschließend findet die Suche im globalen Namensraum und zuletzt im eingebauten Namensraum statt.
Beim Schreiben einer Variablen in einem Codeblock wird diese standardmäßig im Namensraum des Blocks gespeichert.
Zu beachten gibt es noch, dass falls in einem Block eine lokale Variable erstellt wird, alle Benutzungen des Variablenamens auf die lokale Variable verweisen. Durch dieses Verhalten können Fehler wie in Listing 3 entstehen.

```
>>> foo = 3
>>> def fun():
...     print(foo)
...     foo = 5
...
>>> fun()
UnboundLocalError: local variable 'foo' referenced before
assignment
```

Listing 3: Fehler durch ungebundene lokale Variable

Um in Codeblöcken globale Variablen benutzen zu können, gibt es die *global-Anweisung*. Eine lokale Variable der direkt umgebenden Funktion kann mithilfe der *nonlocal-Anweisung* benutzt werden. Die *global*/*nonlocal*-Anweisung sorgt dafür, dass beim Lesen und Schreiben auf die entsprechende Variable zugegriffen wird. Die Anweisungen müssen vor der ersten Benutzung des Bezeichners erfolgen, aber nicht zu Beginn des Codeblocks. Dies führt auf den ersten Blick zu einem seltsamen Verhalten in Listing 4, da der else-Block auch auf die globale Variable zugreift.

```
>>> var = None
>>> if False:
...     global var
...     var = "foo"
... else:
...     var = "bar"
...
>>> print(var)
bar
```

Listing 4: *global*-Anweisung in *if-else*

2.1.5 Das Importsystem

Das Importsystem dient dazu, Zugriff auf den Namensraum eines anderen Moduls zu erhalten. Für die Suche eines Moduls wird dessen vollqualifizierter Name benötigt. Dieser ergibt sich aus dem Modulnamen und den Namen der umgebenden Pakete durch Punkte getrennt. Pakete sind auch Module, dienen aber zusätzlich zur Strukturierung und Bereitstellung einer Namenshierarchie für Module. Der Unterschied zwischen Modulen und Paketen ist, dass Pakete ein Attribut *__path__* besitzen. Der vollqualifizierte Name für das Modul *baz* im Paket *bar*, das im Paket *foo* liegt, wäre *foo.bar.baz*. Für diesen versucht Python zuerst *foo*, dann *foo.bar* und zuletzt *foo.bar.baz* zu importieren.
Das Suchen eines Namens erfolgt dabei als erstes in *sys.modules*, das als Cache für

bereits importierte Module dient. Falls das Modul *foo.bar.baz* importiert wurde, erhält *sys.modules* Einträge für *foo*, *foo.bar* und *foo.bar.baz*. Jeder dieser Einträge verweist auf das zugehörige Modul-Objekt, das bei einer erfolgreichen Suche zurückgegeben wird.
Falls der Name nicht in *sys.modules* gefunden wurde, werden zunächst die eingebauten Module und dann die eingefrorenen Module durchsucht. Eingefrorene Module sind selbständig ausführbare Module für Unix-Systeme. Zuletzt werden alle Einträge des Importierpfades durchsucht, der meistens *sys.path* entspricht. Wird das Modul in den eingebauten Modulen, eingefrorenen Modulen oder Einträgen des Importierpfades gefunden, so muss noch ein entsprechendes *module*-Objekt für diese erzeugt werden, was auch Laden genannt wird. Das erzeugte *module*-Objekt wird dann noch in *sys.modules* gespeichert, bevor es vom Importsystem zurückgegeben wird.
Das hier vorgestellte Verhalten des Importsystems ist das Normalverhalten. Da das Importsystem jedoch fast in allen Aspekten erweiterbar ist, gilt dieses Verhalten nicht für alle Python-Programme.
Für gewöhnlich wird der Importvorgang über eine import-Anweisung in Gang gesetzt. Diese sorgt neben dem Import auch für die Bindung im aktuellen Codeblock, wie in Tabelle 2 zu sehen ist.

Import-Anweisung	Import	Bindung
import foo	foo	foo
import foo.bar.baz	foo.bar.baz	foo
import foo.bar.baz as qux	foo.bar.baz	foo.bar.baz als qux
from foo.bar import baz	foo.bar.baz	foo.bar.baz als baz
from foo import attr	foo	foo.attr als attr

Tabelle 2: Bindungen verschiedener import-Anweisungen (nach dem Beispiel in der Python-Dokumentation)

2.2 Resource-Flow-Graph

[Axic] Der Resource-Flow-Graph (RFG) stellt globale Programmkomponenten und ihre Abhängigkeiten als Graph dar und ist Bestandteil der Axivion Bauhaus Suite. Er basiert auf einem gerichteten Graph, der um folgende Bestandteile erweitert wurde: Alle Knoten und Kanten besitzen einen Typ, der sich hierarchisch strukturieren lässt. Er enthält sogenannte Views, die benannte Teilgraphen darstellen. Die Knoten, Kanten, Views und der Graph selbst können attributisiert werden. Attribute sind typisiert und alle verfügbaren Typen sind in Tabelle 3 dargestellt. Die Visualisierung und auch Modifizierung eines RFG kann mit Gravis erfolgen, der auch Bestandteil der Axivion Bauhaus Suite ist.
Außerdem enthält jeder RFG ein Schema. Dieses legt fest, welche Knoten- und Kantentypen es gibt und wie diese hierarchisch angeordnet sind. Weiter legt es fest, welche

Kantentypen gerichtet zwischen Knotentypen existieren dürfen. Für die Attribute speichert das Schema, welchem RFG-Element diese zugewiesen werden können.

Typ	Attributwerte
toggle	Ein Attribut vom Typ toggle besitzt keinen Wert und ist entweder an ein Element gebunden oder nicht.
int	Ganzzahlen im Bereich von -2147483648 bis 2147483647.
float	Gleitkommazahlen im Bereich von -3.40282E+38 bis 3.40282E+38.
string	beliebig lange Zeichenketten, die UTF-8 kodiert sind.

Tabelle 3: RFG-Attributtypen

2.2.1 Existierende RFG-Schemata

Die Dokumentation der Axivion Bauhaus Suite enthält RFG-Schemata für C, C++ und C#. Neben diesen existieren noch weitere RFG-Schemata in wissenschaftlichen Arbeiten. Eine Übersicht über jene bietet die Tabelle 4.

Sprache	Dokumentation
C	[axib]
C++	[axib]
C#	[axib]
Java	[Bay13]
Visual Basic 6	[Har06]
Cobol	[Mül04]
Ada95	[Nei04]

Tabelle 4: Existierende RFG-Schemata

3 Konzeption eines RFG-Schemas für Python

3.1 Python RFG-Schema

Das von mir konzipierte RFG-Schemata für Python ist aufgebaut auf dem Schema von C. Dies hat folgende Gründe:

- Das Schema ist konformer mit den anderen Schemata, da viele auf dem C-Schema aufbauen.
- Die Darstellung von neu erstellten Knoten- und Kantentypen in Gravis kann von mir nicht beeinflusst werden. Die Darstellung von Knoten- und Kantentypen des C-Schemas ist in Gravis allerdings definiert.
- Gravis erwartet für die Durchführung von Analysen die Existenz von vielen Knoten- und Kantentypen des C-Schemas.

