

Brandenburgische Technische Universität Cottbus-Senftenberg
Institut für Informatik
Fachgebiet Praktische Informatik/Softwaresystemtechnik

Masterarbeit



Brandenburgische
Technische Universität
Cottbus - Senftenberg

Integrationstesten von GraphQL mittels Prime-Path Überdeckung

Integration testing of GraphQL using Prime-Path Coverage

Tom Lorenz

MatrikelNr.: 3711679

Studiengang: Informatik M.Sc

Datum der Themenausgabe: 16.05.2023

Datum der Abgabe: (hier einfügen)

Betreuer: Prof. Dr. rer. nat. Leen Lambers

Gutachter: M.Sc Lucas Sakizoglou

Eidesstattliche Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Wörtlich und inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Die Arbeit ist nicht in gleicher oder vergleichbarer Form (auch nicht auszugsweise) im Rahmen einer anderen Prüfung bei einer anderen Hochschule vorgelegt oder publiziert worden. Der Verfasser erklärt sich zudem damit einverstanden, dass die Arbeit mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft wird.

.....
Ort, Datum

.....
Unterschrift

Inhaltsverzeichnis

1	Abstract	1
2	Einleitung	3
2.1	Motivation	3
2.2	Umsetzung	4
3	Grundlagen / Theorie	5
3.1	Graphentheorie	6
3.1.1	allgemeiner Graph	6
3.1.2	Grundbegriffe der Graphentheorie	6
3.1.3	Zusammenfassung	8
3.2	GraphQL	9
3.2.1	Schema & Typen	9
3.2.2	vordefinierte Typen	10
3.2.3	Resolver	12
3.3	Zusammenhang Graphentheorie und GraphQL	14
3.3.1	Typen als Knoten	14
3.3.2	Felder als Kanten	14
3.3.3	der komplette Graph	16
3.4	Testen	17
3.4.1	Arten von Tests	17
3.4.2	Test-Coverage	18
4	Graphcoverage	20
4.1	Graphcoverage allgemein	20
4.2	Graphcoverage Kriterien	21
4.2.1	Node Coverage	21
4.2.2	Edge-Coverage	21
4.2.3	Edge-Pair Coverage	22
4.2.4	Prime-Path Coverage	23
4.2.5	Complete-Path Coverage	24
4.2.6	abschließender Vergleich der Coverage-Kriterien	24
4.3	Graphcoverage für Code	24
4.4	Graphcoverage für GraphQL	24
5	related Work / verwandte Arbeiten	25
5.1	Property Based Testing	25
5.2	heuristisch suchenbasiertes Testen	26

5.3	Deviation Testing	26
5.4	Query Harvesting	27
5.5	Vergleich der Arbeiten	28
5.6	Andere Arbeiten	28
5.6.1	Empirical Study of GraphQL Schemas	28
5.6.2	LinGBM Performance Benchmark to Build GraphQL Servers	28
5.6.3	GraphQL A Systematic Mapping Study	29
6	Testentwurf	30
6.1	Schema in Graph abbilden	30
6.2	Pfade aus Graph bilden	36
6.3	Coverage-Pfade ermitteln	37
6.3.1	filternder Ansatz	37
6.3.2	generativer Ansatz	38
6.4	Query aus Pfad ermitteln	39
6.5	Test ausführen & Testauswertung	43
6.5.1	Positive Tests	43
6.5.2	Falsch-Negative Tests	46
6.5.3	Negative Tests	46
6.6	Zusammenfassung der Methode und struktureller Vergleich mit Property-based Testing	47
7	Praxis	49
7.1	Tool- / Dependencyauswahl	49
7.1.1	NetworkX	49
7.1.2	Faker	50
7.1.3	PyTest	51
7.2	Umsetzung der Methode	51
7.2.1	Schema in Graph abbilden	51
7.2.2	Pfade aus Graph bilden	53
7.2.3	Coveragepfade ermitteln (maybe remove - TODO)	54
7.2.4	Querys aus Pfad ermitteln	54
7.2.5	Test ausführen / Testfile generieren	54
7.2.6	Testauswertung	54
7.3	Zusammenfassung der Implementation	55
8	Auswertung / Experiment / Vergleich mit Property-based Testing	56
8.1	Vergleichsmetriken	56
8.1.1	Metriken aus Property-based Testing	56
8.1.2	Fehlerfindungskapazitäten	56
8.1.3	Neue Metrik	57
8.2	Threats to Validity / Limitierungen	58
8.2.1	Argumentgeneratoren	58

9 zukünftige Arbeit	39
10 Fazit	40
11 Glossar	62
12 Anhang	63
Literaturverzeichnis	72

1 Abstract

Im Zuge der digitalen Transformation nimmt die Anzahl von Softwareanwendungen rasant zu und insbesondere durch das Internet of Things und die generelle fortschreitende Vernetzung diverser Geräte nimmt in diesem Maße auch die Netzwerklast zu. Bisheriger Standard für Kommunikation von Geräten über das Internet waren REST-APIs diese haben jedoch gewisse Limitierungen wie zum Beispiel: Ineffizienz durch Overfetching/Underfetching, Anzahl an Requests, Versionierung und Komplexität uvm. Mit der Veröffentlichung von GraphQL in 2015 wurde ein Konkurrent zu REST in das Leben gerufen der diese Probleme beheben kann. Durch GraphQL lässt sich insbesondere die Netzwerklast reduzieren da eine GraphQL-Request, im Gegensatz zu REST, mehrere Anfragen in einer einzigen HTTP-Request zusammenfassen kann und dabei auch nur die wirklich gewünschten Daten überträgt. Dadurch, dass jedoch die wachsende Anzahl von Softwareanwendungen auch in immer kritischere Bereiche des Lebens vordringt, ist es enorm wichtig die Qualität der Software sicherzustellen. Eine Methodik zum Sicherstellen der korrekten Funktionalität von Software ist das Testen von Software im Sinne von Validierungstests die sicherstellen sollen, dass die Software vorher definierte Szenarien nach Erwartung behandelt. Für REST-APIs existieren zahlreiche Tools die solche Validierungstests automatisch übernehmen können wohingegen es noch einen Mangel an Tools dieser Art für GraphQL gibt. Im Rahmen der IEEE/ACM 2021 wurde ein Paper veröffentlicht, dass eine Methode vorstellt wie GraphQL-APIs mithilfe von Property-based Tests automatisch getestet werden können. Property-based bezieht sich darauf, dass die Eigenschaften eines Objektes genutzt werden um diese zu testen. Diese Methode generiert zufällig Tests und bietet so eine Möglichkeit, Fehler zu entdecken und generell erstmal eine Grundlage an Tests zu schaffen frei nach dem Motto: lieber irgendwelche Tests als gar keine Tests". Die zufallsbasierte Testgenerierung weist allerdings einige Schwachstellen auf. So kann Sie nicht garantieren, dass die API zu jeder Zeit eine gute Coverage hat denn es können einzelne Routen der API komplett ausgelassen werden. Es sind Testszenarios denkbar, die sehr viele false-positives durchlassen & somit die Qualität der Software nicht ausreichend sicherstellen können. GraphQL ermöglicht außerdem einen potentiell unendlichen Suchraum für die Tests. Um diesem Problem zu entgehen wurde ein einfaches rekursions-limit definiert was jedoch dazu führt, dass die Testabdeckung nur bis zu einem bestimmten Grad überhaupt reichen kann. Längere Pfade die mit GraphQL definiert werden können, werden deshalb niemals getestet.

Mit dieser Arbeit soll ein anderer Ansatz für die automatisierte Testgenerierung untersucht werden. Hierbei soll untersucht werden, inwiefern Graphcoverage auf GraphQL angewendet werden kann zur Generierung von Tests. Mithilfe von einem allgemeinen Graphalgorithmus, der sonst zur Kontrollflussgraphen Analyse benutzt wird, wollen wir zeigen, dass es möglich ist mit einem iterativen Verfahren GraphQL zu testen. Wir

versprechen uns von den Graphcoverage-Algorithmen einer verlässliche und sichere Generierung von Tests die einerseits effizient jegliche Methoden abdecken und andererseits die Anzahl der Tests minimal hält. Um die Methodik zu validieren werden am Ende beide Tools einem Experiment unterzogen um zu sehen ob die hier vorgestellte Methode eine Verbesserung erzielen kann.

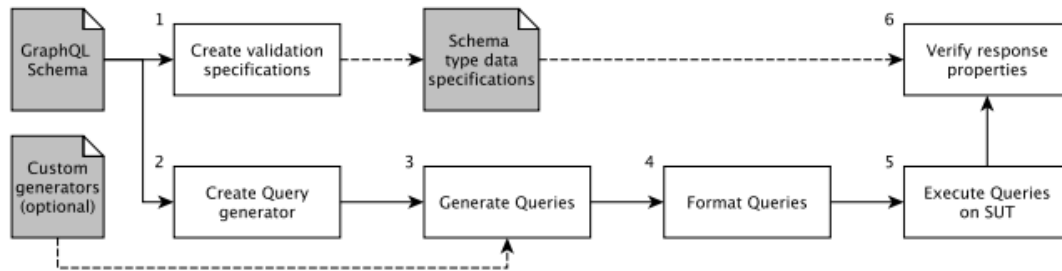
2 Einleitung

In diesem Kapitel wird an das Thema und die Motivation dieser Arbeit herangeführt. Außerdem wird definiert, welche Ziele diese Arbeit erreichen soll und eine grobe Übersicht über die Kapitelstruktur gegeben.

2.1 Motivation

Mit einer steigenden Nutzung von GraphQL wird es immer wichtiger, geeignete Tests für GraphQL-API's zu entwickeln damit eine gute Softwarequalität sichergestellt werden kann. Tests können manuell geschrieben oder automatisch generiert werden. Während bei Unit-Tests für einzelne Methoden der Programmierer frei entscheiden kann, ob er die Tests selbst schreiben will oder doch eher von einem Tool generieren lassen will sind bei Integrations-Tests die Testräume teilweise so groß, dass ein manuelles Schreiben dieser Tests einerseits fehleranfällig aber auch schlicht zu langwierig ist. Für REST-APIs existieren schon automatische Integrationstesttools wie zum Beispiel: EvoMaster [1], Quick-REST [2] oder RESTTESTGEN [3]. GraphQL-APIs haben leider noch einen Mangel an solchen automatischen Testtools. Im Rahmen der IEEE/ACM 2021 wurde mit "Automatic Property-based Testing of GraphQL APIs" [4] eine Methode vorgestellt die diesen Mangel angehen soll. Ergebnis der Arbeit war hierbei ein Prototyp der die vorgestellte Methode umsetzen sollte. Die vorgestellte Methode bezieht sich auf "Property-based Testing" wobei diese gleichzusetzen ist mit "Random Testing" [4][vgl. 2.B]. Durch die starke Typisierung und Schema-Definition, welche prinzipiell ein Graph ist, lässt sich in GraphQL sehr einfach auswerten welche Anfragen nun zulässig sind. Die hier vorgestellte Methode nutzt den Fakt, dass zum Beispiel alle möglichen Anfragen immer im Query-Knoten beginnen müssen und somit alle weitergehenden Felder im Schema innerhalb des Query-Knoten definiert sein müssen. Da die Definition des Schemas der eines Graphens entspricht ist jedoch der mögliche Suchraum potentiell unendlich da GraphQL Zyklen innerhalb des Graphens erlaubt. Der unendliche Suchraum wurde durch ein Rekursionslimit begrenzt allerdings wird so eine schlechtere Test-Coverage erreicht.

Die bisher entwickelte Methode funktioniert auf folgende Weise:



Methode von [4]

wobei neben dem vorher erwähnten Rekursionslimit außerdem Punkt 6. "Verify response Properties" kritisch zu betrachten ist da die Auswertung wirklich nur auf die Properties schaut. Dies bedeutet, dass ein zurückgegebens Objekt nur auf seinen Typ überprüft wird aber nicht, ob seine tatsächlichen Rückgabewerte, die exakt erwartet sind. Hierdurch können false-positives entstehen.

Die vorgestellte Methode wollen wir verbessern durch Änderung einiger Ansätze. Hierbei sollen Graphcoverage-Algorithmen zum Einsatz kommen, die mit iterativen Verfahren auch zyklische Graphen ideal überdecken können und dabei immer verlässlich arbeiten, sodass die generierten Tests stets für vergleichbare Ergebnisse sorgen und nicht davon abhängig sind, dass der Zufall eine gute Überdeckung liefert. Zusammenfassend sei gesagt, dass bei der Testgenerierung und Testauswertung Verbesserungen möglich sind und dies Gegenstand dieser Arbeit sein soll.

2.2 Umsetzung

Zuallererst wird in dieser Arbeit etwas Theorie definiert und in Bezug gesetzt. Wir beginnen damit die Graphentheorie als mathematisches Konzept zu definieren denn dieses liegt GraphQL zugrunde. Daraufgehend kommt eine kleine Einführung in GraphQL und dann bilden wir schon die Schnittstelle von GraphQL zur Graphentheorie. Sobald diese Verbindung erfolgt ist können wir uns dem eigentlichen Problem widmen: Wie kann man mithilfe von Graphcoverage-Algorithmen Tests generieren? Hierfür erfolgt ein letzter Theorie-Exkurs über Software-Tests. Mit diesen Grundlagen schaffen wir es dann unsere Methode zu entwickeln und können Sie auch mit "Automatic Property-based Testing of GraphQL APIs" [4] vergleichen. Unsere Methode wird konzeptionell vorgestellt und dann folgen einige Implementierungsdetails sowie ein Vergleich beider Tools durch Experimente. Abschließen wird die Arbeit mit einem Ausblick für zukünftige Arbeit und einem Fazit über unsere erreichten Verbesserungen.

3 Grundlagen / Theorie

Das automatisierte Testen von GraphQL-APIs erfordert ein spezifisches Domänenwissen in verschiedenen Teilbereichen der Informatik und Mathematik. Dieses Domänenwissen wird in den folgenden Abschnitten aufgrund Lage zweier Lehrbücher erarbeitet und in Kontext gesetzt. Wissen über die Graphentheorie wird benötigt, da GraphQL eine Implementierung von graphenähnlichen Strukturen ist und wir somit Algorithmen darauf anwenden wollen beziehungsweise können. Die mathematische Formalisierung hilft hierbei dann insbesondere bei der Beweisführung für eine allgemeine Termination der zu entwickelnden Algorithmen. Desweiteren ist es nötig sich bewusst zu machen, welche Arten des Testens von Software es gibt und wann man "genug" oder "zu wenig" getestet hat. Hierfür gibt es Metriken die auch insbesondere auf Graphen angewendet werden können und uns dann verraten können ob unsere Tests ausreichend sind. Im konkreten ist das Theorie-Kapitel so strukturiert, dass erst einmal die mathematischen Grundlagen der Graphentheorie vermittelt werden und im Anschluss dazu wird eine Beziehung zwischen GraphQL & Graphentheorie hergestellt. Mit der Beziehung können wir dann zeigen, dass Graphalgorithmen auch bei GraphQL anwendbar sind. Darauffolgend zeigen wir verschiedene Kriterien die eine Testüberdeckung ermöglichen würden und erklären die Unterschiede.

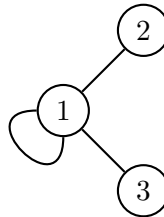
3.1 Graphentheorie

Da GraphQL es ermöglicht, dass komplexe Beziehungen innerhalb eines Datenmodells in Form von Graphen modelliert werden, benötigen wir die Graphentheorie, da diese Methoden liefert um Graphen systematisch zu definieren und analysieren. Desweiteren sind Teile der Testcoverage-Kriterien eng mit Graphenstrukturen verbunden sodass wir gezwungen sind uns einführend mit der Graphentheorie zu beschäftigen damit wir die Grundlagen erarbeiten, die wir später benötigen werden. Generell wird es im folgenden eher etwas theoretischer gehalten, die Zusammenhänge werden sich in späteren Kapiteln erschließen wenn wir Graphen mit GraphQL verbunden oder Tests aus Graphenstrukturen entwickeln.

3.1.1 allgemeiner Graph

Ein Graph ist ein Paar $G = (V, E)$ zweier disjunkter Mengen mit $E \subseteq V^2$ [5, vgl. S.20.1 Graphen]

Elemente von V nennt man Knoten eines Graphens, die Elemente von E nennt man Kanten, Knoten die in einem Tupel von E vorkommen nennt man auch inzident (benachbart) [5, vgl. S.3 0.1 Graphen]. Einen Graphen kann man nun definieren, indem wir zum Beispiel für $V = 1, 2, 3$ wählen und für $E = (1, 1), (1, 2), (1, 3)$. Dargestellt werden können Graphen, indem man die Elemente von V als, zum Beispiel, Kreis zeichnet und dann alle Kanten aus E einzeichnet, indem man die Punkte verbindet [5, vgl. S.2 0.1 Graphen]. Eben definierter Graph hat dann folgende Darstellung:



Es finden sich auch andere Darstellungsformen für Graphen, hier sei insbesondere die Adjazenzmatrix genannt. Für unseren Anwendungsfall belassen wir es jedoch bei Punkten und Linien. Mit dieser Definition lassen sich nun beliebig große, ungerichtete Graphen erstellen.

3.1.2 Grundbegriffe der Graphentheorie

Verschiedene Begriffe sind grundlegend für ein weiterführendes Verständnis der Graphentheorie. Wir werden hier auf einige dieser Grundbegriffe näher eingehen, insbesondere auf diejenigen die im Testkontext unbedingt benötigt werden.

Kante

Die Kante wurde zwar zuvor schon definiert als $e \in E \subseteq V^2$ also ein Tupel (x, y) wobei $x, y \in V$ mit E und V wie in 4.1.1 definiert. Hierbei zeigt die Kante, dass die Knoten

x und y miteinander verbunden sind. Äquivalent ist auch die Aussage, dass y und x verbunden sind. Wir nennen diese Kanten auch ungerichtete Kanten [5, vgl. S.3 0.1 Graphen] Wir wollen nun zwei weitere Arten von Kanten definieren.

gewichtete Kante

Eine gewichtete Kante fügt einen dritten Parameter zu einer Kante hinzu, sein Gewicht. Dies ist ein dritter Parameter einer Kante (x, y) wir erweitern also die Definition um einen dritten Parameter (x, y, z) wobei z meist $z \in \mathbb{R}$ ist. Solche gewichte können benutzt werden um ideale Wege in Graphen zu finden da sie einer Kante eine Zahl zuweisen, die als Kosten genutzt werden können. Gewichtete Kanten werden in unserem Prototypen eine wichtige Rolle spielen allerdings wird bei uns die Kante keine Zahl aus \mathbb{R} als Gewicht bekommen sondern Objekte aus GraphQL.

gerichtete Kante / gerichteter Graph

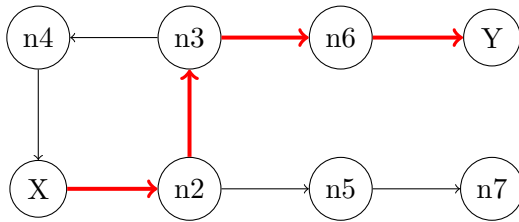
Eine gerichtete Kante besitzt im Gegensatz zu den ungerichteten Kanten eine Orientierung. Dies bedeutet, dass die Kante (x, y) einen Startknoten und einen Endknoten definieren muss. Gekennzeichnet wird dies für den Startknoten durch $init()$ und für den Endknoten durch $end()$. Eine gerichtete Kante vom Knoten x nach y wird somit definiert als $(init(x), end(y))$ [5, vgl. S.26 Verwandte Begriffsbildungen] Die Einführung von gerichteten Kanten ermöglicht es uns, gerichtete Graphen zu erstellen. Gerichtete Graphen sind Graphen, die nur gerichtete Kanten besitzen. Im Prinzip funktionieren gerichtete Kanten wie Einbahnstraßen - es ist nur möglich in die erlaubte Richtung vorzuschreiten. Die gerichteten Graphen sind besonders wichtig für uns, da alle späteren Analysen von GraphQL auf gerichteten Graphen basieren.

Grad eines Knoten

Der Grad eines Knoten gibt an, wie viele andere Knoten mit diesem Knoten verbunden sind und wird als $d_G(node)$ benannt. In ungerichteten Graphen ist der Grad eines Knoten x mit 4 Kanten also $d_G(x) = 4$. Bei gerichteten Graphen unterscheiden wir zwischen Eingangs- und Ausgangsgrad. Der Eingangsgrad ist definiert als Anzahl aller Kanten die auf den Knoten eingehen und der Ausgangsgrad ist genau entgegengesetzt, die Anzahl aller Kanten die ausgehend vom gewählten Knoten sind.

Pfad / Weg

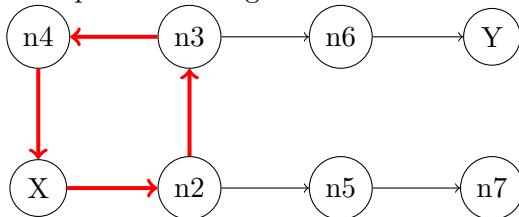
Ein Pfad, oft auch Weg genannt, ist eine Sequenz von Knoten die nacheinander durch Kanten miteinander verbunden sind. [5, vgl. S. 7 0.3] Der Pfad von einem Knoten x zu einem Knoten y über Knoten n_1 bis n_7 hat dann zum Beispiel diese Struktur $p = \{(x, n_2), (n_2, n_3), (n_3, n_6), (n_6, y)\}$ wobei hierbei gilt, dass die Kanten $\{(x, n_2), (n_2, n_3), (n_3, n_6), (n_6, y)\} \in E$. Graphisch würde dies wie folgt aussehen (der Pfad in Rot markiert):



Pfade haben immer eine Länge. Die Länge des Pfades entspricht der Anzahl der Kanten innerhalb dieses Pfades, das bedeutet, dass $\text{Pfadlänge} = |p|$ wobei $p = \{(x, n2), (n2, n3), (n3, n6), (n6, y)\}$. Zu beachten ist, dass Pfade in gerichteten Graphen immer die Orientierung der Kanten beachten müssen. In obigem, gerichteten Pfad wäre also ein Weg $\{(X, N4), (N4, N3)\}$ nicht zulässig.

Zyklus / Kreis

Ein Zyklus, auch Kreis genannt, ist ein Pfad bei dem gilt, dass der Startpunkt und Endpunkt der Pfadsequenz identisch ist. Formell bedeutet dies, dass ein Pfad eine Sequenz $p = \{(X, e1), (e2, e3), \dots, (e_n, X)\}$ ist wobei es auch möglich ist, dass der Kreis Länge 1 hat also die Sequenz $p = \{(X, X)\}$ ist. Zyklen können in gerichteten und ungerichteten Graphen auftreten wobei das Vorkommen in ungerichteten Graphen eigentlich garantiert ist, insofern eine Kante existent ist. Das Auftreten in ungerichteten Graphen ist nicht garantiert aber kann durchaus geschehen. Der Graph aus vorigem Beispiel definiert exemplarisch den Pfad $p = \{(X, n2), (n2, n3), (n3, n4), (n4, X)\}$ welcher ein Kreis ist. Graphisch hervorgehoben:



3.1.3 Zusammenfassung

Wir haben nun eine kleine mathematische Einführung in das Gebiet der Graphentheorie hinter uns. Mithilfe der hier erarbeiteten Begriffe und Definitionen werden wir im folgenden verehrt arbeiten. Insbesondere wenn wir die Coverage-Algorithmen im Bezug des Testens einführen benötigen wir das hier erarbeitete Wissen.

3.2 GraphQL

GraphQL ist eine Open-Source Query-Language (Abfragesprache) und Laufzeitumgebung die von Facebook entwickelt wurde. [6, vgl. Introduction] Die Besonderheiten von GraphQL sind, dass man mit nur einer einzelnen Anfrage mehrere Ressourcen gleichzeitig abfragen kann [7, vgl. No More Over- and Underfetching] und die Daten in einem Schema durch einen Typgraphen definiert sind [7, vgl. Benefits of a Schema Type System]. So lässt sich die Effizienz stark erhöhen, indem weniger Anfragen gestellt werden die zeitgleich eine höhere Informationsdichte haben. Außerdem erleichtert GraphQL die Kommunikation von Schnittstellen, indem die gewünschten Felder schon in der Query definiert werden und direkt den erwarteten Datentyp zusichern. Hier liegt auch der große Vorteil im Vergleich zum direkten technologischen Konkurrenten REST API. Bei REST-APIs sind nämlich für verschiedene Ressourcen auch jeweils eine eigene Anfrage nötig und die Typsicherheit ist nicht so stark gegeben wie bei GraphQL-APIs. [7, vgl. No More Over- and Underfetching] Diese beiden großen Vorteile sorgen dafür, dass GraphQL an Popularität gewinnt und zunehmend eingesetzt wird. Im Kontext dieser Arbeit ist ein tiefgreifendes, technologisches Verständniss von GraphQL essenziell, deshalb wird hier eine tiefgreifende Erklärung von GraphQL folgen.

3.2.1 Schema & Typen

Grundlage einer jeden GraphQL-API ist ein GraphQL-Schema. [6, vgl. Core Concepts] Dieses Schema definiert genau wie die Daten der API aufgebaut sind und welche Informationen existieren. Ein GraphQL-Schema ist eine Sammlung von Typen. Typen sind Objekte einer Datenstruktur. Ein Typ definiert alle Informationen über sich, hierbei wird für jede Information ein Feld angelegt. Das Feld kann entweder ein Standarddatentyp wie String, Integer etc. sein oder ein anderer Typ. Falls das Feld ein anderer Typ ist, so entspricht diese Beziehung einer Kante in einem Graphen. Dies bedeutet, dass eine Abfrage dieses Feldes dann ein Objekt des Types zurückliefert und hier auch die gewünschten Felder definiert werden müssen. Ein sehr einfaches Schema wäre zum Beispiel die Beziehung zwischen Büchern und Autoren. Ein Buch hat einen Titel und einen Author. Ein Author hat einen Namen und ein Geburtsdatum. Zugehöriges Schema für dieses Beispiel sähe wie folgt aus:

```
type Buch {  
  title: String  
  author: Author  
}  
type Author{  
  name: String  
  geburtsdatum: Date  
}
```

Es lässt sich also festhalten, dass ein GraphQL-Typ immer als ein Tupel (Name, Felder)

definiert wird wobei die Felder eine Liste an Tupeln (**Feldname**, **Feldtyp**, **Datentyp**) sind. Hierbei gelten folgende Einschränkungen für die Elemente des Tupels:

Feldname ein eindeutiger Feldbezeichner

Feldtyp gibt Einschränkungen vor, z.B. nicht Null (durch **!**), Listentyp (durch **[]**) etc.

Datentyp der explizite Typ den das Feld hat, kann Standarddatentyp oder anders definierter Type sein

Wenn ein Typ ein Feld enthält, das kein Standarddatentyp ist so entspricht dieses Feld einer Kante in einem Graphen. Dieses Feld muss dann in einer Query näher definiert werden indem angegeben wird, welche Felder nun vom Typ zu dem die Kante führt, ausgegeben werden sollen.

3.2.2 vordefinierte Typen

Jedes GraphQL-Schema definiert initial mehrere Typen die spezielle Aufgaben haben und nicht vom User überschrieben werden können. Hierunter zählen unter anderem auch Standarddatentypen wie Sie gemeinhin bekannt sind. Im folgenden werden wir auf einige wichtige dieser Typen eingehen.

Scalar-Types

Grundlegende Datentypen (Standarddatentypen) werden durch Scalar Types ausgedrückt. Scalar-Types repräsentieren einzelne Werte wie z.B. einen Integer, einen String, Boolean Werte oder auch Datumstypen. Ein Scalar-Type kann nicht vom User geändert werden und enthält auch keine anderen Typen (im Gegensatz zu Objekttypen die das können). Es ist möglich auch eigene Scalar-Types festzulegen jedoch sind in der offiziellen Spezifikation von GraphQL lediglich folgende Scalar-Types definiert:

Typ	Beschreibung
Int	32-bit Integer
Float	Gleitkommazahl nach IEEE 754
String	frei wählbarer Text (leerer String zählt nicht als non-Null!)
Boolean	einzigartiger Identifier, intern behandelt wie ein String
ID	Rohkonstrukt von dem geerbt werden kann für eigen definierte Typen
Scalar Extensions	Rohkonstrukt von dem geerbt werden kann für eigen definierte Typen

[6, vgl. 3.5 Scalars]

Query-Type

Der Query-Type definiert alle erlaubten Anfragen (Leseoperationen) an die GraphQL-API. Hierbei können Anfragen mit und ohne Eingabeparameter angegeben werden. Die definierten Anfragen haben, wie jeder Typ, einen eindeutigen Bezeichner, welcher dann auch in der zustellenden Query benutzt wird. Die Felder der Antwort hängen hierbei vom Feldtypen ab. Ist das Ergebnis der Query-Definition ein Standarddatentyp, so wird es direkt ausgegeben. Ist es ein anderer definierter Typ, so muss näher bestimmt werden welche Felder erwartet werden. Nutzen wir das Beispiel der Bücher Autoren weiter, könnte man wie folgt einen Query-Type definieren:

```
type Query{
  # returns a List of all Books
  getBooks: [Book]
  # returns one random Book
  getBook: Book
  # returns the Author from a Book Title
  getBookByTitle(String title): Author
}
```

Mutation

Der Mutation-Type ist das Pendant des Query-Types. Im Mutation-Type werden alle erlaubten Schreiboperationen definiert. Dies beinhaltet das Erstellen, Aktualisieren und Löschen von Daten in der GraphQL-API. Felder im Mutation-Type haben auch immer einen Rückgabewert, Konvention ist es hierbei, die veränderten Objekte zurückzugeben, sodass der Client als Validierung exakt das Objekt bekommt, welches er sich "gewünscht hat". Die Operationen die mittels Mutation hervorgerufen werden, werden linear abgearbeitet. So ist es in GraphQL möglich, die Operationen miteinander zu verknüpfen. Man stelle sich z.B. vor, dass ein User seine Mail ändern will und gleichzeitig noch seinen Namen. Mit REST wären hier zwei Anfragen nötig, GraphQL kann dies mit einer Anfrage erledigen. Kehren wir wieder auf unser bisheriges Beispiel zurück, ein Mutationstyp hierfür könnte wie folgt aussehen:

```
type Mutation{
  # erstelle einen Author, return den erstellten Author
  createAuthor(name: String!, birthdate: Date): Author
  # erstelle ein Buch, return das erstellte Buch
  createBook(title: String!, author: Author): Book
  # ändere den Titel eines Buches, return des Buches mit geänderten Titel
  changeBookTitle(title: String! , newTitle: String!): Book
  # ändere den Geburtstag eines Autors, return des geänderten Autors
  changeBirthdateFromAuthor(author: Author!, newBirthDate: Date!)
}
```


eine einzelne Mutation kann Pflichtfelder und optionale Felder markieren. Pflichtfelder werden mit einem ! markiert. Diese Felder müssen dann als Argument bei jeder Ausführung mitgesendet werden. Optionale Felder benötigen dies nicht.

Subscription

Eine Besonderheit von GraphQL ist der Subscription-Type. Dieser ermöglicht eine Echtzeitkommunikation zwischen Server und Client, indem mithilfe des WebSocket-Protokolls eine permanente Verbindung zwischen Server und Client hergestellt wird. Diese permanente Verbindung ermöglicht es dem Server direkt Daten an den Client zu senden ohne, dass der Client dafür eine Anfrage schicken muss. Wie auch in allen anderen Typen definiert der Subscription-Type Felder mit einem Namen und Rückgabebetyp. Im Beispiel der Bücher Autoren wäre hierbei nun denkbar, dass eine Subscription für neue Bücher neue Autoren wünschenswert wäre. Hierdurch folgt dann dieser Subscription-Type:

```
type Subscription{
  # neues Buch veröffentlicht
  newBook: Book
  # ein neuer Author erscheint
  newAuthor: Author
}
```

Ein Client könnte sich auf diese Kanäle nun subscriben. Angenommen, wir wollen immer über ein neues Buch informiert werden so muss der Client diese Anfrage stellen:

```
subscription{
  newBook{
    title
    author{
      name
    }
  }
}
```

Hierdurch wird eine permanente Verbindung hergestellt und immer ein Objekt vom Type Book mit den Feldern title & author gesendet, wenn ein neues erschienen ist. Es ist wie auch in anderen Typen möglich, Felder wegzulassen falls diese nicht. Ein spezieller Client muss gewählt werden, der den WebSocket offen lässt, um den die Nachrichten zu empfangen, dies ist jedoch spezifisch abhängig von dem Projekt (Sprache, gewählte GraphQL Server Instanz).

3.2.3 Resolver

Bisher beschäftigen wir uns vorrangig mit der Strukturierung und Typisierung von GraphQL und den Daten die durch das GraphQL Schema dargestellt werden. Ein wichtiger

Baustein fehlt aber noch. Woher kommen die Daten? Wie werden Eingabedaten behandelt? Diese Fragen werden durch die Resolver beantwortet. Ein Resolver ist in GraphQL eine Funktion die zuständig für die Datenabfragen und Strukturierung ist. Im Schema haben wir bisher definiert in welcher Art und Weise wir die Daten haben wollen, der Resolver ist nun dafür zuständig, diese Daten im definierten Format zur Verfügung zu stellen. Die Resolver sind nicht, wie alle vorher benannten Teil von GraphQL offen einsehbar, sondern sind Funktionen einer Programmiersprache. Für jedes Feld im Schema, das Daten enthält, muss ein Resolver implementiert werden, dies umfasst insbesondere die Query, Mutation und Subscription Typen aber auch alle selbstdefinierten Typen. Die konkrete Implementierung der Resolver hängt von verschiedenen Dingen ab, insbesondere jedoch welchen GraphQL-Server man nutzt und welche Programmiersprache verwandt wird. Ein beispielhafter Resolver für die Query eines Buches anhand seines Titels mit ApolloServer in Javascript könnte folgende minimale Syntax haben:

```
const resolvers = {
  Query: {
    book: (parent, args, context, info) => {
      return getBookByID(args.id);
    },
  },
};
```

Wobei hierbei zu beachten sei, dass alle Argumente die mitgegeben werden im `args` Argument gespeichert sind. Die Funktion `getBookByID` gibt ein, dem Schema entsprechendes, `Json`-Objekt zurück. Da die Resolver konkrete Implementierungen außerhalb von GraphQL sind und die einzelnen Resolver untereinander aufrufen können, bedarf es hier einer Reihe an Tests da dieser Code gerne Fehlerhaft sein kann. Ein Klassiker für GraphQL ist es, einzelne Attribute in einem Resolver zu vergessen. Ein Entwickler sollte für jeden Resolver, der ja eine Funktion darstellt, Unit-Tests zur Verfügung stellen. Da sich die einzelnen Resolver aber auch untereinander aufrufen können, muss hier auch die Integrität getestet werden. Hier setzten wir an, indem unsere Testgenerierung darauf aufbaut, jeden möglichen Resolver in mindestens einer Kombination mit anderen Resolvieren zu testen. So kann sichergestellt werden, dass die Resolver untereinander integer sind.

3.3 Zusammenhang Graphentheorie und GraphQL

Nun, da wir ein grundlegendes Wissen über Graphentheorie und GraphQL erlangt haben, müssen wir noch zeigen, dass Graphentheorie auch anwendbar ist auf GraphQL. Ohne diese Verknüpfung könnten wir uns nicht sicher sein, dass die zukünftig vorgestellten Algorithmen ausführbar sind für eine GraphQL-API.