3.2 Attribute

In diesem Abschnitt gehe ich auf die im RFG verwendeten Attribute ein. Attribute besitzen Präfixe, die sie einer bestimmten Bedeutung zuordnen. Die verwendeten Präfixe mit ihrer Bedeutung sind in Tabelle 5 zu finden und sind einheitlich mit den existierenden Schemata.

Präfix	Bedeutung
Linkage	Informationen für den Verknüpfungsprozess
Source	Informationen zur Quelle
Element	Element spezifische Informationen
Metric	Metrikinformationen (Codezeilen, Kommentarzeilen, Halstead, ...)

Tabelle 5: Bedeutung der Attributpräfixe

3.2.1 *Linkage*-Attribute

Die Axivion Bauhaus Suite bietet Funktionen zur Nutzung mehrerer RFGs an. Zum Beispiel ist das Vereinen von mehreren RFGs mit dem Kommando *rfgmerge* möglich. Damit dies funktioniert müssen die Knoten eines RFG die Attribute *Linkage.Name* und *Linkage.Is_Definition* besitzen.

Das Attribut *Linkage.Name* ist vom Typ *string* und soll für jeden Knoten einen eindeutigen Namen speichern.
Linkage.Is_Definition ist vom Typ *toogle*. Das Vorhandensein bedeutet, dass das Element aufgrund einer Definition erstellt wurde. Fehlt es, so wurde das Element nur durch das Benutzen eines Symbols im Code erstellt.

3.2.2 *Soure*-Attribute

Tabelle 6 zeigt eine Liste aller verwendeten *Source*-Attribute. Das Attribut *Source.Name* speichert die Zeichenkette, die in Gravis als Elementname angezeigt wird. Über das Setzen der restlichen *Source-Attribute* ist es in Gravis möglich, direkt zur definierten Codestelle für eine genauere Betrachtung zu springen.

Attribut	Typ	Bedeutung
Source.Name	string	Name
Source.Path	string	Pfad
Source.File	string	Datei
Source.Line	int	Zeile
Source.Column	int	Spalte

Tabelle 6: *Source*-Attribute

3.2.3 *Element*-Attribute

Das Attribut *Element.Is_Artificial* ist vom Typ *toggle*. Das Vorhandensein bedeutet, dass das RFG-Element vom Programm erzeugt wurde, ohne dass dieses Element eine Grundlage im Dateisystem oder Quellcode hat.
Des Weiteren existiert noch das Attribut *Element.Is_Import_All* vom Typ *toggle*. Es kann für *Import*-Kanten gesetzt werden, was bedeutet, dass diese alle Variablen aus dem Namensraum des Ziels importieren. Der RFG enthält noch weitere *Element*-Attribute, die aber alle spezifischer für ein RFG-Element sind. Die Einführung und Erläuterung dieser Attribute erfolgt im Abschnitt des jeweiligen RFG-Elements.

3.2.4 *Metric*-Attribute

Im Python-RFG werden keine *Metric*-Attribute gesetzt, da es den zeitlichen Rahmen dieser Bachelorarbeit übersteigen würde.

3.3 Knoten- und Kantentypen

In Tabelle 8 sind alle Obertypen des Schemas mit ihrer Bedeutung und ihren direkten Untertypen aufgelistet. Knoten der Obertypen existieren im RFG nicht. Die Darstellung von Programmkomponenten im RFG ist in Tabelle 7 aufgelistet. Zusätzlich sind in dieser Tabelle auch die Zielknoten der auslaufenden *Enclosing*-Kanten für die Knotentypen festgehalten. Falls der Startknoten einer *Enclosing*-Kante von *File_System_Entity* erbt, so ist die Bedeutung der *Enclosing*-Kante, dass der Startknoten innerhalb des Zielknoten liegt. Erbt der Startknoten von *Source_Entity*, so wird der Startknoten innerhalb des Zielknoten definiert. Die restlichen Kantentypen mit ihrer Bedeutung und ihren Start- und Zielknoten sind in Tabelle 9 aufgelistet. Im Folgenden gehe ich auf Knotentypen ein, die Besonderheiten aufweisen.

Programmkomponente	Knotentyp	Enclosing-Kante zu
Verzeichnis	Directory	Directory, Package
Datei	File	Directory, Package
Paket	Package	Directory, Package
Modul	Module/Module_Call	Directory Package
Klasse	Class	Package, Module, Class, Function, Method
Funktion	Function	Package, Module, Function, Method
Methode	Method	Class
Globale Variable	Variable	Package, Module
Klassenvariable	Variable	Class
Instanzvariable	Member	Class

Tabelle 7: Knotentypen

Knotentyp	Bedeutung	Unterknoten
File_System_Entity	Ursprung Dateisystem	Directory, File, Package, Module
Source_Entity	Ursprung Quellcode	Class, Routine
Routine	aufrufbar	Function, Method, Variable, Member, Module_Call

Tabelle 8: Knotenobertypen

Name	Startknoten	Bedeutung	Zielknoten
Import	Package, Module, Class, Function, Method	importiert	Package, Module, Class, Function, Method, Variable, Member
Call	Package, Module/Module_Call, Function, Method	ruft auf	Package, Module/Module_Call, Function, Method, Variable, Member
Set	Package, Module, Class, Function, Method	setzt	Variable, Member
Use	Package, Module, Class, Function, Method	benutzt	Package, Module, Class, Function, Method, Variable, Member
Inherit	Class	erbt von	Class

Tabelle 9: Kantentypen - Start- und Zielknoten

3.3.1 *Module*/*Module_Call* und *Package*

Der Typ *Module_Call* wird statt *Module* im *Call*- und *Entry*-View verwendet. Der Grund dafür wird in Abschnitt 3.5.4 *Call*-View erläutert.
Die Rückgabe des Importsystems für eine *import*-Anweisung sollte ein Modul/Paket sein. Da das Importsystem aber erweiterbar gestaltet ist, muss dies nicht immer der Fall sein. Dies wird in Listing 5 demonstriert. Als Resultat können die Typen *Module* und *Package* einlaufende *Call*-Kanten besitzen.

```
def fun():
        print("foo")

import sys
sys.modules['fun'] = fun

import fun
fun()
```

Listing 5: Import liefert Objekt vom Typ function

3.3.2 *Class*

In Konformität mit den existierenden Schemata wird für den Aufruf einer Klasse keine *Call*-Kante zur Klasse sondern zu den Spezialmethoden *__new__()* und *__init__* erstellt. Dies hat den Grund, dass bei einem Klassenaufruf nicht der Code im Klassenrumpf ausgeführt wird. Der Klassenrumpf wird jedoch direkt bei der Klassendefinition ausgeführt. Daher werden Aufrufe innerhalb des Klassenrumpfes dem nächstumgebenden Codeblock, der keine Klasse ist, zugeordnet.

3.3.3 *Function*

Der Knotentyp *Function* wurde neu erstellt und erbt von *Routine*.

3.3.4 *Method*

Method-Knoten können die Attribute in Tabelle 10 besitzen. Diese listet zusätzlich den Typ und die Bedeutung des Attributs auf.