3.3.1 Typen als Knoten

Wie in 4.2.1 schon gezeigt, ermöglicht GraphQL das Erstellen von eigenen Typen. Diese Typen repräsentieren einzelne Datenobjekte. Somit liegt es nahe, dass die einzelnen Typen als Knoten eines Graphens repräsentiert werden können. Bleiben wir bei vorigem Beispiel, so definieren wir einen Typen *Book*:

```
type Book {  
  id: Int  
  title: String  
}
```

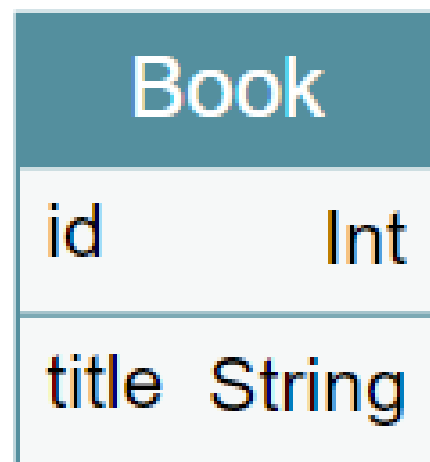
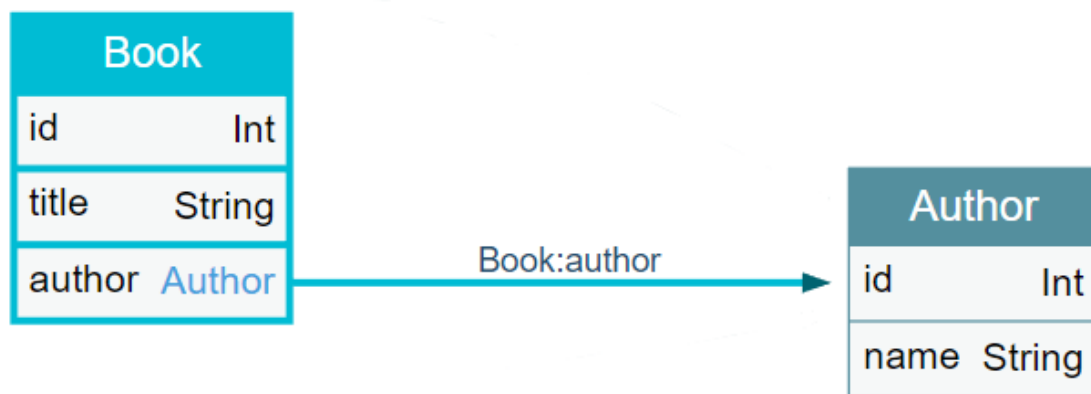


Abbildung 3.1: Book als Knoten

3.3.2 Felder als Kanten

Der eben definierte Typ *Book* besitzt aktuell nur zwei Felder, *id* und *title*. Beides sind Skalare-Datentypen und bilden somit keine ausgehenden Kanten, denn Kanten symbolisieren Beziehungen zwischen Objekten. Wir können einen Graphen mit $G = (V, E)$ wobei $E = \emptyset$ definieren. Allerdings wäre dies sehr restriktiv da wir überhaupt keine Verbindungen haben würden und viele Vorteile von GraphQL würden wegfallen. Es ist aber auch möglich, dass ein Type ein Feld definiert das nicht vom Skalaren Datentyp ist. So können wir das Buch um einen Typen *Author* erweitern wobei der Author gleich definiert wird wie in 4.2.1. Diese Typdefinitionen führen dann zu diesem Graphen:



Wir nutzen also die Typdefinitionen aus um die Beziehungen zwischen einzelnen Objekten zu bekommen. Hierbei wird ein Typ dann mit einem anderen Typen verbunden, wenn dieser als Feld im Ursprungstyp vorkommt. Man muss hierbei klar beachten, dass dies eine gerichtete Beziehung ist. Eben erstellte Beziehung definiert nur, dass ein Book das Feld Author hat. Wir können nach aktuellem Stand nicht die Bücher eines Autors herausfinden da wir diese Kante nicht deklariert haben. Dies ist allerdings möglich. Definieren wir den Author dann also wie folgt:

```

type Book {
  id: Int
  title: String
  author: Author
}
type Author {
  id: Int
  name: String
  written: [Books!]
}

```

so ergibt sich folgender, zyklischer Graph:



3.3.3 der komplette Graph

Wir wissen nun also wie ein GraphQL-Schema einen Graphen repräsentiert und können Algorithmen die auf Graphen funktionieren auch auf einem GraphQL-Schema anwenden. Abschließend zu diesem Kapitel sei gesagt, dass es in GraphQL allerdings nicht möglich ist, jeden einzelnen Knoten direkt abzufragen. Dies hängt stets von der Definition des Query/Mutation-Types ab, denn alle eingehenden Anfragen müssen vom Query-Type bzw. alle eingehenden Veränderungen müssen über den Mutation-Type ablaufen. Der Einfachheit halber werden wir uns jetzt erstmal dem Query-Type widmen. Jede Anfrage die in GraphQL getätigt werden kann, hat ihren Ursprung im Query-Type. Somit folgt auch, dass alle Routen stets im Query-Type entspringen müssen. Dies bedeutet, dass nur Knoten mit Erreichbarkeit vom Query-Type überhaupt abgefragt werden können. In diesem Beispiel:

```
type Query {  
  book: Book  
  author: Author  
}  
  
type Book {  
  id: Int  
  title: String  
  author: Author  
}  
  
type Author {  
  id: Int  
  name: String  
  written: [Books!]  
}  
  
type Publisher {  
  name: String  
  published: [Book]  
}
```

wäre es nicht möglich einen Knoten des Publishers abzufragen da dieser Knoten nicht erreichbar ist vom Query-Type.

3.4 Testen

Indem technische Geräte und somit auch Software im umfangreichen Maßstab Einzug nehmen in nahezu alle Bereiche des Lebens ist es wichtig die Sicherheit, Qualität und Zuverlässigkeit von Software sicherzustellen. Um all dies sicherzustellen sind strukturelle Tests von Software nötig. Ziel ist es sicherzustellen, dass die Software den definierten Anforderungen und Spezifikation entspricht. Hierbei sind diverse Techniken und Ansätze verfolgt. Im Rahmen dieser Arbeit wird sich insbesondere auf Integrationstests fokussiert also Tests, die Zusammenarbeit von einzelnen Softwarekomponenten (Units) sicherstellen sollen.

3.4.1 Arten von Tests

Es existieren verschiedene Arten von Tests welche unterschiedliche Ziele haben. Nachfolgend wird eine Auswahl verschiedener Arten erklärt wobei wir in der Granularität aufsteigen, wir fangen mit feingranularen Unit-Tests an und steigen immer weiter zu kompletten System-Tests auf. In der Mitte davon liegt das Integrations-Testen. Diese Methode wird mit dieser Arbeit für einen Anwendungsfall benötigt. Die Erklärung der anderen Arten erfolgt für eine einfachere Einordnung ebendieser.

Unit Testing

Feingranulare Tests die zur Aufgabe haben, einzelne Funktionen/Methoden zu testen. Es wird insbesondere darauf geachtet, dass die einzelne Methode genau das macht, was Sie soll. Es existieren diverse Tools, die diesen Schritt vereinfachen. Diese Tools helfen einem dabei, dass man Tests definieren kann & ein erwartetes Verhalten der Funktion angibt. Eine Auswertung dieser Funktionalität wird dann vom Tool übernommen und bei etwaigen Fehlern gibt es eine Erklärung nach dem Muster: +ABC+ war erwartet, DEF ist eingetreten. Unit-Tests werden bei der Entwicklung der einzelnen Funktionen von der Software entwickelt (sollten sie zumindest). Im Idealfall wird Test-Driven-Development verwendet, hierbei wird erst der Test entwickelt und dann die Methode entwickelt, die diese Tests erfüllen muss.

Integration Testing

Eine Granularitätsebene höher sind die Integrationstests. Hier wird getestet, ob einzelne Komponenten der Software miteinander gut zusammen arbeiten, d.h. ob sie integer sind. Die Testentwicklung ist hierbei meist eher komplex da einzelne Komponenten der Software selbst eine sehr hohe Komplexität haben können. Eine automatische Testentwicklung ist auf dieser Granularitätsebene eigentlich eher selten der Fall, allerdings bietet sich im Kontext von GraphQL die Automatisierung durchaus an da die Zusammenarbeit von Softwarekomponenten mit klar definierter Struktur lohnenswert scheint. Will man allerdings die Intergration von großen Librarys als einzelne Komponente in der eigenen Anwendung testen, so gestaltet sich dies meist schwer und muss manuell getestet wer-

den da meist komplexe Datenstrukturen händisch gemockt (Mocken als Kapitel oder Glossar?) werden müssen.

System Testing

Auf der Ebene des System-Testing wird das komplette entwickelte System getestet. Hierbei ist sicherzustellen, dass alle funktionalen Anforderungen an das System eingehalten werden. Die Tests werden hierbei so realistisch wie möglich ausgeführt, d.h. das Testsystem soll möglichst nah am späteren Produktivsystem sein und Testfälle sollen die funktionalen Anforderungen der Software abdecken. So sind insbesondere alle Geschäftsprozesse zu testen. Im allgemeinen werden System-Tests als Blackbox Tests durchgeführt, dies bedeutet, dass nur externe Funktionsmerkmale getestet werden, z.B. ob eine gewünschte Interaktion stattfand. Hierbei wird keine Rücksicht auf interne Zustände genommen. Ziel ist es, die Funktionsweise so zu testen wie ein realer Benutzer die Software nutzen würde.

3.4.2 Test-Coverage

Es ist nun bekannt, welche Arten des Testens es gibt. Allerdings ist noch nicht klar, wie viele Tests ausreichend sind. Man könnte nun argumentieren, dass man einfach jede einzelne Eingabe in einem Programm testen könnte um zu sehen, dass für jede Eingabe die korrekte Ausgabe kommt. Hierbei bemerkt man jedoch relativ schnell, dass dies mit heutigen Prozessoren nicht mehr möglich ist. Als Beispiel sei hier eine simple Addition von 2 64-bit Integer genannt. Für eine komplette Testung dieser simplen Addition gibt es 2^{64} 18 Trillionen Kombinationen. Mit einem 3GHz Prozessor wäre eine vollständige Testung nach $2^{64} / 3.000.000.000$ 6.149.571 Sekunden (69 Tage) erledigt. Es ist also ersichtlich, dass schon so eine vermeintlich einfache Funktion nicht komplett testbar ist. Daher wird also ein strukturierter Ansatz benötigt, der den Testraum einerseits klein hält, andererseits trotzdem dafür sorgt, dass Fehler ausgeschlossen werden können. Hierfür wurden formale Coverage-Kriterien entwickelt.[8, vgl. S.17]

Coveragekriterien

Wie gezeigt ist ein "vollständiges Testen also ein ausprobieren aller Möglichkeiten einfach unmöglich. Hierdurch sind wir gezwungen einen anderen Ansatz zu verfolgen. Coveragekriterien liefern hierbei einen Ansatz die einem dabei helfen können, sinnvolle Tests zu entwickeln und zu entscheiden, wann genug Tests entwickelt wurden. Die Grundidee ist hierbei, dass wir Test-Requirements definieren. Ein Test Requirement ist ein spezielles Element eines Software-Artefakts, dass von einem Testfall erfüllt sein muss. [8, vgl. S.17] Hiermit ist gemeint, dass ein Test-Requirement ein spezifisches Kriterium definiert, dass durch einen Test überprüfbar wird. Ein Beispiel für ein Test-Requirement wäre es, dass wir prüfen, ob die Division von 2 und 1 wirklich 2 ergibt. Ein Coveragekriterium ist dann eine Regel beziehungsweise eine Sammlung von Regeln die eine Menge an Test-Requirements erzeugen. [8, vgl. S.17] Bleiben wir bei dem eben genannten Beispiel wäre ein Coveragekriterium hierbei, dass wir die Division durch 0 auch noch überprüfen da

diese speziell ausgeschlossen wird. Es gibt diverse Methoden wie ein Coveragekriterium entwickelt werden kann. Einige grundlegende Typen wären zum Beispiel Call-Coverage, Branch-Coverage, Boundary-Coverage oder Statement-Coverage. (Man könnte hier noch weiter darauf eingehen aber das wäre nur um mehr Text zu haben TODO)

4 Graphcoverage

Wie zuvor gesehen, existieren verschiedene Coverage-Kriterien um Testabdeckung zu prüfen. Graphcoverage ist hierbei eine Herangehensweise um graphenbasierte Datenstrukturen zu überdecken. Graphen können nämlich ähnliche Probleme aufweisen wie vorheriges Beispiel der Addition. Die Addition zweier 64-bit Integer ist wenigstens endlich, Graphenstrukturen haben sogar unter Umständen unendliche Testräume. Umso wichtiger ist es hier, dass Überdeckungskriterien formuliert werden können, die diesen möglicherweise unendlichen Suchraum stark verkleinern und dennoch eine ausreichende Testung ermöglichen. Da wir vorher ergründet haben, dass GraphQL sich als gerichteten Graph darstellen lässt, können wir nun die Graphcoverage nutzen, um Tests mithilfe der Graphcoverage zu erstellen. Wie genau die Coverage erstellt wird und daraus Tests resultieren, werden im folgenden geklärt. Gerichtete Graphen sind die Grundlage für viele Coverage-kriterien, wobei die Grundidee hierbei ist, Sachverhalte als Graphen zu modellieren und dann eine ausreichende Überdeckung zu finden. [8, vgl. Software-testing S. 27 2.1] In 4.1.2 wurden gerichtete Graphen bereits eingeführt, daher können wir direkt fortfahren und verschiedene Kriterien definieren, die einen Graphen überdecken. Wir erklären zuerst verschiedene Techniken, die einen Graphen überdecken und erklären dann ihre Anwendung an Beispielen. Erst sortieren wir Graphcoverage ein im Kontext von Code-Coverage und bilden im zweiten Schritt eine Coverage für GraphQL.

4.1 Graphcoverage allgemein

Um Graphcoverage zu nutzen, verfeinern wir zuerst die allgemeine Definition von gerichteten Graphen. Ein gerichteter Graph ist ein Paar $G = (V, E)$ zweier disjunkter Mengen mit $E \subseteq V^2$ wobei alle Elemente aus E gerichtete Kanten sind. [vgl. 4.1.2] Die Definition erweitern wir nun mit:

Menge N von Knoten

Menge N_0 von Anfangsknoten, wobei $N_0 \subseteq N$

Menge N_f von Endknoten, wobei $N_f \subseteq N$

Menge E von Kanten, wobei $E \subseteq N \times N$. Hierbei ist die Menge als $initx \times targety$ definiert.

[8, 2.1 Overview]

Mithilfe dieser Definition können nun zum Beispiel Kontrollflussgraphen abgebildet werden indem die Einstiegspunkte die Anfangsknoten sind und die Endknoten die Austrittspunkte. Ein Pfad innerhalb von eben definierten Graphen, mit möglicher Länge

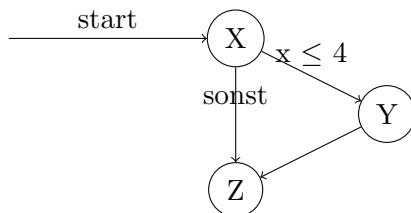
Null, der in einem Knoten N_0 startet und in einem Knoten N_f endet, nennt sich Testpfad [8, vgl. S. 28] Ziel ist es nun mit Hilfe von Coverage-Kriterien Testpfade zu ermitteln die unser Problem ausreichend testen - was hierbei ausreichend ist, ist von Fall zu Fall unterschiedlich.

4.2 Graphcoverage Kriterien

Mithilfe voriger Definition können wir nun Coveragekriterien entwickeln, die uns Testpfade liefern, die je nach Kriterium für eine spezielle Abdeckung des Graphens mit Test-Requirements sorgen.

4.2.1 Node Coverage

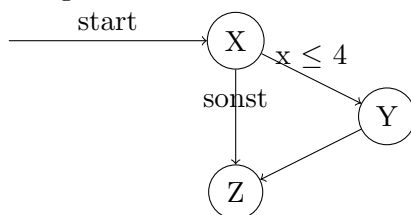
Node-Coverage ist ein Coveragekriterium, dass alle Knoten, die von N_0 erreichbar sind, in einem Graphen abdecken soll. Definieren wir folgenden, sehr einfachen Graphen:



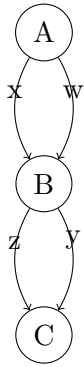
So wäre die Node-Coverage mit einem Test einzigen Test erfüllbar. Dieser ist der Pfad $X \rightarrow Y \rightarrow Z$. Es ist auch denkbar, dass wir zwei Pfade oder mehr nutzen allerdings erfüllt dieser Pfad schon unser Kriterium daher geben wir uns vorerst zufrieden. Wir sehen schnell, dass dieser Ansatz noch Lücken aufweist, da der Pfad $X \rightarrow Y \rightarrow Z$ das Kriterium erfüllt, allerdings wird eine Kante $X \rightarrow Z$ nicht im Test berücksichtigt und kann somit ungetestet bleiben. Wir führen also noch andere Kriterien ein, die eher geeignet wären.

4.2.2 Edge-Coverage

Edge-Coverage ist ein Coveragekriterium, dass die Kanten in einem Graphen abdecken soll. Ziel der Edge-Coverage ist es, dass jede Kante des Graphens durch mindestens einen Test abgedeckt wird. Um die Edge-Coverage für vorheriges Beispiel zu erreichen, benötigen wir schon zwei Routen. Der Graph:



wird über die Pfade $X \rightarrow Y \rightarrow Z$ und $X \rightarrow Z$ überdeckt. Edge-Coverage hat allerdings auch Probleme Graphen vollständig zu überdecken. Man nehme folgendes Beispiel:



Pfade die laut Edge-Coverage ausreichen um den Graphen zu überdecken wären:

$x \rightarrow z$

$w \rightarrow y$

Hierbei wird allerdings außer acht gelassen, dass in x auch Änderungen passieren können die Auswirkungen im Programm haben können. So sind die Routen $x \rightarrow y$ und $w \rightarrow z$ in der Edge-Coverage nicht berücksichtigt. Allerdings wären diese auch zu testen. Wir sehen also, dass wir immer noch kein ideales Kriterium gefunden haben.

4.2.3 Edge-Pair Coverage

Das Edge-Pair Coveragekriterium ist eine Erweiterung der Edge-Coverage, indem hier auch die Beziehungen von einzelnen Kanten untereinander berücksichtigt werden um das zuvor aufgetretene Problem zu lösen. Nach [8, Introduction to Software Testing] ist Edge-Pair Coverage: "Alle erreichbaren Pfade von Länge bis zu 2 im Testgraphen". Ziel dieses Coverage-Kriteriums ist es, dass alle möglichen Kantenpaare abgedeckt sind. Eben definiertes Beispiel hätte mit Edge-Pair Coverage eine Überdeckung mit:

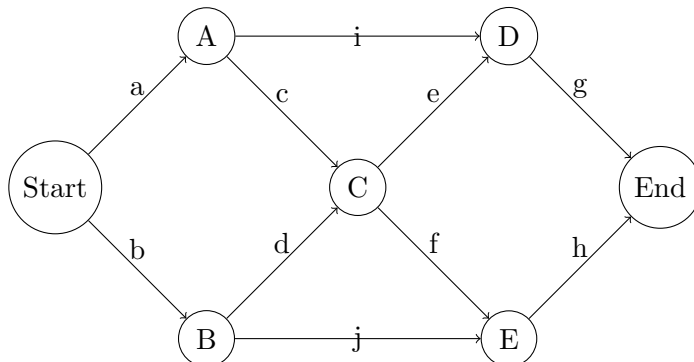
$x \rightarrow z$

$w \rightarrow y$

$x \rightarrow y$

$w \rightarrow z$

Dieses simple Beispiel wird durch Edge-Pair Coverage gut abgedeckt. Edge-Pair Coverage neigt allerdings dazu, extrem große Suchräume zu erzeugen und nur Pfadkombinationen bestimmter Länge zu betrachten. [8, vgl. S. 35] Hierdurch werden bestimmte Kombinationen von Pfaden immer noch nicht berücksichtigt.



Nach der Definition von Edge-Pair Coverage ermitteln wir erstmal alle Pfadkombinationen der Länge "bis zu 2" Dies wären:

$Start \rightarrow A \rightarrow C$
 $Start \rightarrow A \rightarrow D$
 $Start \rightarrow B \rightarrow C$
 $Start \rightarrow B \rightarrow E$
 $A \rightarrow C \rightarrow D$
 $A \rightarrow C \rightarrow E$
 $B \rightarrow C \rightarrow D$
 $B \rightarrow C \rightarrow E$
 $A \rightarrow D \rightarrow End$
 $B \rightarrow E \rightarrow End$
 $C \rightarrow D \rightarrow End$
 $C \rightarrow E \rightarrow End$

Hierdurch ergeben sich dann diese Testpfade:

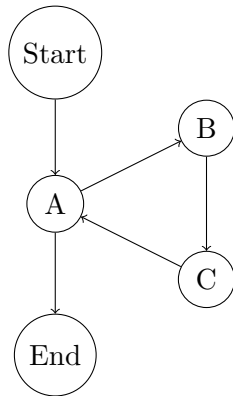
$Start \rightarrow A \rightarrow C \rightarrow D \rightarrow End$
 $Start \rightarrow A \rightarrow C \rightarrow E \rightarrow End$
 $Start \rightarrow B \rightarrow C \rightarrow D \rightarrow End$
 $Start \rightarrow B \rightarrow C \rightarrow E \rightarrow End$

allerdings fehlen auch hier wieder Pfade.

Zum Beispiel der Pfad $Start \rightarrow A \rightarrow D \rightarrow End$ fehlt. Hierdurch bleiben wieder Teile des Graphens unüberdeckt.

4.2.4 Prime-Path Coverage

Die Prime-Path Coverage verlangt, dass jeder (Prime)Primärpfad durch mindestens einen Testpfad abgedeckt sein muss. Ein Primärpfad ist definiert als ein einfacher Pfad, der nicht vollständig als zusammenhängender Teil in einem anderen einfachen Pfad enthalten ist. [8, vgl. S. 35] Hierbei ist ein einfacher Pfad dann ein Pfad, in dem keine Kanten und keine Knoten wiederholt werden, mit Ausnahme möglicherweise des ersten und letzten Knotens (wenn sie gleich sind, handelt es sich um einen Kreis). In diesem Graphen:



Sind dies alle Prime-Paths:

$Start \rightarrow A \rightarrow End$
 $Start \rightarrow A \rightarrow B \rightarrow C$
 $A \rightarrow B \rightarrow C \rightarrow A$
 $B \rightarrow C \rightarrow A \rightarrow B$
 $C \rightarrow A \rightarrow B \rightarrow C$
 $B \rightarrow C \rightarrow A \rightarrow End$

und schon zwei Testpfade würden ausreichen um die Prime-Path Coverage zu erfüllen. Die Testpfade die das Testrequirement erfüllen sind:

$Start \rightarrow A \rightarrow End$
 $Start \rightarrow A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow A \rightarrow End$

4.2.5 Complete-Path Coverage

Als letztes Coveragekriterium wollen wir die Complete-Path Coverage einführen. Ziel dieses Coveragekriteriums ist es, dass jeder mögliche Pfad mit einem Test abgedeckt werden kann. Wir brauchen dieses Kriterium nicht ausführlich definieren, da in zyklischen Graphen eine Complete-Coverage nicht möglich ist. Kreise in Graphen führen zu einer unendlich großen Menge an Pfaden und wir können keine unendlich große Menge behandeln. Die Complete-Path Coverage ist sinnvoll wenn wir Kreise verbieten und der Testgraph nicht zu groß ist. Um unsere Methode jedoch von [4, Property-based Testing] abzugrenzen wollen wir explizit Kreise erlauben. Ja sogar fördern um zu symbolisieren warum unsere Methode eine Verbesserung darstellt.

4.2.6 abschließender Vergleich der Coverage-Kriterien

4.3 Graphcoverage für Code

4.4 Graphcoverage für GraphQL

5 related Work / verwandte Arbeiten

Da GraphQL eine stetig wachsende Beliebtheit verzeichnet [9][vgl. Language Features] steigt auch der Bedarf und das Interesse an Testmethoden. Aktuell gibt es für GraphQL noch eine Lücke an produktionsreifen Testtools, insbesondere automatischen Testtools. Eine wachsende Anzahl an research-tools beziehungsweise untersuchten Methoden ist allerdings zu verzeichnen. In diesem Kapitel sollen diese Methoden benannt werden und Verwandheiten, Unterschiede oder thematische Überschnitte von dieser und anderen Arbeiten benannt werden.

5.1 Property Based Testing

In "Automatic Property-based Testing of GraphQL APIs" [4] wird der Ansatz des Property-based Testing verfolgt, um Integrationstests zu erstellen. Property-based Testing ist laut dem Paper heute Synonym mit Random Testing" [4][vgl. 2B] wobei zufällig hierbei meint, dass die Eingabedaten und Routen zufällig generiert werden. Wie Eingangs schon erwähnt, hat die Methode einige Limitierungen. Wir wollen diese hier noch einmal aufgreifen und vertiefen. Der allgemeine Funktionsablauf der Testgenerierung laut Paper ist wie folgt:

1. Vom Schema, generiere Typ-Spezifikationen
 2. Generiere einen Generator der zufällig eine Liste an Query-Objekten erstellen kann
 3. Generiere n Querys
 4. Transformiere die Queries in GraphQL-Format
 5. Führe die Queries auf dem SUT (system under test) aus
 6. Evaluiere die Ergebnisse auf ihre Properties
- [[4][vgl. 3. Proposed Method]]

Insbesondere Punkt 2. und Punkt 6. weisen Verbesserungsbedarf auf. Punkt 2 wird auch der Hauptunterscheidungspunkt beider Arbeiten sein, denn hier sind dann zwei gänzlich unterschiedliche Konzepte am Werk. Dieser ist beim Property-Based Testing nämlich ein Query-Generator der mithilfe der Clojure-Bibliothek Serene[10] Clojure.Specs[11] generiert und diese Clojure.Specs[11] dann nutzt um mit der Clojure-Bibliothek Malli[12] dann Daten für die Testqueries zu generieren. Unsere herangehensweise wird sich hiervon gänzlich unterscheiden. Im Sinne von Property-based Testing ist diese Herangehensweise allerdings eine sehr sinnvolle gewesen da Malli[12] de-facto Standard für Property-based

Testing in der Clojure-Welt ist. Geht man jedoch davon aus, dass das Ziel eine ideale Überdeckung des Graphens jeder Größe und jeder Struktur ist, so ist diese herangehensweise nicht die beste.[4][vgl. 3C] Laut dem Paper gilt "ein größeres und mehr rekursives (GraphQL)-Schema würde nicht skalieren und der (zufällig) iterative Ansatz ist besser als eine Breitensuche"[4][vgl. 3C]. Diese Behauptung betrachten wir als falsch und behaupten, dass es besser möglich ist. Dies zu zeigen bleibt Gegenstand der folgenden Arbeit.

5.2 heuristisch suchenbasiertes Testen

EvoMaster[1] ist ein Open-Source Tool welches sich automatisiertes Testen von Rest-APIs und GraphQL APIs zur Aufgabe gemacht hat. Aktuell kann durch EvoMaster sowohl WhiteBox Testing als auch BlackBox Testing durchgeführt werden jedoch ist ein Whitebox Test mittels Vanilla-EvoMaster nur für Rest-APIs möglich die mit der JVM lauffähig sind. Im Paper "White-Box and Black-Box Fuzzing for GraphQL APIs"[13] wurde ein System on-Top für EvoMaster erstellt welches GraphQL Tests generieren kann. Hierbei soll sowohl WhiteBox als auch BlackBox Testing möglich sein. Das erstellte Framework in diesem Paper arbeitet nach folgendem Prinzip:

WhiteBox Testing ist möglich insofern Zugang zum GraphQL-Schema und zum Source Code der API gegeben ist. Andernfalls ist nur BlackBox Testing möglich. Zur Testgenerierung wird ein genetischer Algorithmus genutzt welcher die Tests generiert. Wie dieser genetische Algorithmus genau funktioniert kann im Paper selbst nachgelesen werden[13]. Im Vergleich mit unserer geplanten Arbeit mittels des Prime-Path-Algorithmus ergeben sich einige Unterschiede, diese sind unter anderem: Nutzung eines evolutionären Algorithmus Many-Independent-Objective (MIO). Im Paper selbst wird davon ausgegangen, dass andere evolutionäre Algorithmen unter Umständen passender wären als der MIO Algorithmus für die Testgenerierung. Jedoch ist ein evolutionärer Algorithmus auch immer ein stochastisch, heuristisch sich dem Optimum annähernder Algorithmus. (Beleg hierfür) Im Gegensatz dazu ist der Ansatz dieser Arbeit ein iterativer Algorithmus der ideale Überdeckungen auf direkte Art bietet und im ersten Durchlauf direkt sein ideales Ergebnis ermittelt. Die ideale Lösung bezieht sich hierbei auf bestimmte Code-Coverage Kriterien die durch unseren Algorithmus erfüllt werden. Inwiefern der evolutionäre Algorithmus diese Kriterien erfüllt bleibt offen, es ist jedoch davon auszugehen, dass er sich einer idealen Lösung dieser Kriterien nur annähert da er eben ein stochastischer Algorithmus ist. (beleg oder Quelle)

5.3 Deviation Testing

Da GraphQL dynamisch auf Anfragen reagiert und es somit möglich ist, in seiner Anfrage einzelne Felder mit einzubeziehen oder auch auszuschließen ist dies im Grunde genommen ein einzelner Test-Case. Im Paper Deviation Testing: A Test Case Generation

Technique for GraphQL APIs" wird diese Gegebenheit benutzt und aus einer selbstdefinierten Query werden hier einzelne Test-Cases gebildet. Ein solcher Test macht je nach Implementierung der GraphQL-Resolver durchaus Sinn, da im Backend Felder durchaus zusammenhängen können und es Bugs geben kann wenn Resolver fehlerhaft definiert sind. z.B. könnte folgende Definition zu solchen Fehlern führen:

(hier BSP mit Code einfügen)

Da Deviation Testing jedoch nur bestehende Tests erweitert um mögliche Felder mitzutesten werden hier keine neuen Tests generiert. Durch Deviation Testing werden bestehende Tests nur erweitert allerdings muss eine Edge-Coverage gegeben sein damit diese Arbeit ein zufriedenstellendes Ergebnis erzeugt. Eine Edge-Coverage in einem komplexen Graphen ist allerdings sehr wahrscheinlich schwer umsetzbar mit manuellem Test schreiben. Eine Paarung von Edge-Coverage mit Deviation-Testing wäre sicherlich Interessant. Genau so wäre es interessant Deviation Testing als Teil unserer Arbeit zu nutzen indem mit diesem Tool die Tests erweitert werden. (initialer Plan war es, einfach immer alle Felder eines Nodes zu testen, hierdurch wäre es möglich auch alle Varianten noch zu testen)

5.4 Query Harvesting

Klassisches Testen von Anwendungen beinhaltet, dass möglichst das komplette System getestet wird bevor es verwendet wird. Im Paper "Harvesting Production GraphQL Queries to Detect Schema Faults" wird ein gänzlich anderer Ansatz verfolgt. Hierbei ist es nicht wichtig die gesamte GraphQL-API vor der Veröffentlichung zu testen sondern echte Queries die in Production ausgeführt werden zu sammeln. Der Ansatz der hierbei verfolgt wird begründet sich daraus, dass ein Testraum für GraphQL potentiell unendlich sein kann und es sehr wahrscheinlich ist, dass nur ein kleiner Teil der API wirklich intensiv genutzt wird, sodass auch nur dieser Teil wirklich stark durch Tests abgedeckt werden muss. AutoGraphQL läuft hierbei in zwei Phasen wobei in der ersten Phase alle einzigartigen Anfragen geloggt werden. In der zweiten Phase werden dann aus den geloggten Anfragen Tests generiert. Hierbei wird für jede geloggte Query genau ein Test-Case erstellt. Bei dieser Art des Testens wird insbesondere darauf Wert gelegt, dass es keine Fehler im GraphQL Schema gibt. Dies ist ein wichtiger Teil um GraphQL-API's zu testen allerdings noch kein vollständiger Test denn hier wird außer Acht gelassen, dass eine Query konform zum GraphQL-Schema sein kann aber trotzdem falsch indem zum Beispiel falsche Daten zurückgegeben werden durch falsche Referenzierung oder ähnlichem. In dem zu entwickelndem Tool sollten alle Querys die von AutoGraphQL geloggt werden auch berücksichtigt werden da sie durch den Prime-Path Algorithmus auch ermittelt werden. Es kann allerdings sinnvoll sein AutoGraphQL als Monitoring-Software mitlaufen zu lassen und weitere etwaige Fehler hiermit zu loggen und automatisch daraus Test-Cases erstellen zu können damit zukünftig keine Fehler dieser Art mehr passieren.

Arbeit / Kriterium	Property Based Testing	heuristisch-suchen-basiertes Testen	Deviation-Testing	Query Harvesting
Generierungsart	Zufallsbasierte Routengenerierung	Heuristische Suche	Erweiterung von bestehenden Tests	Tracken von Querys und daraus Tests generieren
Überdeckung	Zufällig, stark abhängig von Schema	abhängig ob Zugang zu Source Code, Zufällig aber optimaler	stark abhängig von selbst geschriebenen Tests	stark Abhängig von User-requests
Orakel	simples Raten	mit Source Code: Analyse	Aus entwickelten Tests	Aus gestellten Querys
Ausführzeit	vor Prod	vor Prod	vor Prod	Verifikation / Wartung
Use-Case	allgemeines Testen	allgemeines Testen	allgemeines Testen	Testen bei Code-Änderung

5.5 Vergleich der Arbeiten

5.6 Andere Arbeiten

Hier ist eine kurze Übersicht über andere Arbeiten, dieses Kapitel ist sehr unwahrscheinlich in einer Abgabeverision. Es dient eher als Notizensammlung in einer hübscheren Form.

5.6.1 Empirical Study of GraphQL Schemas

Eine umfangreiche Untersuchung von Praktiken in GraphQL. Unterteilt in verschiedene Metriken wie z.B. Anzahl der Objekttypen, Querys etc. Interessant ist allerdings die Untersuchung von zyklischen Schemas. Insbesondere, wie groß diese Zyklen werden können und wie sie begrenzt werden. Dies ist interessant für spätere Auswertungen. Allerdings bringt diese Arbeit nicht viel für das direkte Testen.

5.6.2 LinGBM Performance Benchmark to Build GraphQL Servers

Eher eine Untersuchung wie GraphQL-APIs unter Last performen bzw. wie Effizient sie sind. Der benutzte Query-Generator kann interessant sein aber es ist schwer einschätzbar wie dieser in unserem Kontext genutzt werden kann.