Attribut	Typ	Bedeutung
Element.Is_Static	toggle	Statische Methode
Element.Is_Classmethod	toggle	Klassenmethode
Element.Is_Getter	toggle	Methode mit **@property** Dekorator
Element.Is_Setter	toggle	Methode mit **@*.setter** Dekorator

Tabelle 10: Method-Knotenattribute

3.3.5 *Variable* und *Member*

Eine Besonderheit des Python-Schemas zu den existierenden Sprachen stellt dar, dass *Variable*- und *Member*-Knoten einlaufende *Call*-Kanten besitzen können. Der Grund dafür ist, dass in Python Funktionen in Variablen gespeichert werden können. Auslaufende *Call*-Kanten können sie nicht besitzen, da statisch nicht ermittelt werden kann, was für ein Objekt in einer Variablen zum Aufruf gespeichert ist.

3.4 Wieso ist der Knotentyp *Type* nicht im RFG enthalten?

Wie bereits in Abschnitt 2.1.1 erwähnt, besitzen Variablen in Python keinen Typ. Dennoch lassen sich ihnen durch eine statische Analyse Typen zuordnen, wie diese Arbeit [Sal04] zeigt. Der Aufwand dafür übersteigt allerdings den zeitlichen Rahmen dieser Arbeit. Daher habe ich mich dafür entschieden, Typen nicht in das Schema aufzunehmen.

3.5 Views

3.5.1 *Code-Facts*

Die *Code-Facts*-View ist die Basissicht des RFG und enthält alle architekturrelevanten RFG-Elemente. *File-Knoten* und *Directory-Knoten* sind nicht enthalten. Eine Ausnahme bilden künstliche *Directory-Knoten*, die zur Strukturierung von Abhängigkeiten des Programms, wie etwa der Standard Bibliothek in die View aufgenommen wurden. In Tabelle 11 sind alle möglichen *Enclosing*-Kanten nach ihrem Startknoten aufgelistet. Die restlichen Kanten sind in Tabelle 12 dargestellt.

Startknoten	Zielknoten
Directory	/
Package	Directory, Package
Module	Directory, Package
Class	Package, Module, Class, Function, Method
Function	Package, Module, Function, Method
Method	Class
Variable	Package, Module, Class
Member	Class

Tabelle 11: Code-Facts-View - Enclosing-Kanten

Kante	Startknoten	Zielknoten
Import, Call, Use	Package, Module, Class, Function, Method	Package, Module, Class, Function, Method, Variable, Member
Set	Package, Module, Class, Function, Method	Variable, Member
Inherit	Class	Class

Tabelle 12: Code-Facts-View - Kanten (ohne Enclosing)

3.5.2 *File*

Die *File*-View dient als hierarchische Sicht des Programms und enthält nur Kanten vom Typ *Enclosing*. An Knoten kommen in ihr alle mit dem Attribut *Linkage.Is_Definition* vor.

3.5.3 *Import*

Die *Import*-View dient zur Visualisierung von Importen. Dazu enthält sie nur *Import*- und *Enclosing*-Kanten. Ihre Knotenmenge gleicht der der *Code-Facts*-View.

3.5.4 *Call*

Die *Call*-View dient als Aufrufsicht des RFG und enthält deshalb nur *Call*-Kanten. Die vorkommenden Knotentypen sind *Function*, *Method*, *Variable*, *Member* und *Module_Call*. Sie dient als Eingabe für die tote Code- und Zyklenanalyse. Diese Analysen berücksichtigen aber nur den Typ *Routine* und seine Untertypen. Daher wurde die Vererbung von *Variable* und *Member* so geändert, dass sie von *Routine* erben. Für den Typ *Module* war dies nicht möglich, da er von *File_System_Entity* erbt. Daher wurde der Knotentyp *Module_Call* zum Schema hinzugefügt. Dieser erbt von *Routine* und stellt den Typ *Module* im *Call* und *Entry*-View dar. Da es im *Call*-View keine Hierarchie gibt, ergeben Pakete wenig Sinn. Daher werden von Paketen nur die *__init__.py* Module im *Call* und *Entry*-View abgebildet.

3.5.5 *Entry*

In Konformität mit den existierenden Schemata soll die *Entry*-View die Einstiegspunkte eines Programms enthalten. Weiter dient sie als eine Eingabe für die Analyse von totem Code. Daher werden in der *Entry*-View *Module*-Knoten als *Module_Call*-Knoten dargestellt. Der genaue Grund ist im vorherigen Abschnitt 3.5.4 *Call*-View erläutert. Ein Einstiegspunkt für ein Python-Programm ist ein Modul mit dem Namen __*main*__. Weiter kann jedoch prinzipiell jedes Modul als Einstiegspunkt gewählt werden. Beim Start soll aber häufig ein anderer Code als beim Importieren ausgeführt werden. Wird ein Modul als Einstiegspunkt genutzt, so wird sein Name auf __*main*__ gesetzt, auch wenn dieser vorher anders war. Module, die bewusst als Einstiegspunkt dienen sollen, überprüfen deshalb meistens, ob ihr Name __*main*__ ist und führen, falls ja, einen anderen Code aus. Um die *Entry*-View sinnvoll zu halten, habe ich daher die Entscheidung getroffen, nur Module in die *Entry*-View aufzunehmen, deren Name __*main*__ ist oder die ihren Namen auf __*main*__ testen.

4 Generierung des Python-RFG

Für die Generierung eines RFG aus einem Python-Programm habe ich *py2rfg* [py2] geschrieben. Dieses habe ich als reines Python-Programm implementiert, da eine Python Installation auf einem System, auf dem ein RFG aus einem Python-Programm generiert werden soll, wahrscheinlich ist. Insbesondere ist die reine Implementierung in Python möglich, da die Axivion Bauhaus Suite eine Python Schnittstelle für die RFG-Bibliothek besitzt. Ein weiterer Vorteil ist, dass *py2rfg* sich somit selber analysieren kann. Die Anbindung der Axvion Bauhuas Suite an py2rfg erfolgt über das Setzen einer Umgebungsvariablen. Die verwendete Python Version entspricht der Version 3.7.6, da dies die aktuellste Version (Stand 13.02.2020) ist, die mit der mir zur Verfügung gestellten Version der Axivion Bauhaus Suite kompatibel ist.

4.1 Eingabe

Als Eingabe werden folgende Pfade erwartet:

1. Programmpfad, der zum analysierenden Python-Programm führt
2. Ausgabepfad, der zum Ablageort für den generierten RFG führt
3. Beliebig viele Abhängigkeitspfade, die zu Python-Programmen führen, von denen das zu analysierende Programm abhängt

4.2 Schritte der RFG Generierung

Die Generierung des RFG gliedert sich in folgende Schritte:

1. Erstellung eines neuen RFGs und Definition seines Schemas
2. Analyse der Standard Bibliothek
3. Analyse der Pfade zu den Abhängigkeiten
4. Analyse des Programms
5. Speichern des generierten RFG als *.rfg* Datei unter dem Ausgabepfad

Die erstellte *.rfg* Datei besitzt dabei als Namen den Basisnamen des Eingabepfads.

4.3 Analyse von Programmen

Im Folgenden ist die Analyse von Programmen beschrieben, die für die Standard-Bibliothek, die Abhängigkeiten und das eigentlich zu analysierende Programm exakt gleich aussehen. Die Analyse eines Programms teilt sich in die Analyse des Dateisystems und in die Analyse der Module auf.

4.3.1 Knoten in *py2rfg*

Damit in den Analysen nicht berücksichtigt werden muss, wie genau ein Knoten erstellt wird und wie bestimmte Kanten und Attribute gesetzt werden, habe ich diese Funktionalität in die Knoten ausgelagert. Zu diesem Zweck existiert für jeden Knotentyp eine Klasse. Außerdem wurden abstrakte Superklassen, die gemeinsame Funktionen von Knotentypen implementieren, angelegt, um Code-Redundanz zu vermeiden. Weiter können diese Superklassen dazu genutzt werden, Abfragen an bestimmte Eigenschaften von Knoten zu stellen. So kann die Abfrage, ob ein Knoten *Package*-Knoten enthalten kann, mit *isinstance(knoten, ContainsPackages)* statt *type(Knoten) in (Directory, Package)* erfolgen. Dies ist besonders sinnvoll bei Eigenschaften, die viele, aber nicht alle, Knotentypen teilen.
Um nicht komplett eine eigene Knotendarstellung entwickeln zu müssen und am Ende alle Informationen nochmal in den RFG zu schreiben, baut meine Knotendarstellung auf der Knotendarstellung der RFG-Schnittstelle auf. Dies geschieht in der Klasse *Node*, die Superklasse von allen Knotentyp-Klassen ist. Da eine Vererbung durch die Schnittstelle nicht möglich ist, wurde dies durch eine Komposition gelöst. So wird bei der Initialisierung eines Knoten ein entsprechender RFG-Knoten erstellt und in einer Instanzvariablen

gespeichert.
Die Abbildung 1 zeigt alle Knotentyp-Klassen mit ihren relevanten Attributen und Methoden. Zusätzlich sind ihre Superklassen bis auf *Node* und *ABC* abgebildet, die Klasse *ABC* dient zur Erstellung abstrakter Klassen. Einen Überblick über die Attribute und Methoden der Klasse *Node* gibt Tabelle 13. Durch diese Knotendarstellung lässt sich im restlichen Programm leicht ein Knoten in einem anderen Knoten erstellen oder sich eine Kante zwischen zwei Knoten ziehen. Dies verdeutlicht das Codebeispiel 6 aus *py2rfg*. Für jeden Knoten und jede Kante wird, falls möglich, Quellcode über die *Source*-Attribute angebunden.

```
def visit_ClassDef(self, node):
        name = node.name
        process_keywords(node.keywords)

        cls = self.code_block.create_class(name, node.lineno,
                                           node.col_offset)
```

Listing 6: Codebeispiel zur Erstellung eines *Class*-Knotens aus *_code_block_visitor.py*

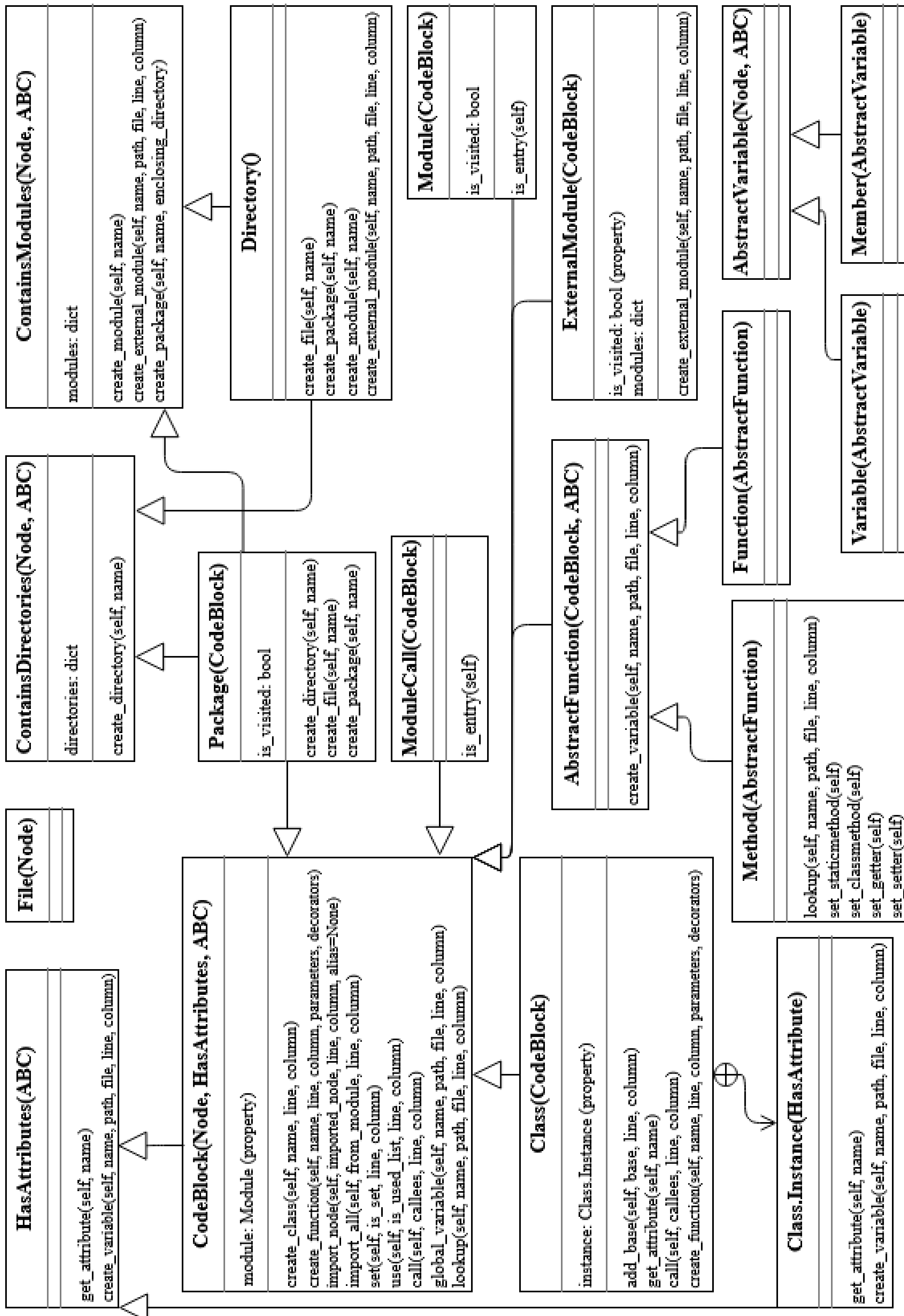


Abbildung 1: Klassendiagramm Knotentypen

Typ	Name
instance variable	enclosed_nodes
property	rfg
property	node
property	node_type
property	views
property	enclosing_node
property	full_path
property and setter	name
property and setter	linkage_name
property and setter	path
property and setter	file
property and setter	line
property and setter	column
property and setter	is_definition
property and setter	set_artificial(self)
method	_insert_edge(self, edge_type, target, line, column)
method	_insert_enclosing_edge(self, enclosing_node, attributes=None)

Tabelle 13: Schnittstelle Klasse Node

4.3.2 Analyse des Dateisystems

Die Analyse des Dateisystems überprüft, ob der Programmpfad zu einem Verzeichnis, Paket oder Modul führt und legt einen entsprechenden Knoten an. Dieser wird als Ausgangsknoten für das Programm gespeichert. Führt der Pfad zu einem Verzeichnis oder Paket, so werden für alle Einträge unter diesem Pfad die entsprechenden Knoten angelegt. Dabei kann es sich um die Knotentypen *Directory*, *Package*, *File* und *Module* handeln. Für alle Einträge der gefundenen Verzeichnisse und Pakete wird gleich verfahren, außer dass bei Verzeichnissen Pakete/Module als Verzeichnisse/Dateien behandelt werden. Dies hat den Grund, dass die Pakete und Module nicht ohne weiteres importiert werden können und es sich wahrscheinlich um Testdaten handelt, die irrelevant für die Analyse sind.