5.6.3 GraphQL A Systematic Mapping Study

Richtig gute Übersicht wie GraphQL Unter der Haubefunktioniert

6 Testentwurf

Wie zuvor gesehen kann ein GraphQL-Schema mit Graphcoverage-Kriterien überdeckt werden. Wir wollen diese Gelegenheit nun nutzen um eine Methode zur Generierung von Integrationstests für GraphQL zu entwickeln. Hierbei werden wir uns in einigen Teilen an der Methode von [4, Property-based Testing of GraphQL-APIs] bedienen. Allerdings werden auch deutliche Unterschiede existieren. Zuerst werden wir unsere Methode entwickeln und dann folgt ein Vergleich mit der Methode aus [4, Property-based Testing of GraphQL-APIs]

6.1 Schema in Graph abbilden

Unser Ziel ist es, dass Wissen aus 5.2 anzuwenden und mithilfe von Graphcoverage die Test zu generieren. Hierfür benötigen wir erstmal einen Graphen, den wir untersuchen wollen. Mithilfe der Introspection-Query kann von einer GraphQL-API das komplette Schema mit seinen Types abgefragt werden. Die Introspection-Query ermöglicht es uns, dieses Schema von einer API abzufragen. Hierbei ist zu beachten, dass nicht alle APIs dies unterstützen da einige diese Funktion ausgeschaltet haben oder ein Depth-Limit in der Anfrage implementieren und die Anfrage zu komplex ist. Beides vernachlässigen wir in unserer Methode da im Entwicklungskontext solche Maßnahmen weggelassen werden können. Stellt man nun die Introspection-Query an eine GraphQL-API so hat die Response immer auch die erwartete Struktur. Im besonderen bedeutet das für uns, dass wir ein JSON-Objekt mit dieser Struktur bekommen:

```
1  {
2      "data": {
3          "__schema": {
4              "queryType": {},
5              "mutationType": {},
6              "subscriptionType": {},
7              "types": [],
8              "directives": []
9          }
10     }
11 }
```

Listing 6.1: Schema-Response

Bei den Feldern *queryType* , *mutationType* und *subscriptionType* wird jeweils ein Objekt erwartet. Wobei wir hier mit der Introspection-Query nur den namen dieser Fel-

der abfragen. Diese Felder sind nämlich, wie in 4.2.2 festgestellt grundlegende Typen eines jeden GraphQL-Schemas können aber unter Umständen *null* sein oder von den vordefinierten Namen *Query*, *Mutation* und *Subscription* abweichen. Um solche Abweichungen abzufangen, werden diese mit abgefragt. Im Feld *types* finden wir dann alle möglichen Typdefinitionen. Hierbei ist das Feld *types* eine Liste, die alle definierten Typen des Schemas enthält. Dies beinhaltet sowohl Custom-Types als auch die eingebauten Skalaren Datentypen. Ein einzelner Type ist wie folgt definiert:

```
1      {
2          "kind": "",
3          "name": "",
4          "description": "",
5          "fields": [],
6          "inputFields": [],
7          "interfaces": [],
8          "enumValues": [],
9          "possibleTypes": []
10     }
```

Listing 6.2: Type-Field

Um nun aus dem Schema einen Graphen zu erstellen, benötigen wir die Felder *kind*, *name*, *fields*. *kind* ist die Angabe, von welchem Typ das Feld ist. Hierbei gibt es 9 Möglichkeiten, die dieses Feld annehmen kann.

- **ObjectTypeDefinition (OBJECT):** Repräsentiert ein Objekt mit Feldern.
- **ScalarTypeDefinition (SCALAR):** Eingebaute oder benutzerdefinierte Typen wie *Int*, *Float*, *String*, *Boolean* und *ID*.
- **InputObjectTypeDefinition (INPUT_OBJECT):** Erlaubt das Übergeben komplexer Objekte als Argumente.
- **InterfaceTypeDefinition (INTERFACE):** Repräsentiert eine Liste von Feldern, die andere Objekttypen enthalten müssen.
- **UnionTypeDefinition (UNION):** Kann einen von mehreren Arten von Objekttypen repräsentieren.
- **EnumTypeDefinition (ENUM):** Ein Skalartyp, der auf eine bestimmte Liste von Werten beschränkt ist.
- **ListTypeDefinition (LIST):** Repräsentiert eine Liste von Werten eines bestimmten Typs.
- **NonNullTypeDefinition (NON_NULL):** Ein Modifikator, der angibt, dass der angewandte Typ nicht null sein kann.

- **DirectiveDefinition (DIRECTIVE):** Passt das Verhalten von Feldern oder Typen Schema an.

Um einen Graphen aus dem Schema zu entwickeln benötigen wir nur Felder vom Typ *OBJECT*. Die Menge aller Objekte vom Typ *OBJECT* sind die Menge aller Knoten unseres Graphens. Um nun die Kanten, also die Beziehungen zwischen diesen einzelnen Knoten zu bekommen müssen wir uns die Definition eines Typens näher ansehen. Wie in *Type – Field* gesehen, definiert ein Type immer ein Feld *fields*. In diesem Feld *fields* verbirgt sich die Informationen aller Kanten, die ausgehend von diesem Knoten sind. Das Feld *fields* beinhaltet Objekte folgender Struktur:

```

1      {
2          "name": "",
3          "description": "",
4          "args": [],
5          "type": {},
6          "isDeprecated": "",
7          "deprecationReason": ""
8      }

```

Listing 6.3: Type-Field

Wobei für die Kantensuche das Feld *type* besonders wichtig ist. Dieses ist wie folgt definiert:

```

1      {
2          "kind": "",
3          "name": "",
4          "ofType": null
5      }

```

Listing 6.4: Type-Field

Wenn nun der Eintrag *kind* den Wert *OBJECT* trägt, so ist klar, dass unser hier definiertes *OBJECT* eine Kante zum Knoten *name* besitzt. Man kann nun einmal über alle Einträge von *types* gehen und jeden Eintrag vom Typ *OBJECT* als Knoten anlegen. In einem zweiten Durchlauf kann man dann über alle *fields* von jedem Type gehen und die Kanten zwischen den Knoten ziehen. Hierzu folgt nun noch ein minimales Beispiel eines sehr kleinen Schemas und das Mapping zum dazugehörigen Graphen. Wir werden hierbei Schritt für Schritt vorgehen. Zuerst definieren wir ein GraphQL-Schema:

```

1  type Query {
2      book(id: ID!): Book
3      author(id: ID!): Author
4      publisher(id: ID!): Publisher
5  }
6

```

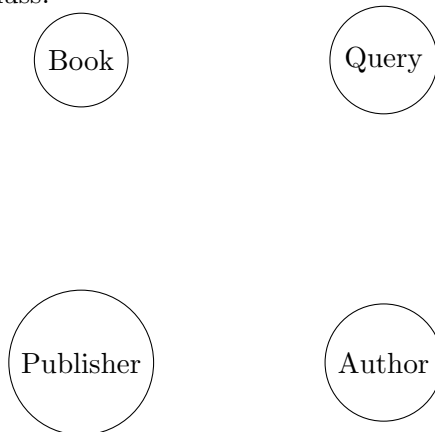
```

7  type Book {
8      id: ID
9      title: String
10     author: Author
11     publisher: Publisher
12 }
13
14 type Author {
15     id: ID
16     name: String
17     books: [Book]
18 }
19
20 type Publisher {
21     id: ID
22     name: String
23     books: [Book]
24 }

```

Listing 6.5: Schema Definition

Für dieses Schema erhalten wir dann eine JSON-Response zurück im Format wie in 6.1 vorgestellt. Die vollständige Response ist in minimale Schema Response zu finden. In dieser Response werden 4 Typen vom Typ *OBJECT* definiert, diese sind wie erwartet unsere eben definierten types *Query*, *Book*, *Author* und *Publisher*. Durch die erste Iterierung können wir also nun folgern, dass unser zu erstellende Graph 4 Knoten besitzen muss.



In einer zweiten Iteration prüfen wir nun alle *fields* Einträge des jeweiligen Types die vom Typ *OBJECT* sind. Beginnend im Query Type finden wir dort 3 Einträge in *fields*. Jedes Feld besitzt einen Namen, in diesem Beispiel sind dies *book*, *author* und *publisher*.

1

```
{
```

```

2      "name": "book",
3      "description": null,
4      "args": [
5        {
6          "name": "id",
7          "description": null,
8          "type": {
9            "kind": "SCALAR",
10           "name": "ID",
11           "ofType": null
12         },
13         "defaultValue": null
14       }
15     ],
16     "type": {
17       "kind": "OBJECT",
18       "name": "Book",
19       "ofType": null
20     },
21     "isDeprecated": false,
22     "deprecationReason": null
23   }

```

Listing 6.6: book Field

```

1      {
2        "name": "author",
3        "description": null,
4        "args": [
5          {
6            "name": "id",
7            "description": null,
8            "type": {
9              "kind": "SCALAR",
10             "name": "ID",
11             "ofType": null
12            },
13            "defaultValue": null
14          }
15        ],
16        "type": {
17          "kind": "OBJECT",
18          "name": "Author",
19          "ofType": null

```

```

20     },
21     "isDeprecated": false,
22     "deprecationReason": null
23 }

```

Listing 6.7: author Field

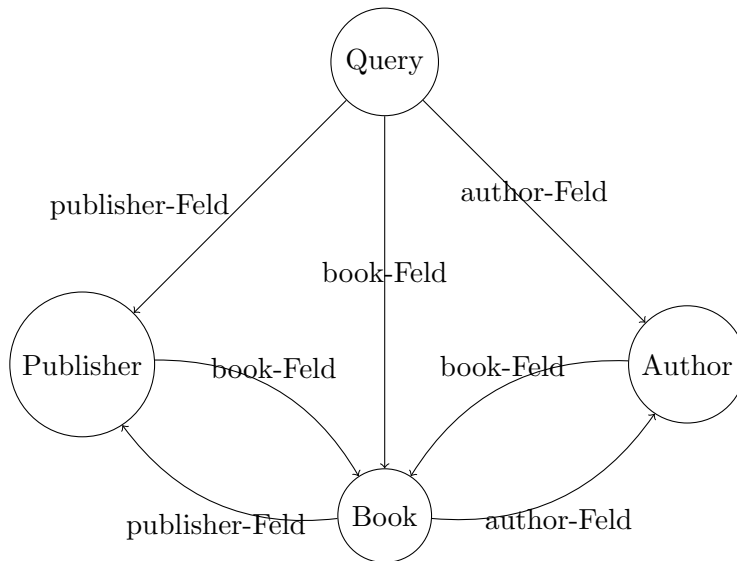
```

1      {
2        "name": "publisher",
3        "description": null,
4        "args": [
5          {
6            "name": "id",
7            "description": null,
8            "type": {
9              "kind": "SCALAR",
10             "name": "ID",
11             "ofType": null
12           },
13           "defaultValue": null
14         ]
15       },
16       "type": {
17         "kind": "OBJECT",
18         "name": "Publisher",
19         "ofType": null
20       },
21       "isDeprecated": false,
22       "deprecationReason": null
23     }

```

Listing 6.8: publisher Field

Eine gerichtete Kante muss nun ausgehend vom Query-Knoten gezogen werden jeweils zum *type* jedes einzelnen Feldes. Die Kante erhält als Gewicht hierbei dann exakt die Feld-Definition. So wird es später möglich aus Pfaden Querys zu bilden. Nachdem alle Knoten iteriert wurden und die Felder untersucht sind, ergibt sich folgender Graph für unser Schema:



Aus diesem Graphen können wir nun unsere Tests entwickeln. Hierzu in den folgenden Kapiteln mehr.

6.2 Pfade aus Graph bilden

Dieser Schritt ist optional, je nach Implementation der Algorithmen für die Graphüberdeckung. Es ist hierbei möglich einen generativen oder filternden Ansatz zu verfolgen. Wird ein generativer Ansatz verfolgt, so wird direkt 6.3 angewandt. Alternativ generieren wir erstmal alle möglichen Simple-Paths (noch definieren - TODO) ausgehend von einem Startknoten. Wir müssen die Simple-Paths vom Startknoten generieren, da in GraphQL nur Anfragen erlaubt sind, die in diesem Startknoten beginnen. Im allgemeinen Fall ist dies der Knoten *Query*. Diese Simple-Paths werden dann im nächsten Schritt gefiltert durch das jeweilige Coverage-Kriterium. In unserem Beispielgraphen sind die Simple-Paths folgende:

- (Query, Book)
- (Query, Author)
- (Query, Publisher)
- (Query, Author, Book)
- (Query, Publisher, Book)
- (Query, Book, Author)
- (Query, Book, Publisher)
- (Query, Publisher, Book, Author)
- (Query, Author, Book, Publisher)

6.3 Coverage-Pfade ermitteln

Wie zuvor unterschieden, gibt es zwei Verfahren um die Pfade zu generieren. Wir werden hier beide Verfahren getrennt voneinander Zuerst betrachten wir den filternden Ansatz da er thematisch zum vorherigen Kapitel abschließend ist.

6.3.1 filternder Ansatz

Aus der zuvor gewonnen Menge an Simple-Paths können wir nun je nach Coverage-Kriterium die Pfade, die für unser Coverage-Kriterium wichtig sind, herausfiltern. Exemplarisch nutzen wir hierfür einmal die Prime-Path Coverage. Es ist jedoch denkbar auch andere Coverage-Kriterien zu verwenden. Wir erinnern uns: Ein Prime-Path ist ein Pfad, der weder selbst Teil eines anderen Pfades ist noch sich wiederholt. Dies bedeutet, dass wir alle Pfade aus den Simple-Paths danach filtern müssen, dass diese weder Teilpfad eines anderen Pfades sind noch, dass sie sich wiederholen. Hier kann man folgenden Pseudo-Code nutzen um einen Filter für PrimePaths zu entwickeln:

```
Input: alle_Pfade
```

```
prime_paths = []
```

```
Für alle_Pfade:
```

```
    Wenn istPrimePfad(Pfad, alle_Pfade):
```

```
        prime_paths += Pfad
```

```
    Sonst verwerfe Pfad
```

```
return prime_paths
```

```
Funktion istPrimePfad(möglicherPrimePath, alle_Pfade):
```

```
    Für pfad in alle_Pfade:
```

```
        Wenn möglicherPrimePath != pfad and istTeilpfad(möglicherPrimePath, pfad)
```

```
            return False
```

```
    return True
```

```
Funktion istTeilpfad(möglicherPrimePath, pfad):
```

```
    Wenn Länge(möglicherPrimePath) > Länge(pfad):
```

```
        return False
```

```
    Für jeden TeilPfad von Pfad:
```

```
        Wenn TeilPfad = möglicherPrimePath:
```

```
            return True
```

```
    return False
```

Dieser PseudoCode bewirkt, dass wir für jeden Pfad ermitteln ob dieser ein PrimePath

ist. Hierbei iterieren wir über jeden einzelnen Pfad und prüfen ob dieser ein PrimePfad ist. Hierbei fügen wir einen Pfad der Liste alle Prime-Paths hinzu, wenn dieser die Funktion `istPrimePfad` mit `True` erfüllt. Die Funktion liefert hierbei nur `True` zurück, wenn der Pfad sich nicht wiederholt und er die Funktion `istTeilpfad` mit `False` belegt. Somit erreichen wir genau, dass nur diejenigen Pfade mit `True` in die Liste gegeben werden, die genau unsere eingangs definierten Bedingungen erfüllen. Nutzen wir nun diesen Pseudo-Code auf unserer Menge der SimplePaths so bekommen wir folgende PrimePaths:

- (Query, Book, Author)
- (Query, Publisher, Book, Author)
- (Query, Book, Publisher)
- (Query, Author, Book, Publisher)

Diese filternde Methode ist im Allgemein einfacher zu implementieren, bietet jedoch einen großen Nachteil: Die Generierung der SimplePaths ist sehr rechenintensiv und außerdem werden hierfür Pfade nur entfernt aus der größeren Menge. Dies bedeutet, dass die SimplePaths eine schon sehr gute Abdeckung bieten und die PrimePaths den Testraum nur potentiell verkleinern da wir eben nur aus der SimplePath Menge dann Pfade löschen. Die Methode des Filterns eignet sich also am ehesten für kleine Schemas da hier das bilden der SimplePaths noch relativ rechenarm ist und zur Validierung von verschiedenen CoverageKriterien da die generative Implementierung komplexer ist. Hierdurch kann man also vorher überprüfen, ob es sich lohnt ein anderes CoverageKriterium mittels generativem Ansatz zu implementieren. Der generative Ansatz gewinnt dann an Bedeutung wenn die GraphQL-Schemas größer werden da so Rechenzeit gespart werden kann. Es sei außerdem erwähnt, dass die Ausführungszeit der Tests auch relevant sein kann und somit die Anzahl an Pfaden durch Filtern reduziert werden muss.

6.3.2 generativer Ansatz

Mittels eines generativen Ansatzes lassen sich von einem gegebenen Startpunkt die Pfade ermitteln, welche die gewünschte Coverage erreichen. Hierbei unterscheidet sich der konkrete Ansatz je nach Coverage-Kriterium wieder. Im allgemeinen wird eine Funktion `generatePaths(Startpunkt, Graph)` erwartet. Wobei der Startpunkt im Allgemeinen der *Query* Knoten ist und *Graph* den kompletten Graphen abbildet. Erwarteter Rückgabewert der Methode ist eine Liste von Pfaden die das jeweilige CoverageKriterium erfüllen. Wir werden auch hier wieder einen Pseudo Code vorstellen, der für die Generierung von Prime-Paths zuständig ist aber allerdings den generativen Ansatz verfolgt.

```
testPfade = []
Generiere_Prime_Pfade(startknoten, [], testPfade, Graph)
```

```

Function Generiere_Prime_Pfade(Knoten, Pfad, testPfade, Graph):
    Füge den Knoten dem Pfad hinzu
    Für alle FolgeKnoten von Knoten:
        Wenn FolgeKnoten nicht in Pfad:
            generiere_Prime_Pfade(FolgeKnoten, testPfade, Pfadliste, Graph)
    Wenn Ist_PrimePfad?(Pfad, testPfade):
        Füge Pfad den testPfaden hinzu

Function Ist_PrimePfad?(neuerPfad, testPfade):
    Für alle Pfade in TestPfade:
        Wenn neuerPfad ein Teilpfad eines Pfades ist
            return False
    return True

```

Der Pseudocode startet im Startpunkt des Graphens und iteriert dann rekursiv über die Folgeknoten des Graphens. Hierbei wird stets in Beachtung gehalten, dass ein Knoten in einem Pfad nicht zweimal vorkommen kann. So vermeiden wir Kreise und eine mögliche, unendlichen Generierungsraum. Für jeden generierten Pfad wird dann überprüft ob dieser ein Prime-Path ist. Dies ist der Fall wenn er kein Teilpfad eines anderen Pfades ist und sich nicht selbst wiederholt. Wir stellen sicher, dass der Pfad sich nicht wiederholen kann indem wir Kreise erst gar nicht zulassen. Im zweiten Schritt prüfen wir dann ob ein Pfad ein Teilpfad ist. Endergebniss sind dann die Prime-Paths des Graphens. Dieser Code ergibt dann folgende Pfade für die PrimePath-Coverage die den Graphen überdecken:

- (Query, Book, Author)
- (Query, Publisher, Book, Author)
- (Query, Book, Publisher)
- (Query, Author, Book, Publisher)

Wir sehen diese sind identisch zum filternden Ansatz.