4.3.3 Analyse der Module

Nach der Analyse des Dateisystems steht eine Liste von Modulen (inklusive Paketen) zur Verfügung, die bei der Analyse der Module schrittweise durchgearbeitet wird. Dies erfolgt in dem Paket *visit_modules*. Dazu wird für jedes Modul ein abstrakter Syntaxbaum (engl. abstract syntax tree (AST)) erzeugt. Ein AST bildet den Quelltext als abstrakte Grammatik in Baumdarstellung ab. Bei Paketen wird hierzu die *__init__.py* Datei verwendet. Das Erzeugen der ASTs übernimmt das *ast* Modul der Standard Bibliothek [ast]. Mithilfe der Klasse *NodeVisitor* des *ast* Moduls lässt sich über den AST laufen. Starten lässt sich dies mit dem Aufruf der Methode *visit()* mit einem Knoten des AST. Die Methode *visit()* ruft, falls vorhanden, die entsprechende Besuchermethode für den AST-Knoten auf, deren Name aus *visit_* und dem Klassennamen des AST-Knoten zusammengesetzt wird. Für eine Funktionsdefinition heißt die entsprechende Besuchermethode *visit_FunctionDef*. Falls für einen Knoten keine Besuchermethode definiert ist, wird die Methode *generic_visit* aufgerufen. Diese ruft wiederum *visit()* für alle Kindsknoten auf. Somit wird jeder Knoten des ASTs besucht, wenn keine Besuchermethoden definiert sind.
Die von mir definierten Besuchermethoden habe ich zur Strukturierung auf die Klassen *CodeBlockVisitor*, *CallSetUseVisitor* und *ImportVisitor* aufgeteilt, wie in Abbildung 2 zu sehen ist. Das Besuchen der AST-Knoten erfolgt getrennt in Codeblöcken. So wird für jeden Codeblock ein eigenes Objekt der Klasse *CodeBlockVisitor* erstellt. Gestartet wird das Laufen über den AST durch den Aufruf der Methode *visit_body(self, body)* in *CodeBlockVisitor*.
Die Klasse *CodeBlockVisitor* definiert Besuchermethoden für Klassen- und Funktionsdefinitionen. Die Besuchermethode für Klassendefinitionen sorgt für die Erstellung des entsprechenden Knotens und das Besuchen des Rumpfes der Klassendefinition in einer neuen Instanz von *CodeBlockVisitor*. Die Besuchermethoden für Funktionsdefinitionen regelt die Erstellung des entsprechenden Knotens und fügt diesen Knoten einer Liste von Funktionen des Codeblocks hinzu. Sobald alle AST-Kindsknoten für den übergebenden AST-Knoten besucht wurden, werden alle Funktionsrümpfe in einer neuen Instanz von *CodeBlockVisitor* besucht. Dies hat den Grund, dass der Rumpf von Funktionsdefinitionen im Gegensatz zu Klassendefinitionen nicht direkt ausgeführt wird sondern erst beim Aufruf der Funktion. Deshalb werden Funktionsrümpfe zuletzt besucht, damit in diesen auf alle Klassen, Funktionen und Variablen, die im Module definiert wurden, zugegriffen werden kann. Anzumerken ist, dass falls es zur Definition von Klassen/Funktionen mit dem gleichen Namen kommt, für jede Definition Knoten im RFG angelegt werden. Der Name im RFG der Klassen/Funktion unterscheidet sich dann durch eine Zahl am Ende.
In der Klasse *CodeBlockVisitor* kommt es außerdem zur Überprüfung, ob es sich um ein *main*-Module handelt. Dazu existiert eine Besuchermethode für *if*-Anweisungen, die für den Code ***if*** *__name__ == '__main__':* oder ***if*** *'__main__'== __name__:* das Modul zur *Entry*-View hinzufügt, falls es sich um ein *Module*-Knoten handelt.
In der Klasse *ImportVisitor* werden Besuchermethoden für Importe definiert. Dabei werden die zu importierenden Module wie beim gewöhnlichen Importvorgang gesucht. Dabei können jedoch nur Module gefunden werden, die zuvor bei der Analyse des Dateisystems

erfasst worden sind. Wird das Modul gefunden, so wird für dieses der AST generiert und durchlaufen, falls noch nicht geschehen, bevor mit dem Durchlaufen des ASTs des aktuellen Codeblocks fortgefahren wird. Für Module die nicht gefunden wurden wird ein *Module*-Knoten erstellt, bei dem das Attribut *Linkage.Is_Definition* nicht gesetzt ist. Für Module ohne Paket liegt dieser *Module*-Knoten zur Strukturierung im künstlichen *Directory*-Knoten *external_modules*. Wird in einem nicht gefundenen Modul ein weiteres Modul erwartet, so wird das nicht gefundene Modul zum Paket.
Zum Schluss implementiert die Klasse *CallSetUseVisitor* Besuchermethoden, die sich mit dem Aufrufen, Setzen oder Nutzen von Variablen beschäftigen. Hier gibt es das Problem, welcher Knoten bei Nutzung oder Aufruf eines Namens gemeint ist, da in einem Codeblock gegebenenfalls Klassen/Funktionen/Variablen mit dem gleichen Namen existieren können. Da durch eine statische Analyse nicht immer eine genaue Bestimmung möglich ist, habe ich mich für eine Überapproximation entschlossen. Bei Nutzung/Aufruf eines Namens werden *Use/Set* Kanten für alle Knoten, die mit diesen Namen im Codeblock liegen, angelegt. Das Setzen eines Namens ist jedoch ohne Probleme möglich, da es sich hierbei immer um eine Variable (Globale Variable, Klassenvariable, Instanzvariable) handelt.

4.3.4 Generierung eindeutiger *Linkage*-Namen

Mit *Linkage*-Name ist die Rede vom Wert im Attribut *Linkage._Name*. Da zum Beispiel ein *Class*-Knoten einen *Variable*- und *Member*-Knoten mit dem gleichen Namen enthalten kann, setzt sich das Ende jedes *Linkage*-Namens aus dem Namen und dem Typ des Knotens getrennt durch ein Dollarzeichen zusammen. Dies reicht allerdings noch nicht für eindeutige *Linkage*-Namen aus, da zum Beispiel ein anderer *Class*-Knoten existieren kann, der auch einen *Member*-Knoten mit diesem *Linkage*-Namen besitzt. Ist dies der Fall, so werden jeweils die gleichen *Linkage*-Namen um den Namen des umschließenden Knotens erweitert. Der neue *Linkage*-Name setzt sich dann aus dem Namen und dem alten *Linkage*-Namen getrennt durch einen Punkt zusammen. Dies geschieht solange bis der *Linkage*-Name eindeutig ist. Als Voraussetzung für die Generierung eindeutiger *Linkage*-Namen ergibt sich aber, dass sich bei der Analyse von Programmen der Programmname unterscheiden muss.

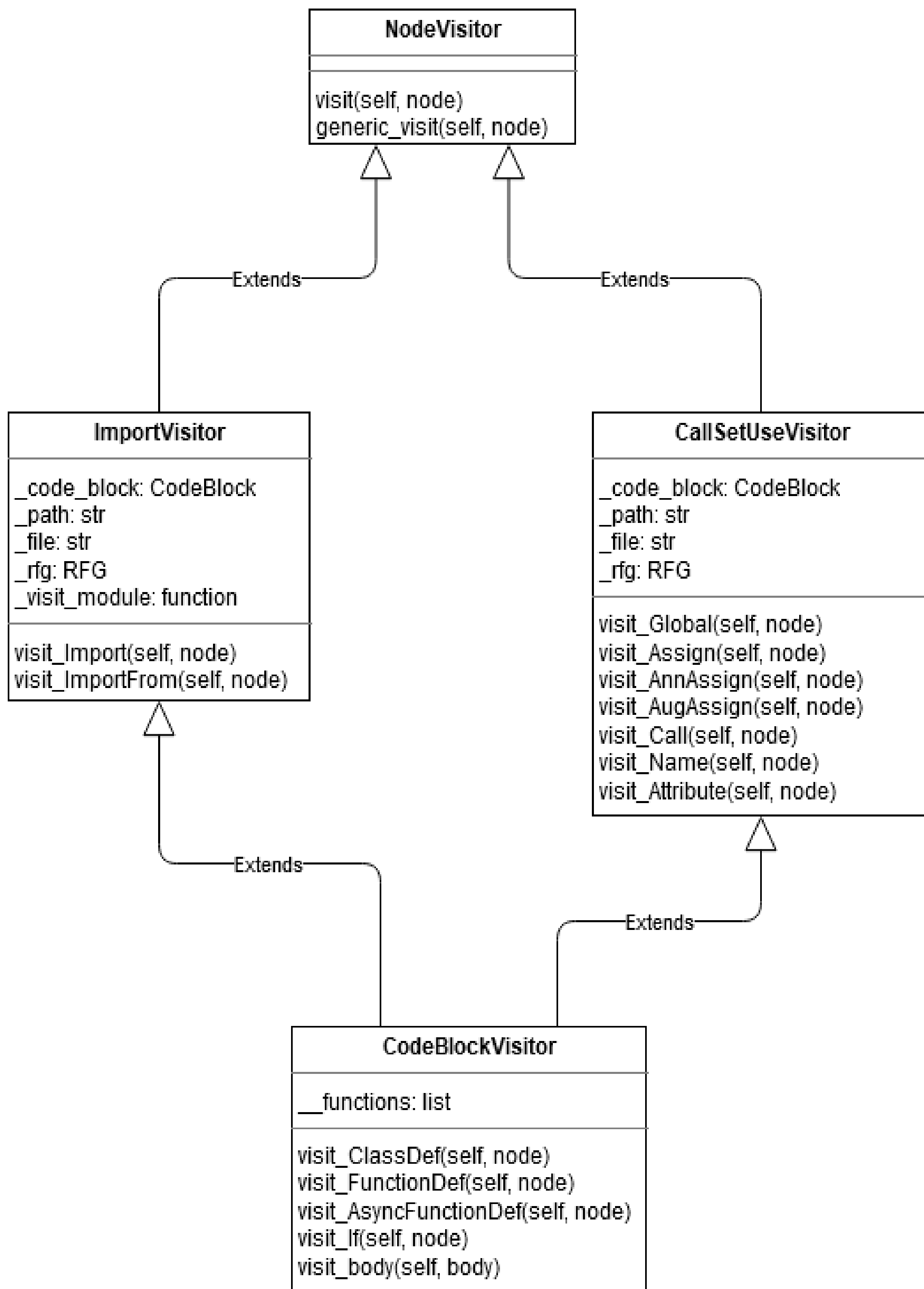


Abbildung 2: Visitors ohne private Methoden

5 Auswertung

5.1 Korrektheit

Um zu gewährleisten, dass die von *py2rfg* erzeugten Abhängigkeitsgraphen mit der Konzeption 3 übereinstimmen, habe ich eine Reihe von Tests implementiert. Eine umfangreiche Testabdeckung zu erzielen, war im zeitlichen Rahmen der Bachelorarbeit nicht möglich. Aus diesem Grund habe ich versucht, für jeden Knoten und Kantentyp minimale Tests zu schreiben, um wenigstens in Teilen alle Elemente abzudecken. Grundlegend besitzen die Tests den folgenden Aufbau:

1. Generierung des RFGs eines eigenständigen Python-Programms, das in den Test-Ressourcen liegt, durch *py2rfg*
2. Überprüfung, ob das zu testende RFG-Element mit allen Attributen im RFG existiert
3. Überprüfung des Schemas für jeden View des RFG

Neben diesen Tests existiert noch ein Test der einen RFG für *py2rfg* mit der Abhängigkeit zu Bauhaus erstellt. Dieser führt eine Überprüfung des Schemas durch und überprüft, ob alle *Linkage*-Namen eindeutig sind.
Die Überprüfung des Schemas erfolgt mithilfe des Skripts *dg_schema_checker.py*. Dieses stellte Axivion mir zur Verfügung, es gehört aber nicht zur Axivion Bauhaus Suite. Da es wahrscheinlich schon zur Prüfung der existierenden Schemata der Axivion Bauhaus Suite eingesetzt wurde, gehe ich davon aus, dass es korrekt funktioniert.

5.2 Analysen

Wie bereits in der Motivation 1.1 erwähnt, sollten die Analyse Werkzeuge der Axivion Bauhaus Suite, die auf dem RFG aufbauen, auf den generierten RFG anwendbar sein. Da der RFG bisher keine Metrikinformationen enthält, ist eine Metrikanalyse bisher nicht durchführbar. Es sollte jedoch in der Zukunft kein Problem darstellen, dem RFG Metrikinformationen hinzuzufügen, um eine Metrikanalyse anwendbar zu machen. Die Analyse von Zyklen, totem Code und Architektur lassen sich dabei erst einmal erfolgreich auf den RFG anwenden. Auf die Frage, von welchem Nutzen diese Analysen sind, gehe ich im folgenden für jede Analyse konkret ein.