6.4 Query aus Pfad ermitteln

Da wir nun die Pfade, je nach Coveragekriterium ermittelt haben müssen wir nun eine Query aus diesen Pfaden bilden. Hierbei nutzen wir diverse Hinweise die im GraphQL-Schema gegeben werden und einen Teil der Arbeit aus [4, Property-Based Testing]. Jeder ermittelte Pfad startet im Query-Knoten. Da wir zuvor den Kanten jeweils die Feldattribute mitgegeben haben, können wir nun sehr einfach ermitteln welche Operationen wir ausführen müssen, um von einem Knoten zum nächsten zu kommen. Dies bedeutet,

dass die Kanteninformationen ausreichen, um eine Query zu bilden. Wir gehen hierbei den kompletten Pfad ab und geben einen Query-String zurück der für GraphQL zulässig ist. Eine valide GraphQL-Query beginnt mit `{}` und folgt dann mit der ersten Felddefinition. Eine Felddefinition kann vom Type *SCALAR* oder *OBJECT* sein. Ist ein Feld vom Type *SCALAR* so kann dieses einfach der Query hinzugefügt werden als erwartetes Feld. GraphQLs Spezialität ist es, dass nur wirklich angeforderte Felder in einer Response existieren. Im Kontext des Integrationstesten wollen wir aber alle möglichen Felder abfragen einfach um sicher zu gehen, dass diese korrekt installiert wurden. Es wäre denkbar auch Variationen der einzelnen Felder zu implementieren, dass nicht in jeder Query immer alle Felder eines Typen berücksichtigt werden, dies wird hier jedoch zuerst vernachlässigt und wir inkludieren alle Felder. Ist ein Feld vom Type *OBJECT* so bedeutet dies, dass dieses Feld nur angegeben werden muss, wenn der nachfolgende Knoten im Pfad exakt den Typen dieses Feldes hat! Das Feld wird dann wieder eingeleitet durch `{}`. Innerhalb dieser Felder ist nun wieder jedes Feld des Typen *SCALAR* hinzuzufügen und das Feld des nächsten Knotens zu suchen sowie hinzuzufügen. Sollte ein Feld vom Type *OBJECT* Argumente benötigen, so steht dies im Schema und wir können diese generieren. Input-Argumente können vom Type *INPUT – OBJECT* sein oder *SCALAR*. Sollte es ein Scalar Type sein, so bedienen wir uns der Methode aus [4, Property-based Testing] und generieren jeweils Zufallsargumente in Form des jeweiligen Datentyps. Ein String bedeutet also, dass wir einen beliebig langen String erzeugen, ein Integer ist eine Zufallszahl usw. Sollte das Argument vom Type *INPUT – OBJECT* sein so lässt es sich in seine Bestandteile zerlegen welche auch wieder *SCALAR* Types sind und dann werden diese auch zufällig entsprechend generiert und dem *INPUT – OBJECT* Type entsprechend angeordnet. Argumente folgen dann jeweils zwischen dem Namen des Felds vom *OBJECT* und den sich öffnenden, geschweiften Klammern. Für unser Beispiel von vorher bedeutet dies also, dass wenn wir aus dem Pfad (*Query, Book, Publisher*) eine Query bilden wollen, wir folgende Schritte zu erfüllen haben:

1. Finde Kante von *Query* zu *Book*
2. Ermittle ob die Kante (*Query, Book*) Argumente benötigt.
3. Ermittle alle *SCALAR* Types die *Book* definiert.
4. Ermittle die Kante von *Book* zu *Publisher*
5. Ermittle ob die Kante (*Book, Publisher*) Argumente benötigt.
6. Ermittle alle *SCALAR* Types die *Publisher* definiert. (Pfad zuende daher kein Folge-Typ)
7. Baue Query-String aus den erlangten Informationen

Wir werden nun Schritt für Schritt alle Schritte einzeln durchgehen und eine Query generieren.

Dieses Prozedere muss nun mit allen ermittelten Pfaden durchgeführt werden um aus den Pfaden die Queries zu generieren welche die Tests darstellen. Da wir uns bei der

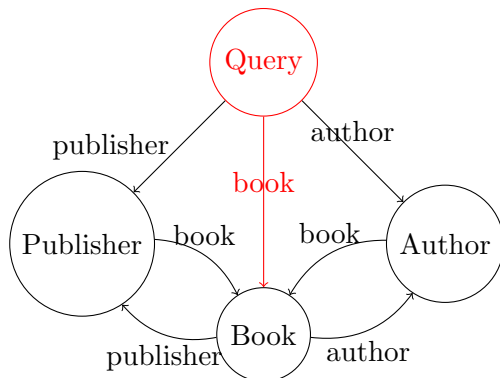


Abbildung 6.1: Schritte 1 - 3

Wir betrachten die Kante (Query, Book). Durch unsere vorherige Definition des Graphens wissen wir, dass diese Kante das Attribut *book* trägt und uns somit hinweist, welches Feld wir benötigen um diese Kante in unserer Query abzubilden. Dieses entspricht *Listing6.6* und wir erkennen, dass das Book-Field ein Argument *id* vom Typ *ID* besitzt. Daher wissen wir, dass wir dem Book-Field ein Argument (*id* : *RandomString*) mitgeben müssen. Im nächsten Schritt suchen wir noch alle *SCALAR* Types die in Book definiert sind. Hierzu schauen wir in der minimal-schema-response im Type *Book* nach und finden, dass ein *Book* die Felder *title* und *id* vom *SCALAR* Type hat.

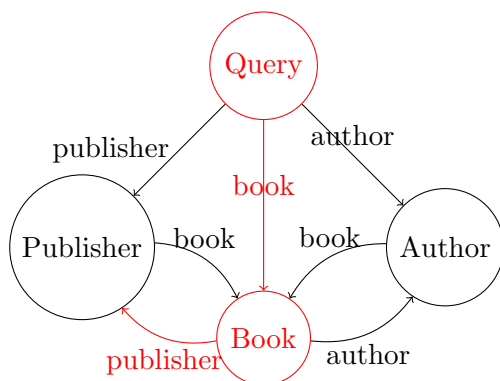
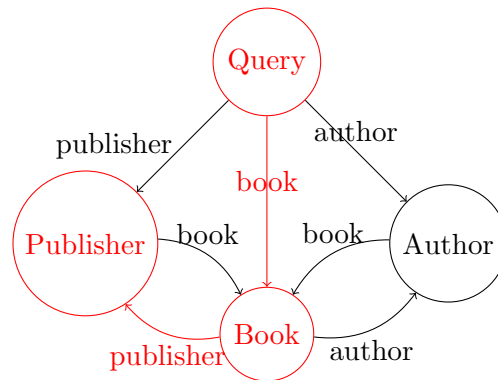


Abbildung 6.2: Schritte 4 - 6

Im Folgenden wollen wir die Kante (Book, Publisher) betrachten um unseren Pfad abzuschließen. Wie zuvor finden wir, dass die Kante das Attribut *publisher* trägt. Wir müssen nun allerdings nicht im Query-Type das Feld *publisher* suchen sondern im aktuell befindlichen Knoten (Book). Hierbei finden wir, dass vom Type *Book* das Feld *publisher* kein Argument benötigt. Wir müssen daher nur noch ermitteln, welche Felder vom *SCALAR* Type in Publisher definiert werden. Dies sind die Felder *id* und *name*.



Wir haben nun unseren Pfad vollständig abgebildet. Nun müssen wir aus den gewonnenen Informationen einen Query-String bilden. Hierfür starten wir mit zwei geschweiften Klammern:

```
1 {
2 }
```

Als Erstes fügen wir das gewählte Feld "Book" mit seinem Argument hinzu und fügen wieder zwei geschweifte Klammern hinzu für die Felder des Types. In diese geschweiften Klammern kommen zuerst die *SCALAR* Types. Dadurch erhalten wir die Query:

```
1 {
2     book(id: RandomString){
3         id
4         title
5     }
6 }
```

Danach folgt das Feld, das die nächste Kante abbildet. In diesem Fall hat es kein Argument, aber wir fügen geschweifte Klammern mit dem Kantenbezeichner hinzu. Dadurch erhalten wir die Query:

```
1 {
2     book(id: RandomString){
3         id
4         title
5         publisher {}
6     }
7 }
```

Zum Schluss fügen wir alle *SCALAR*-Felder des *Publisher*-Types hinzu, um auch die letzte Kante abzubilden. Diese Felder sind *id* und *name*. Die fertige Query lautet also:

```
1 {
2     book(id: RandomString){
3         id
4         title
5         publisher {
6             id          42
7             name
8         }
9     }
10 }
```

Diese Query deckt unseren Pfad (*Query*, *Book*, *Publisher*) ab.

Generierung der Argumente für Querys auf [4, Property-based Testing] beziehen ist es unter Umständen möglich, dass Querys Argumente bekommen, die nicht zur zugrundeliegenden Datenstruktur passen. D.h. in der Standardmethode wird für ein zufälligen Integer-Wert direkt der gesamte Integer Wertebereich genutzt. Ist der Datenraum jedoch sehr klein, so kann es sein, dass kein einziger existenter Wert getroffen wird und die Query nur eine leere Antwort wiedergibt da das Element nicht gefunden wurde. Hier kommt es auf die spezielle Implementierung einerseits des SUT an andererseits auch auf die Programmierung der Argumentgeneratoren. Hierzu jedoch im Praxisteil mehr.

6.5 Test ausführen & Testauswertung

Für die Testausführung halten wir uns wieder ähnlich zum [4, Property-based Testing]. Durch die zufällige Argument-Generierung ist es leider nicht möglich, strukturiert zu überprüfen, ob die zurückgelieferten Daten passend sind zu dem was wir angefragt haben. Hier hat unser Ansatz die selben Probleme wie das [4, Property-based Testing]. Zur Überprüfung der Querys stellen wir diese ganz einfach per HTTP-Post an den zu testenden GraphQL-API Endpunkt. In der Response erhalten wir dann einen StatusCode welcher üblicherweise zwischen 2XX und 5XX liegt. Wir werden nun die einzelnen StatusCodes aufbrechen, da diese inherent wichtig sind für unsere Testauswertung. Responses von 2XX werden in Kapitel *PositiveTests* behandelt, 4XX in Falsch-Negative Tests und 5XX in Negative Tests.

6.5.1 Positive Tests

Einen positiven Test zeichnet aus, dass dieser einen HTTP-Response Code von 2XX erhält. Im Allgemeinen Fall sagt dies aus, dass alles mit der Anfrage gut lief und wir eine Antwort erhalten haben die unseren Erwartungen entspricht. Dies bedeutet, dass für unser Beispiel zuvor die Query

```
1      {
2          book(id: RandomString){
3              id
4              title
5              publisher {
6                  id
7                  name
8              }
9          }
10     }
```

eine Response zurückgeliefert hat die dieser erwarteten Struktur entspricht. Im Allgemeinen wird also erwartet, dass unsere Query dann zum Beispiel diese Response liefert:

```
1      {
2          "data": {
```



```

3         "book": {
4             id: "1",
5             title: "Moby Dick"
6             publisher: {
7                 id: "1",
8                 name: "Testverlag"
9             }
10        }
11    }
12 }

```

Allerdings bedeutet dies, dass wir den glücklichen Fall haben, dass das Argument von *book(id : Argument)* die passende ID für ein Buch war. Dadurch, dass wir jedoch unsere Eingabevariablen zufällig generieren, können wir nicht davon ausgehen, dass uns dies regelmäßig gelingt. Sehr wahrscheinlich wird eine Response zwar den Status-Code 200 liefern allerdings werden die enthaltenen Daten der Response dieses Format haben:

```

1    {
2        "data": {
3            "book": null
4        }
5    }

```

Listing 6.9: mangelhafte Response

Gegen dieses Verhalten wollen wir Maßnahmen einführen damit sichergestellt werden kann, dass zumindest eine signifikante Aussage über die tatsächliche Testabdeckung ausgeführt werden kann - denn wie zuvor erwähnt, werden Funktionen tiefer im Pfad erst ausgeführt, wenn das Objekt zuvor erfolgreich ermittelt wurde.

Unterschiede in den Pfadlängen

Diese Methode verbessert zwar nicht die Testergebnisse allerdings gibt Sie uns Informationen darüber wie viel von unserem Pfad in Wirklichkeit abgedeckt wurden. Dadurch lässt sich der Erfolg der Tests besser abschätzen da wir so messen können ob die Querys wirklich die Funktionen ausgeführt haben. Hierzu wird die Pfadlänge des Pfades der zur Erstellung der Query genutzt wurde als erwartete Pfadlänge angenommen. Die Pfadlänge der Antwort wird dann als tatsächliche Pfadlänge genommen. Der Unterschied zwischen erwarteter und tatsächlicher Pfadlänge ist dann unser Auswertungsmerkmale für diesen speziellen Test. Die Pfadlänge der Response ist die maximale Tiefe der JSON-Response verringert um 1.

$$\text{Tiefe des Pfades} = \text{Tiefe des JSON-Response-Objekts} - 1$$

Demnach hätte folgende Response eine Tiefe von 2

```

1      {
2          "data": {
3              "book": {
4                  id: "1",
5                  title: "Moby Dick"
6                  publisher: {
7                      id: "1",
8                      name: "Testverlag"
9                  }
10             }
11         }
12     }

```

Listing 6.10: vollständige Response

Und die leere Antwort hätte eine Tiefe von 1

```

1      {
2          "data": {
3              "book": null
4          }
5      }

```

Listing 6.11: mangelhafte Response

Der Unterschied zwischen beiden signalisiert uns dann, ob die erfolgreiche Query denn die komplette Query durchlaufen ist oder nur ein Teil davon. Hierdurch gelingt uns eine Auswertung. Wir können die Pfadlängen aller erwarteten Pfade addieren, das gleiche können wir auch mit den tatsächlichen Pfaden machen. So erreichen wir zwei Zahlen und mit diesen können wir eine Prozentuale Einschätzung abgeben, wieviel Prozent unserer Tests insgesamt ausgeführt wurden. Wir rechnen hierfür:

$$\text{Prozent der tatsächlichen Abdeckung} = \frac{\text{tatsächliche Gesamtpfadlänge}}{\text{erwartete Gesamtpfadlänge}} * 100$$

Wir sollten einen Wert nahe der 100% anstreben. Dies würde bedeuten, dass unsere generierten Tests auch alle Funktionen getestet haben. Andernfalls bedeutet ein Prozentsatz unter 100% eben, dass nicht alle Funktionen tatsächlich von den Querys überdeckt wurden.

Zufallsgeneratoren der Argumente

Unser zuvor vorgestellte Methode liefert uns einen Hinweis darauf, wie gut unsere Querys tatsächlich getestet haben. Dieser Ansatz verfolgt das Ziel die Querys eine bessere tatsächliche Abdeckung zu erreichen. Hierbei ist das Anpassen der Generatoren für die Argumente im Fokus. In der vorgestellten Methode unter 6.4 erstellen wir komplett

zufällig Argumente für die Funktionen. Dies bedeutet, dass z.B. der Type *ID* als String gewertet wird. Dieser Type ist meist sehr bedeutend, da dieser häufig als Argument angegeben wird und eine spezielle Struktur hat. Es hängt natürlich stark von der eigenen Implementierung der GraphQL-API ab allerdings wenn in der Implementierung eine *ID* definiert ist als Zahlenstring, so kann es sich durchaus lohnen, dass der Argumentgenerator für die ID auch speziell auf Zahlenstrings angepasst wird. Alternativ kann auch eine Liste aller existenten IDs angegeben werden und zufällig ausgewählt werden. Wir erhöhen hierdurch letztendlich einfach die Chance, dass zufällig eine tatsächliche ID gewählt wird die auch in den Daten des SUT existent ist. Eine exakte Vorgehensweise für diese Methode ist allerdings stark abhängig vom System das zu testen ist und daher stark Fall abhängig.

Anzahl der Querys

Wir haben bisher eingeführt, dass wir messen können, welche Abdeckung unsere Querys tatsächlich haben und haben festgelegt wie wir diese Verbessern können. Da wir zufallsbasiert die Argumente generieren können wir nicht davon ausgehend, dass die erste Query direkt passt und eine ideale Pfadlänge bietet. Es kann passieren, muss aber nicht. Erhöhen wir jetzt die Anzahl der generierten Querys pro Pfad, so erhöhen wir auch die Wahrscheinlichkeit, dass zumindest eine Query die ideale Pfadlänge für diesen Pfad erreicht. Hierfür müssen wir *Schritt 7.* aus 6.4 so oft wie gewünscht wiederholen, sodass wir verschiedene Querys mit verschiedenen Argumenten erhalten. Bei dieser Methode ist nämlich wichtig, dass sich die Argumente der Querys verändern da wir sonst einfach mehrfach die selbe Query stellen mit der gleichen zu erwartenden Antwort.

6.5.2 Falsch-Negative Tests

Responses mit dem HTTP-StatusCode 4XX sind zu sehr hoher Wahrscheinlichkeit Fehler in der Query. Fehler in der Query können passieren wenn falsche Argument-Typen angegeben werden, ein Syntaxfehler auftritt oder zwingende Argumente fehlen. Eine konkrete Implementierung sollte beachten, dass diese Fehler vorkommen können und sie explizit als Falsch-Negativen Tests werten dann hierbei hat das SUT nichts falsch gemacht sondern die Implementierung des Testtools ist falsch. Einige GraphQL-APIs reagieren mit einem 400er Fehlercode auf nicht gefundene Einträge. Dies ist jedoch abhängig vom SUT und muss gesondert betrachtet werden.