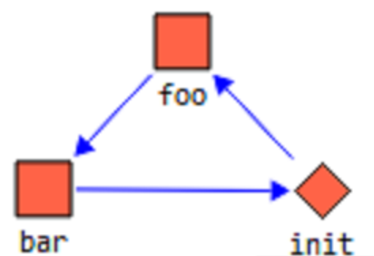
5.2.1 Zyklenanalyse

Als Grundlage der Zyklenanalyse dient die *Call*-View. Die Analyse kann somit nur Elemente umfassen, die Bestandteil der *Call*-View sind. Abgebildet in der *Call*-View sind

alle vorkommenden Funktionen, Methoden, Module, globale Variablen, Klassenvariablen und Instanzvariablen. Wieso Module, globale Variablen, Klassenvariablen und Instanzvariablen abgebildet sind, wurde bereits im Kapitel 3 *Konzeption eines RFG-Schema für Python* erläutert. Weiter werden in der *Call*-View alle Aufrufe von Bezeichnern abgebildet. Dabei wird eine *Call*-Kante zu allen RFG-Elementen erstellt, die mit diesem Bezeichner assoziiert werden. Der Aufruf einer Klasse, wird jedoch korrekterweise auf die Spezialmethoden *__new__()* und *__init__()* abgebildet.
Ein Problem des RFGs für die Zyklenanalyse ist, dass im *Call*-View Aufrufe innerhalb des Modulblocks nicht berücksichtigt werden. Diese Aufrufe werden nur beim ersten Importieren des Moduls ausgeführt. Mithilfe der Startknoten in der *Entry*-View könnte dieses Problem teilweise behoben werden, da so angenähert werden kann, welches Modul als erstes von einem anderen Modul importiert wird.
Ein anderes Problem der Zyklenanalyse ist, dass ein Aufruf einer Variablen immer eine Sackgasse darstellt. Das Problem besteht aufgrund der statischen Analyse des Programms und nicht der Implementierung des RFG. Eine statische Analyse kann nicht ermitteln, welches Objekt zum Zeitpunkt des Aufrufs in einer Variablen gespeichert ist.
Außerdem ist ein Problem, dass beim Aufruf eines Bezeichners alle RFG-Elemente aufgerufen werden, die mit diesem Bezeichner assoziiert werden. Dieses Problem liegt auch an der statischen Analyse des Programms, da nicht genau ermittelt werden kann, auf welches Objekt genau ein Bezeichner verweist, wie in Listing 5.2.1 dargestellt. Dennoch ist es für die Zyklenanalyse sinnvoller, alle RFG-Elemente aufzurufen, statt keinem. Da es einerseits besser ist, nicht vorhandene Zyklen darzustellen, als vorhandene Zyklen wegzulassen. Andererseits ist es ein schlechter Programmierstil, Bezeichnern Objekte mit unterschiedlicher Bedeutung zuzuweisen, da die Aussagekraft des Bezeichners verloren geht. Die zusätzlichen Zyklen zeigen dies auf.
Ein Beispiel zur Zyklenanalyse ist in Abbildung 3 zu sehen.

Abbildung 3: Beispiel einer Zyklenanalyse

```
1 def foo():
2     def bar():
3         Baz()
4     bar()
5
6
7 class Baz:
8     def __init__(self):
9         foo()
```



```
1 if boolean:
2         def bezeichner():
3                 pass
4 else:
5         bezeichner = "foo"
```

5.2.2 Tote Code-Analyse

Die tote Code-Analyse nimmt als Eingabe die *Call*- und *Entry*-View. Alles, was bereits in Abschnitt 5.2.1 Zyklenanalyse zur *Call*-View gesagt wurde, trifft auch auf die tote Code-Analyse zu. In der *Entry*-View sind die Startknoten der Analyse enthalten. Die Analyse wird durch meine Abschätzung der Startknoten in Abschnitt 3.5.5 beeinflusst.
In Abbildung 4 ist ein Ausschnitt der toten Code-Analyse von py2rfg dargestellt. Der Ausschnitt zeigt, dass für die Generierung des RFG die Standard-Bibliothek, Abhängigkeiten und das Programm analysiert werden und dass die Analyse für alle gleich verläuft. Der Knoten *find* / *visit* ist der Einstiegspunkt für die Analyse des Dateisystems / der Module.

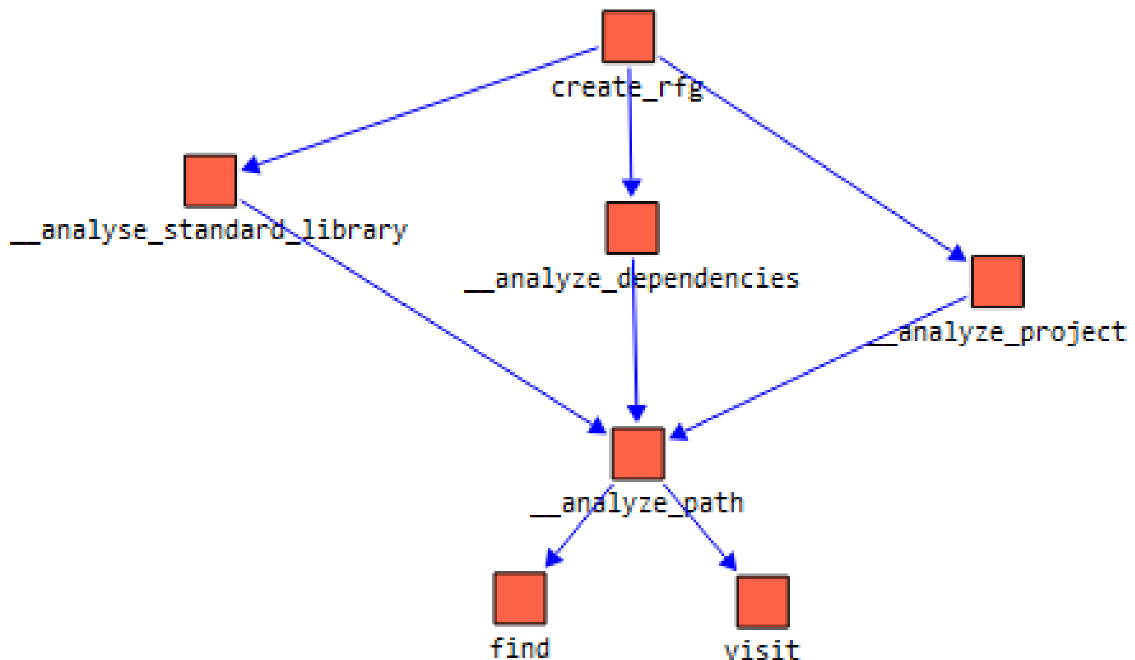


Abbildung 4: Ausschnitt der toten Code-Analyse von py2rfg

5.2.3 Architekturanalyse

Die *Code-Facts*-View dient als Eingabe für die Architekturanalyse. Da sie alle Elemente der *Call*-View enthält, treffen die in Abschnitt 5.2.1 Zyklenanalyse gesagten Punkte zur *Call*-View auch auf die Architekturanalyse zu. Das Problem des Aufrufs eines Bezeichners, der mehrere RFG-Elemente assoziiert, lässt sich auf das Nutzen von Bezeichnern ausweiten.
Abbildung 5 zeigt eine Architekturanalyse der Abhängigkeiten von py2rfg. Grüne Kanten stehen für eine Übereinstimmung der Soll- und Ist-Architektur, während rote Kanten für eine Abhängigkeit in der Ist-Architektur stehen, die nicht in der Soll-Architektur vorhanden ist.