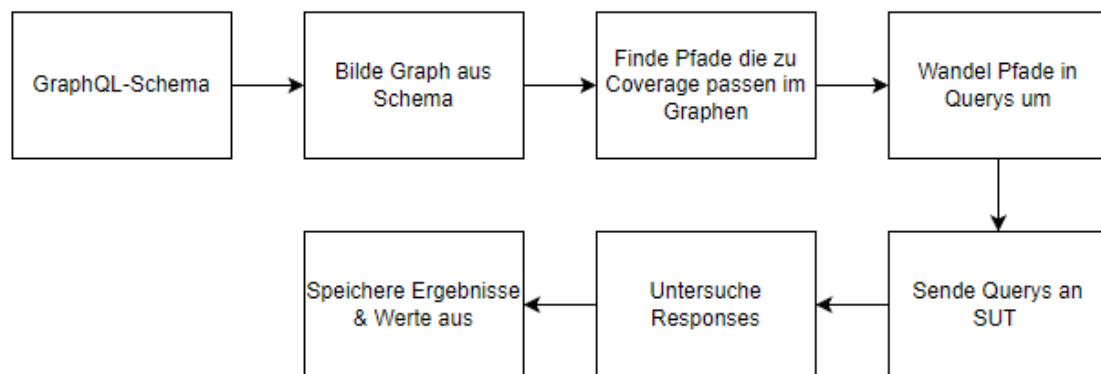
6.5.3 Negative Tests

Sollte eine Response den StatusCode 5XX erhalten, so haben wir einen tatsächlichen Fehler in der Programmierung gefunden da der Statuscode auf einen Internen ServerFehler hinweist. Ein Internet Serverfehler entspricht einer falschen Programmierung also einem Fehler den unsere Methode gefunden hat. Typischerweise liefert eine GraphQL-API den exakten Fehler mit. Aus dieser Fehlermeldung kann dann der Entwickler Schlüsse ziehen wieso dieser Fehler auftrat. Sollte eine GraphQL-API diesem Testsystem standhalten

und keine Response mit 500er zurückliefern, so bedeutet dies nicht, dass die GraphQL-API keinen Fehler enthält. Man kann also nicht davon ausgehen, dass diese Methode alle Fehler findet die in einer GraphQL-API existieren. Dies liegt einerseits daran, dass nicht jede Kante getestet wird. Andererseits auch daran, dass die Response nicht überprüft wird auf ihre tatsächlichen zurückgegebenen Daten. So ist es denkbar, dass Tests zwar positiv sind da sie Daten mit StatusCode 200 zurückliefern, diese allerdings nicht den erwarteten Daten entsprechen. Hierzu sollen alle generierten Tests sowie ihre Response abgespeichert werden damit diese von einem Entwickler manuell noch untersucht werden können.

6.6 Zusammenfassung der Methode und struktureller Vergleich mit Property-based Testing

Wir wollen im folgenden die eben vorgestellte Methode noch einmal kurz zusammenfassen damit diese übersichtlicher wird. Unsere hier vorgestellte Methode funktioniert grob gesehen so:



Wie zu sehen, ist der ganz grobe Ablauf ähnlich zum [4, Property-based Testing] allerdings unterscheidet sich die Methode in einigen Teilen sehr stark vom [4, Property-based Testing]. Wir fügen in unserer Methode den Schritt hinzu, dass wir einen Graphen erstellen welcher die Knoten und Kanten des GraphQL-Schemas repräsentiert während in [4, Property-based Testing] ausgehend vom Query-Type zufällig bis zu einer bestimmten Pfadlänge (dem Rekursionslimit) die Pfade gebildet werden indem immer zufällig Felder hinzugefügt werden. Durch unsere Methode erreichen wir, dass Pfade jeder Länge, die durchaus länger sein können als ein definiertes Rekursionslimit, abgedeckt werden und somit die Tests eine bessere Coverage erreichen können. Unser Ansatz erlaubt es außerdem, verschiedene Coverage-Kriterien zu implementieren. So ist man nicht gezwungen auf einer Methode zu verharren sondern kann je nach Implementierung die Pfadgenerierung anpassen nach den individuellen Anforderungen ohne, dass in anderen Schritten etwas geändert werden muss. Bei der Umwandlung der Pfade in Querys unterscheidet

sich unser Ansatz ein wenig von [4, Property-based Testing]. In unserer Methode generieren wir aus dem Pfad direkt die Query und generieren die nötigen Argumente "on-the-fly" während sie erkannt werden. Im Property-based Ansatz wird ein Datenobjekt als ganzes erstellt, dass die Query später generieren kann. In der technischen Umsetzung unterscheiden sich beide Methoden, im Ergebnis bekommen Sie jedoch strukturell gleiche Querys. Die Ausführung der Tests hingegen unterscheidet sich überhaupt nicht mehr zum [4, Property-based Testing]. Einzig und allein unsere Einführung der erwarteten vs. tatsächlichen Pfadlänge ist ein neuer Ansatz der die Qualität der zu testenden Querys messbar macht - dies fehlt im [4, Property-based Testing]. Dort ist man im unklaren darüber wie gut die Tests genau getestet haben und ob die erwartete Coverage auch mit der tatsächlichen Übereinstimmt.

Wir haben nun unsere Methode im groben vorgestellt und Unterschiede zum schon bestehenden Ansatz [4, Property-based Testing] erörtert. Im folgenden wollen wir uns der praktischen Umsetzung dieser Methode widmen und einen Prototypen entwickeln. Dieser Prototyp soll dann gegen das [4, Property-based Testing Tool] antreten und möglichst zeigen, dass die eben entwickelte Methode eine Verbesserung darstellt.

7 Praxis

Nach der Einführung der Methode im vorherigen Kapitel soll nun der entwickelte Prototyp umfassend erklärt werden. Der entwickelte Prototyp lässt sich unter finden und testen. Eine erklärende Readme existiert im Root-Verzeichnis. Voraussetzungen zum Ausführen der Anwendung ist Python und diverse Dritt-Bibliotheken die in der Readme vermerkt sind.

7.1 Tool- / Dependencyauswahl

Um die vorgestellte Methode umzusetzen war insbesondere wichtig, dass eine einfache und mächtige Bibliothek für die Definition und Bearbeitung von Graphen zur Verfügung steht. Die erste Wahl fiel hierbei auf NetworkX, eine Graphenbibliothek in Python. Sie wurde ausgewählt da der Ersteller schon einige Erfahrungen mit dieser Bibliothek hat und somit eine effiziente Umsetzung möglich war. Dadurch, dass diese Bibliothek als Grundlage gewählt wurde hat sich die Programmiersprache Python schnell ergeben. Im folgenden werden einige weitere benutzte Bibliotheken kurz vorgestellt sodass der Applikationsstack übersichtlich wird. Wir werden auch auf NetworkX und seine Features eingehen. Es werden nicht alle Bibliotheken eine Berücksichtigung finden sondern nur diese, die einen signifikanten Einfluss auf das Programm haben und besonders herausstechen.

7.1.1 NetworkX

NetworkX ist eine Python-Bibliothek für *Erstellung, Manipulation und Untersuchung der Struktur, Dynamik und Funktionen komplexer Netzwerke* [14, vgl. Startseite] Mit einer Star-Anzahl von 12.8k[15] auf GitHub ist networkX eine sehr beliebte Bibliothek. NetworkX ist die ideale Wahl um Graphen zu erstellen für unseren Use-Case denn es nimmt jeden möglichen Datentypen als Wert für einen Knoten und Kante. Wir können also sehr simpel Graphen definieren. Für das simple Beispiel von Author, Book, Publisher und deren Verbindungen benötigen wir nur folgende Zeilen:

```
1 import networkx as nx
2
3 G = nx.Graph()
4 G.add_edge("Query", "Book", "book")
5 G.add_edge("Query", "Author", "author")
6 G.add_edge("Query", "Publisher", "publisher")
7
```

```

8      G.add_edge("Publisher", "Book", "book")
9      G.add_edge("Book", "Publisher", "publisher")
10
11     G.add_edge("Book", "Author", "author")
12     G.add_edge("Author", "Book", "book")

```

Diese wenigen Zeilen reichen aus um unseren Graphen mit allen Knoten und Kanten zu definieren. Wie zuvor eingeführt existiert auch das Kantengewicht, dass der Feldbezeichner eines Types ist. Auf diesem Graphen können wir dann diverse Algorithmen ablaufen lassen. Diverse Hilfsfunktionen helfen dabei eine effiziente Programmierung zu erlangen. Hierbei seien insbesondere folgende Hilfsfunktionen genannt:

draw

```

1      nx.draw(G, with_labels=True)

```

Zeichnet einen den erstellten Graphen in ein beliebiges Format. So fällt es einfach große Graphen darzustellen.

shortest_path

```

1      shortest_path = nx.shortest_path(G, Node1, Node5)

```

Die Funktion *shortest_path* gibt eine Liste von Kanten zurück, die den kürzesten Weg zwischen zwei Knoten angibt.

neighbors

```

1      G.neighbors(Node)

```

Diese Funktion liefert alle Nachbarn eines Knotens. In unserem Kontext eine sehr wichtige Funktion wie wir später noch sehen werden.

7.1.2 Faker

Die gewählte Datengenerierungsbibliothek ist *Faker*[16]. Mit *16k*[16] Sternen auf GitHub ist Faker eine noch beliebtere Bibliothek als NetworkX. Faker ist eine Bibliothek die es sehr einfach macht Daten zu generieren. Da wir im Kontext von GraphQL Argumenten nur sehr einfache Datentypen als Argumente benötigen reicht uns diese Bibliothek komplett aus da sie es schafft uns schnell und unkompliziert Daten in genau dem Format zu generieren wie wir sie brauchen. Angenommen wir benötigen einen String der 10 Zeichen lang ist, so reicht eine Zeile:

```

1      random_string = fake.pystr(min_chars=10, max_chars
                                =10)

```

Selbiges falls wir eine Zufallszahl benötigen zwischen 1 bis 1000

```
1 random_number = fake.random_int(min=1, max=1000)
```

Diese Schema des Einzelers zieht sich für alle simplen *SCALAR* Types in GraphQL. Daher fällt die Wahl für die Datengenerierung auf diese Bibliothek.

7.1.3 PyTest

Nicht zwingend notwendig ist ein Testframework. Allerdings soll unsere Implementation der Methode Tests erstellen sodass diese zu einem späteren Zeitpunkt erneut ausgeführt werden können. Somit können wir z.B. überprüfen ob eine Korrektur des Servercodes eine Verbesserung gebracht hat. Hierfür wollen wir die Tests mithilfe eines Testframeworks erstellen. Die Wahl hierfür fiel dabei auf PyTest. PyTest ist ein Testframework für Python welches eine simple und einfache Testdefinition ermöglicht. Ein Test für eine einfache Funktion *inc* kann mit *test.inc* umgesetzt werden.

```
1 def inc(x):  
2     return x + 1  
3  
4 def test_inc():  
5     assert inc(3) == 5
```

Dies reicht schon vollkommen aus für unsere Testentwicklung daher wurde sich für diese Bibliothek entschieden.

7.2 Umsetzung der Methode

Für die Umsetzung der Methode werden wir durch die einzelnen Teile des Codes gehen und die jeweiligen Stellen erklären die einzelne Schritte der entwickelten Methode durchgehen. Hierbei gehen wir chronologisch in den einzelnen Schritten vor so wie in der Methode definiert.

7.2.1 Schema in Graph abbilden

Wie in der Vorstellung der Methode unter 6.1 bilden wir das GraphQL-Schema in einem Graphen ab. Hierfür nutzen wir die zuvor erwähnt Python Graphbibliothek NetworkX. Um die Informationen zu erlangen die für die Bildung des Graphens wichtig sind führen wir die Introspection-Query 12 aus. Das Ergebnis ist dann das vollständige GraphQL-Schema der API. Hierbei sei angemerkt, dass einige GraphQL APIs so eine Introspection-Query verbieten, sei es einerseits durch direktes verbieten oder ein Tiefenlimit in den Querys. Egal was hierbei der Fall ist, die zu testende API muss unsere Introspection-Query 12 unterstützen. Die Query wird mit einem simplen HTTP-POST an die zu testende URL gesendet.


```

1     r = requests.post(testUrl, json={'query': queries.
      introspection_query})
2     json_data = json.loads(r.text)
3     with open('schema.json', 'w') as f:
4         json.dump(json_data, f)

```

Und die Response wird als JSON-Objekt in *json_data* gespeichert. Zudem speichern wir das JSON-Objekt der aktuellen API auch ab damit diese später gegebenenfalls untersucht werden kann. Es wurde ein Modul *Graphhandler* entwickelt, dass verschiedene Graphoperationen für einen übernimmt. Im Graphhandler ist eine Funktion *buildGraph* definiert. Diese generiert einen Graphen von einem gegebenen Startknoten, einem leeren Graphen und dem Schema. Hierbei werden nur erreichbare Knoten von Startknoten berücksichtigt. Setzt man den Startknoten auf den Knoten *Query* so inkludieren wir auf diese Weise nur alle erreichbaren Teile des Graphens ausgehend von *Query*. Dies ist insofern sinnvoll da andere Typen, wenn sie nicht von *Query* aus erreichbar sind, nicht Teil des Testraumes wären da diese in keiner validen Anfrage vorkommen können. Die Funktion, die den Graphen generiert sieht wie folgt aus:

```

1 def buildGraph(graph, type_name, type_dict):
2     if type_name.startswith(nonSchemaTypePrefix) or
      type_name in baseDatatypes:
3         pass
4     else:
5         for adjacentNode in type_dict[type_name]['fields']:
6             if graph.has_edge(type_name, adjacentNode['type',
              ]['name']):
7                 return
8             else:
9                 if adjacentNode['type']['name'] and
              adjacentNode['type']['name'] not in
              baseDatatypes:
10                    graph.add_edge(type_name, adjacentNode['
              type']['name'])
11                    graph[type_name][adjacentNode['type']['
              name']]["data"] = adjacentNode
12                    buildGraph(graph, adjacentNode['type']['
              name'], type_dict)
13                if adjacentNode['type']['kind'] == 'LIST'
              and adjacentNode['type']['ofType']['name'
              ] not in baseDatatypes:
14                    graph.add_edge(type_name, adjacentNode['
              type']['ofType']['name'])
15                    graph[type_name][adjacentNode['type']['
              ofType']['name']]["data"] =

```

16

```

        adjacentNode
    buildGraph(graph, adjacentNode['type']['
        ofType']['name'], type_dict)

```

Die Funktion `buildGraph` arbeitet rekursiv. Vom Startknoten (im Allgemeinen *Query*) aus rufen wir die Funktion auf allen Folgeknoten von *Query* auf. Dies sind alle Knoten die den Type *OBJECT* besitzen und nicht mit einem `--` beginnen oder ein Basisdatentyp sind. GraphQL kann eigene Objekte definieren welche mit `--` starten, diese schließen wir explizit aus genau wie alle *SCALAR* Types. Jeder Knoten definiert nun in seinem *fields* Eintrag zu welchen Feldern er Beziehungen hat. Hierbei muss unterschieden werden, dass ein Eintrag entweder vom Type *OBJECT* oder *LIST* sein kann, um zulässig zu sein. Sollte es sich um einen *LIST* Eintrag handeln müssen wir prüfen von welchem Type die *LIST* ist. Wenn ein Knoten nun unsere Bedingungen erfüllt, so wird dieser dem Graphen hinzugefügt und auf ihm selbst wird `buildGraph` ausgeführt. So erlangen wir die gesamte Graphstruktur da ausgehend von *Query* jeder erreichbare Knoten hinzugefügt wird und dann von diesem Knoten eben wieder alle erreichbaren Knoten hinzugefügt werden.

7.2.2 Pfade aus Graph bilden

Der Graphhandler implementiert verschiedene Coveragekriterien. Das Tool benötigt eine Liste *paths* die aus Kanten besteht.

```

1 paths = graphhandler.generate_prime_paths("Query", graph)

```

Eine Funktion `generate_CoverPaths` kann jedes erdenkliche Coveragekriterium umsetzen. In unserem Fall sind es insbesondere PrimePaths, umgesetzt durch die Funktion `generate_prime_paths("Query", graph)`. Diese Funktion ermittelt die PrimePaths ausgehend vom Startknoten *Query* im definierten Graphen. Will man ein anderes Coveragekriterium umsetzen, so muss die Funktion `generate_CoverPaths` eine Liste der Kanten zurückgeben die dem Coveragekriterium entsprechend den Graphen überdecken. Die Generierung der PrimePaths sehen wir uns nun noch genauer an. Wir starten mit der Funktion `generate_prime_paths()`. Diese bekommt einen Startknoten und einen Graphen. Sie gibt dann die Liste der Pfade genau wie zuvor spezifiziert zurück.

```

1 def generate_prime_paths(startknoten, g):
2     pfaadliste = []
3     generate_paths(startknoten, [], pfaadliste, g)
4     return pfaadliste

```

Es wird die Funktion `generate_paths` aufgerufen wobei `generate_paths` sich immer wieder rekursiv selbst aufruft wenn die Nachfolger des Knotens noch nicht im Pfad sind. So werden Kreise verhindert, alle Pfade die so erzeugt werden sind SimplePaths.

```

1 def generate_paths(n, path, pathList, g):
2     path.append(n)
3     for m in g.successors(n): # successors gibt die
        Nachfolger wieder

```

```

4         if m not in path:
5             generate_paths(m, copy.deepcopy(path), pathList,
6                             g)
7         if is_prime_path(path, pathList):
            pathList.append(path)

```

Sollte ein SimplePath dann ein PrimePath sein so wird dieser zurückgegeben. Ein Pfad ist ein Prime-Pfad, wenn er kein Teil eines bereits existierenden Pfades ist und keine Kreise enthält. Diese drei hier vorgestellten Funktionen setzen den Pseudocode aus 6.3.2 um

```

1 def is_prime_path(new_path, pathList):
2     for exisiting_paths in pathList:
3         if is_subpath(new_path, exisiting_paths):
4             return False
5     return True