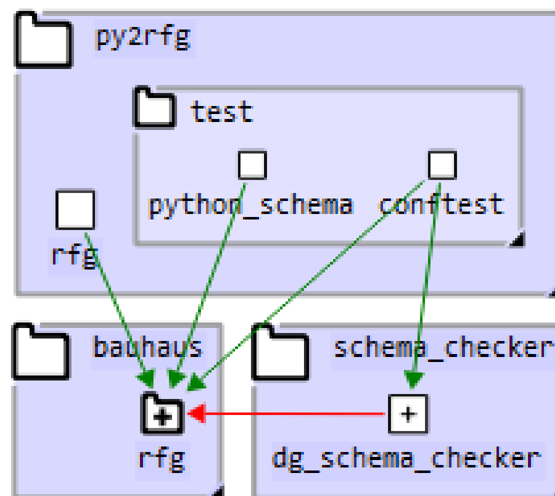


Abbildung 5: Architekturanalyse der Abhängigkeiten von py2rfg

6 Fazit

Als Ergebnis dieser Arbeit existiert das Programm py2rfg [py2], das für ein Python-Programm einen RFG generiert. Die Schritte von der Eingabe des Python-Programms bis zur Ausgabe des RFGs sind erläutert. Dabei werden besondere Implementierungsdetails vorgestellt. Des Weiteren existiert ein Schema für diesen RFG, das insbesondere auf die dynamischen Aspekte von Python eingeht. Außerdem ist ein Ergebnis dieser Arbeit, die aussagekräftige Durchführung von toten Code-, Zyklen- und Architekturanalysen auf dem generierten RFG, obwohl dieser viele dynamische Eigenschaften nicht erfasst. Als Letztes werden für bestehende Probleme der Analysen Lösungsansätze vorgestellt.

6.1 Ausblick

Diese Arbeit kann für die Entwicklung weiterer Abhängigkeitsgraphen für andere Sprachen, besonders dynamischen, hilfreich sein. Kurz vor Abgabe dieser Arbeit wurde eine ähnliche Arbeit [Sch20] abgegeben. Diese Arbeit befasst sich mit der Konzeption und Generierung eines RFG für JavaScript, das auch eine dynamische Programmiersprache ist. Aus zeitlichen Gründen war ein Eingehen auf diese Arbeit nicht möglich. Dennoch möchte ich sie hier als Quelle für die Entwicklung weiterer Abhängigkeitsgraphen erwähnen.

6.1.1 Erweiterung des RFG

In diesem Abschnitt möchte ich einige Ideen für die Erweiterung des RFG vorstellen. Eine davon ist die Erweiterung um Metrikinformationen, um Metrikanalysen auf dem RFG zu ermöglichen. Die Erweiterung um Typen wäre auch möglich, wie in Abschnitt 3.4 bereits angesprochen. Von Vorteil wäre zudem die Integration von eingebauten Bezeichnern, die im Module *builtins.py* liegen. Die Integration von häufig benutzten Bestandteilen der Standard Bibliothek, die nicht in Python geschrieben sind, wäre auch sinnvoll. Weiterhin könnten zum besseren Codeverständnis Kanten für das Überschreiben von Methoden hinzugefügt werden. Zum besseren Codeverständnis lässt sich der RFG auch um eine Menge von *Element*-Attributen erweitern. Beispiele hierfür sind das Setzen von *Element*-Attributen für Generatoren, geschlossenen Klassen und abstrakten Klassen, Methoden und Member. Zuletzt ließe sich noch das Konzept von Metaklassen und asynchronen Funktionen zur Erweiterung des RFG betrachten.

Abbildungsverzeichnis

1 Klassendiagramm Knotentypen 18
2 Visitors ohne private Methoden 22
3 Beispiel einer Zyklenanalyse 24
4 Ausschnitt der toten Code-Analyse von py2rfg 25
5 Architekturanalyse der Abhängigkeiten von py2rfg 26

Listings

1 Beispiel zu Duck-Typing 3
2 Attribut wird zur Laufzeit hinzufügt 3
3 Fehler durch ungebundene lokale Variable 5
4 *global*-Anweisung in *if-else* 5
5 Import liefert Objekt vom Typ function 12
6 Codebeispiel zur Erstellung eines *Class*-Knotens aus *_code_block_visitor.py* 17

Tabellenverzeichnis

1 Namensräume 4
2 Bindungen verschiedener import-Anweisungen (nach dem Beispiel in der Python-Dokumentation) 6
3 RFG-Attributtypen 7
4 Existierende RFG-Schemata 7
5 Bedeutung der Attributpräfixe 8
6 *Source*-Attribute 9
7 Knotentypen 10
8 Knotenobertypen 11

9 Kantentypen - Start- und Zielknoten 11
10 Method-Knotenattribute 12
11 Code-Facts-View - Enclosing-Kanten 13
12 Code-Facts-View - Kanten (ohne Enclosing) 14
13 Schnittstelle Klasse Node 19

Literatur

[ast] *Python 3.7 - ast*. https://docs.python.org/3.7/library/ast.html#abstract-grammar, . – Last accessed on 2020-02-13

[axia] *Axivion*. https://www.axivion.com/de, . – Last accessed on 2020-02-13

[axib] *Axivion Bauhaus Suite*. https://www.axivion.com/de/produkte-58#produkte_bauhaussuite, . – Last accessed on 2020-02-13

[Axic] AXIVION GMBH (Hrsg.): *Axivion Bauhaus Suite Documentation*. Axivion GmbH, Nobelstr. 15, 70569 Stuttgart, Germany: Axivion GmbH

[Bay13] BAYBULATOV, Emil: *Generierung eines globalen Abhängigkeitsgraphen für Java in Eclipse*, Universität Bremen, Diss., 2013

[bre] *Universität Bremen - Fachbereich 3 - Arbeitsgruppe Softwaretechnik*. https://www.informatik.uni-bremen.de/st/index.php, . – Last accessed on 2020-02-13

[Har06] HARDER, Jan: Generierung und Erweiterung des Resource Flow Graph für Visual Basic 6 Programme. (2006)

[Kos03] KOSKINEN, Jussi: Software maintenance costs. In: *Information Technology Research Institute, ELTIS-Project University of Jyväskylä* (2003), S. 16

[Kos08] KOSCHKE, Rainer: Zehn Jahre WSR–Zwölf Jahre Bauhaus. In: *Software archeology and the handbook of software architecture* (2008)

[Mül04] MÜLLER, Markus: *Konzeption und Generierung eines RFG für COBOL*. Universitätsbibliothek der Universität Stuttgart, 2004

[Nei04] NEINERT, Sascha: *Extraktion statischer Abhängigkeiten aus Ada95-Programmen mittels ASIS*, Verlag nicht ermittelbar, Diss., 2004

[pro] *Projekt Bauhaus*. https://www.informatik.uni-bremen.de/st/projektedetails.php?id=&projekt_id=20, . – Last accessed on 2020-02-13

[py2] *py2rfg*. https://git.hhu.de/flmag100/py2rfg, . – Last accessed on 2020-02-13

[pyt] *Python Reference Manual*. https://docs.python.org/3.7/reference/index.html, . – Last accessed on 2020-02-13

[Sal04] SALIB, Michael: *Starkiller: A static type inferencer and compiler for Python*, Massachusetts Institute of Technology, Diss., 2004

[Sch20] SCHWARZ, Kevin: *Ein JavaSript Abhängigkeitsgraph*, Universität Düsseldorf, Diss., 2020

[stu] *Universität Stuttgart - Institut für Softwaretechnologie - Abteilung Programmiersprachen und Übersetzerbau (bis 30.09.2019)*. https://www.iste.uni-stuttgart.de/de/ps/, . – Last accessed on 2020-02-13

[tio] *TIOBE Index*. https://www.tiobe.com/tiobe-index/, . – Last accessed on 2020-02-13