```

7.2.3 Coveragepfade ermitteln (maybe remove - TODO)

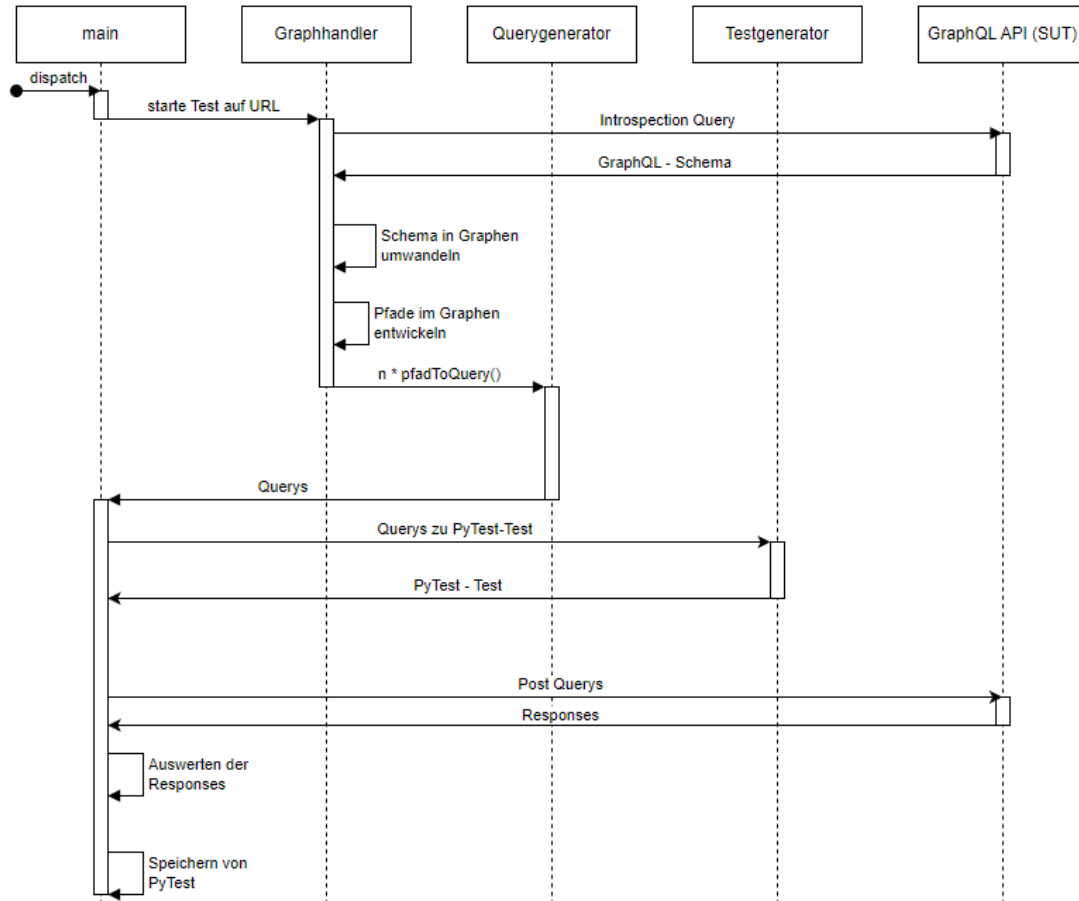
7.2.4 Querys aus Pfad ermitteln

7.2.5 Test ausführen / Testfile generieren

7.2.6 Testauswertung

7.3 Zusammenfassung der Implementation

Die entwickelte Methode lässt sich in folgendem Sequenzdiagramm gut abbilden:



8 Auswertung / Experiment / Vergleich mit Property-based Testing

8.1 Vergleichsmetriken

Bevor wir einen tatsächlichen Vergleich beider Methoden durchführen werden erst einmal die Metriken eingeführt in denen sich Verglichen wird. Hierdurch wird einfacher verständlich welche Punkte miteinander verglichen werden. Wir werden einige neue Metriken einführen aber auch Metriken nutzen die in *Property-based Testing*[4] genutzt wurden.

8.1.1 Metriken aus Property-based Testing

In *Property-based Testing* wurden zwei Metriken eingeführt, um die Methode zu evaluieren. Hierbei wurden zwei Forschungsfragen entwickelt.

1. Welche Schema Coverage kann mit der Methode erreicht werden? [4, vgl. RQ1]
2. Wie gut ist die Fehlerfindungskapazität der Methode? [4, vgl. RG2]

Forschungsfragen aus Property-based Testing

Zur Auswertung der Methode wurden zwei Testsysteme genutzt. Das erste Testsystem ist eine eigens entwickelte GraphQL-API die bekannte Fehler besitzt [4, vgl. A.1]. Testsystem 2 ist GitLab. Ein häufig genutztes Tool für GitServer mit DevOps Kapazitäten. Gitlab bietet seine API auch als GraphQL an und durch seine riesige Größe eignet sich GitLab als solides Testsystem. [4, vgl. A2] Unser entwickelter Prototyp soll in exakt dem gleichen Umfeld seine Tests generieren. Wir erwarten, dass wir möglichst die selben Fehler finden wie die ursprüngliche Methode und positiv wäre, wenn wir mehr und neue Fehler finden würden. Beide Forschungsfragen werden im folgenden noch einmal näher erläutert da diese ein wenig spezialisiert sind und Wissen über Methode ist wichtig um die Ergebnisse korrekt einordnen zu können.

8.1.2 Fehlerfindungskapazitäten

Mit Fehlerfindungskapazitäten ist gemeint wie zuverlässig die Methode tatsächliche Fehler findet. Hierfür werden die beiden zuvor benannten APIs getestet und es wird geprüft ob die Methode die Fehler finden konnte. Um zu verifizieren, dass die Methode möglichst viele Fehler findet gibt es eine Test API die initial mit bekannten Fehlern versehen wird. Die *Property-based Methode* hat 11 von 15 Fehlern im speziell vorbereiteten System

gefunden. Bei GitLab wurden 15 bugs gefunden. Unsere entwickelte Methode soll mindestens die gleichen Fehler finden und idealerweise mehr.

Schema Coverage / Schema (Ab)Überdeckung

Dadurch, dass *Property-based Testing* auf zufallsbasierter Testgenerierung basiert stellt sich hier die Frage, wie gut die Methode die API abdeckt und inwiefern die generierten Tests ausreichend sind. Dies kommt insbesondere zu tragen, wenn die maximale Pfadlänge ausgehend vom Query Knoten größer ist als die erlaubte Rekursionstiefe des Prototypens. In Property-based Testing wird definiert, dass die generierten Tests eine Full-Schema Coverage erreichen, wenn gilt:

Definition 1 *Für alle Objekte des Schemas: Bilde alle Tupel $\{Object, Field\}$. Ein Schema hat eine ideale Coverage wenn alle Tupel durch einen Test abgedeckt sind.*

Wie in Property-based Testing schon erwähnt: *da keine Coverage Metric für GraphQL Blackbox Test Auswertung existiert, starten wir mit einem sehr einfachen und intuitiven Ansatz* [4, vgl. B. Measuring Schema Coverage]. In der Tat ist das vorgestellte Coveragekriterium ein sehr einfaches Kriterium. Es lässt zum Beispiel die Beziehungen zwischen allen Knoten aus und beachtet nur, dass alle Knoten inbegriffen sind mit allen Feldern. Hiermit entspricht das definierte Coverage-Kriterium einer Kombination aus Edge- und Nodecoverage. Denn alle Knoten müssen abgedeckt sein und alle Kanten ausgehend von den Knoten. Wie zuvor gesehen ist ein solches Kriterium allerdings noch nicht ausreichend für eine ideale Testabdeckung, da zum Beispiel die verschiedenen Kantenkombinationen außer acht gelassen werden und sich somit doch noch Fehler im Code befinden können. Wesentlicher Unterschied beider Methoden ist insbesondere, dass *Property-based Testing* überprüfen muss ob es diese Coverage erreicht. Unsere vorgestellte Methode stellt sicher, dass diese Coverage erreicht ist bevor sie aufhört mit dem generieren. Die Überprüfung der Schema-Coverage in Property-based Testing geschah durch ausprobieren. Hierbei wurde ausprobiert wie viele Testgenerierungen benötigt wurden, um das definierte Kriterium zu erfüllen. Um ein 100% Coverage beim GitLab Schema zu erreichen waren verschiedene Anzahlen an Iterationen nötig bei verschiedenen Rekursionslimits. Eine 100% Coverage wurde bei GitLab nur erreicht wenn 10000 Tests mit Rekursionslimit 4 erstellt wurden. Die Berechnungszeit war hierbei 931 Sekunden. Dies ist zwar der schlechteste Wert in der gesamten Statistik und man könnte meinen, dass der Vergleich nun nicht genau wäre - allerdings ist dies auch der einzige Wert der verlässlich 100% Coverage geliefert hat. Unser Ziel ist es also, weniger als 10.000 Tests und 930 Sekunden zu benötigen um das hier definierte Coveragekriterium zu erfüllen.

8.1.3 Neue Metrik

Näheres betrachten des Codes von *Property-based Testing* offenbarte einen signifikanten Fehler in der Definition der Schema Coverage. Hierbei ist besonders wichtig zu wissen, wie GraphQL unter der Haube funktioniert. GraphQL verarbeitet die Schritte einer

Query sequentiell. Nutzen wir die zufällig generierte Query aus *Property based Testing* Fig. 9[4]. Die Query lautet:

1

```
{projects(id: "7x8Z"){description members{name}}}
```

Stellen wir nun diese Anfrage an die API und es existiert kein *Project* mit der id 7x8Z so hat die Funktion einen return Value von *null*. Ein Return Value von null bedeutet jedoch, dass GraphQL den Pfad nicht weiter auswerten wird und die Funktion des Resolvers hinter dem *members* Feld nicht ausgeführt wird. Diese Query würde jedoch die Coverage für die Tupel Project und Members erfüllen aus der vorigen Definition, ohne, dass diese wirklich getestet wurde. Laut Property-based Testing wird hierdurch angenommen, dass die Query erfolgreich ist wenn die Query erfolgreich ist. Allerdings haben wir nun ungetesteten Code der als getestet betrachtet wird. Wir wollen nun eine Metrik einführen die überprüft wieviel der zufällig generierten Querys tatsächlich komplett getestet haben. Hierfür wird folgende Metrik eingeführt:

Definition 2 Für alle Querys und dazugehörigen Responses wird die Pfadlänge bestimmt. Eine erfolgreiche Query hat dann zwei mögliche Szenarien:

1. $Pfadlänge(Query) = Pfadlänge(Response)$
2. $Pfadlänge(Query) \neq Pfadlänge(Response)$

Tritt Fall 1 ein so hat die Query wirklich alle Funktionen getestet. Tritt Fall 2 ein so hat die Query nicht alle Funktionen getestet. Zählt man nun alle Querys zusammen kann man auswerten zu wieviel Prozent die gesamte erwartete Pfadlänge tatsächlich ausgeführt wurde indem die $Pfadlänge(Response)$ hinzugezogen wird.

8.2 Threats to Validity / Limitierungen

Bevor wir mit dem eigentlichen Vergleich beginnen muss noch kurz eingeordnet werden inwiefern die Experimente zu betrachten sind und unter welchen Voraussetzungen der Vergleich geschieht.

8.2.1 Argumentgeneratoren

Wie in 8.1.3 angesprochen ist es wichtig, dass GraphQL für jede Funktionen einen Wert ungleich *null* bekommt, sodass der Pfad weitergegangen werden kann und die Funktionen in diesem getestet werden. Um Bedingungen zu begünstigen werden die Argumentgeneratoren für jedes Experiment angepasst, sodass es sehr viel wahrscheinlich ist, dass die generierten Argumente auch zum SUT passen und die allgemeine Query-Qualität hierdurch besser wird. Es hängt dann explizit davon ab wie sehr die Argumentgeneratoren angepasst werden denn ein einfaches anpassen hat sich *Property-based Testing*[4] auch erlaubt. Hierbei sei zum Beispiel erwähnt, dass eine Type *ID* in GraphQL als String wert definiert ist, häufig in Implementierung jedoch als Zahlenstring genutzt wird. Eine beispielhafte Anpassung wäre hier nun, dass wir den Generator für den Type *ID* so

anpassen, dass er nur Argumente für ID zurückliefert die ein Zahlenwert sind in einem gewissen Bereich der durch das SUT abgebildet wird.

9 zukünftige Arbeit

Hisdfsdf

10 Fazit

Hi

11 Glossar

Im Text werden einige Fachbegriffe genutzt. Hier findet sich deren Erklärung

Begriff Erklärung

IEEE/ACM

HTTP

HTTP-Request

API

REST

GraphQL

Overfetch

Underfetch

Evolutionärer Algorithmus

SUT

IoT

12 Anhang

Introspection-Query

```
1      query IntrospectionQuery {
2  --schema {
3      queryType {
4          name
5      }
6      mutationType {
7          name
8      }
9      subscriptionType {
10         name
11     }
12     types {
13         ...FullType
14     }
15     directives {
16         name
17         description
18         locations
19         args {
20             ...InputValue
21         }
22     }
23 }
24
25
26 fragment FullType on __Type {
27     kind
28     name
29     description
30     fields(includeDeprecated: true) {
31         name
32         description
33         args {
34             ...InputValue
```

```

35     }
36     type {
37         ...TypeRef
38     }
39     isDeprecated
40     deprecationReason
41 }
42 inputFields {
43     ...InputValue
44 }
45 interfaces {
46     ...TypeRef
47 }
48 enumValues(includeDeprecated: true) {
49     name
50     description
51     isDeprecated
52     deprecationReason
53 }
54 possibleTypes {
55     ...TypeRef
56 }
57 }
58
59 fragment InputValue on __InputValue {
60     name
61     description
62     type {
63         ...TypeRef
64     }
65     defaultValue
66 }
67
68 fragment TypeRef on __Type {
69     kind
70     name
71     ofType {
72         kind
73         name
74         ofType {
75             kind
76             name
77             ofType {
78                 kind

```

```

79         name
80       ofType {
81         kind
82         name
83         ofType {
84           kind
85           name
86           ofType {
87             kind
88             name
89             ofType {
90               kind
91               name
92             }
93           }
94         }
95       }
96     }
97   }
98 }
99 }

```

minimale Schema Response

```

1  {
2    "data": {
3      "__schema": {
4        "queryType": {
5          "name": "Query"
6        },
7        "mutationType": null,
8        "subscriptionType": null,
9        "types": [
10         {
11           "kind": "OBJECT",
12           "name": "Query",
13           "description": null,
14           "fields": [
15             {
16               "name": "book",
17               "description": null,
18               "args": [

```

```

19         {
20             "name": "id",
21             "description": null,
22             "type": {
23                 "kind": "SCALAR",
24                 "name": "ID",
25                 "ofType": null
26             },
27             "defaultValue": null
28         }
29     ],
30     "type": {
31         "kind": "OBJECT",
32         "name": "Book",
33         "ofType": null
34     },
35     "isDeprecated": false,
36     "deprecationReason": null
37 },
38 {
39     "name": "author",
40     "description": null,
41     "args": [
42         {
43             "name": "id",
44             "description": null,
45             "type": {
46                 "kind": "SCALAR",
47                 "name": "ID",
48                 "ofType": null
49             },
50             "defaultValue": null
51         }
52     ],
53     "type": {
54         "kind": "OBJECT",
55         "name": "Author",
56         "ofType": null
57     },
58     "isDeprecated": false,
59     "deprecationReason": null
60 },
61 {
62     "name": "publisher",

```

```

63         "description": null,
64         "args": [
65             {
66                 "name": "id",
67                 "description": null,
68                 "type": {
69                     "kind": "SCALAR",
70                     "name": "ID",
71                     "ofType": null
72                 },
73                 "defaultValue": null
74             }
75         ],
76         "type": {
77             "kind": "OBJECT",
78             "name": "Publisher",
79             "ofType": null
80         },
81         "isDeprecated": false,
82         "deprecationReason": null
83     }
84 ],
85     "inputFields": null,
86     "interfaces": [],
87     "enumValues": null,
88     "possibleTypes": null
89 },
90 {
91     "kind": "OBJECT",
92     "name": "Book",
93     "description": null,
94     "fields": [
95         {
96             "name": "id",
97             "description": null,
98             "args": [],
99             "type": {
100                 "kind": "SCALAR",
101                 "name": "ID",
102                 "ofType": null
103             },
104             "isDeprecated": false,
105             "deprecationReason": null
106         },

```



```

107         {
108             "name": "title",
109             "description": null,
110             "args": [],
111             "type": {
112                 "kind": "SCALAR",
113                 "name": "String",
114                 "ofType": null
115             },
116             "isDeprecated": false,
117             "deprecationReason": null
118         },
119         {
120             "name": "author",
121             "description": null,
122             "args": [],
123             "type": {
124                 "kind": "OBJECT",
125                 "name": "Author",
126                 "ofType": null
127             },
128             "isDeprecated": false,
129             "deprecationReason": null
130         },
131         {
132             "name": "publisher",
133             "description": null,
134             "args": [],
135             "type": {
136                 "kind": "OBJECT",
137                 "name": "Publisher",
138                 "ofType": null
139             },
140             "isDeprecated": false,
141             "deprecationReason": null
142         }
143     ],
144     "inputFields": null,
145     "interfaces": [],
146     "enumValues": null,
147     "possibleTypes": null
148 },
149 {
150     "kind": "OBJECT",

```

```

151     "name": "Author",
152     "description": null,
153     "fields": [
154         {
155             "name": "id",
156             "description": null,
157             "args": [],
158             "type": {
159                 "kind": "SCALAR",
160                 "name": "ID",
161                 "ofType": null
162             },
163             "isDeprecated": false,
164             "deprecationReason": null
165         },
166         {
167             "name": "name",
168             "description": null,
169             "args": [],
170             "type": {
171                 "kind": "SCALAR",
172                 "name": "String",
173                 "ofType": null
174             },
175             "isDeprecated": false,
176             "deprecationReason": null
177         },
178         {
179             "name": "books",
180             "description": null,
181             "args": [],
182             "type": {
183                 "kind": "LIST",
184                 "name": null,
185                 "ofType": {
186                     "kind": "OBJECT",
187                     "name": "Book",
188                     "ofType": null
189                 }
190             },
191             "isDeprecated": false,
192             "deprecationReason": null
193         }
194     ],

```

```

195     "inputFields": null,
196     "interfaces": [],
197     "enumValues": null,
198     "possibleTypes": null
199   },
200   {
201     "kind": "OBJECT",
202     "name": "Publisher",
203     "description": null,
204     "fields": [
205       {
206         "name": "id",
207         "description": null,
208         "args": [],
209         "type": {
210           "kind": "SCALAR",
211           "name": "ID",
212           "ofType": null
213         },
214         "isDeprecated": false,
215         "deprecationReason": null
216       },
217       {
218         "name": "name",
219         "description": null,
220         "args": [],
221         "type": {
222           "kind": "SCALAR",
223           "name": "String",
224           "ofType": null
225         },
226         "isDeprecated": false,
227         "deprecationReason": null
228       },
229       {
230         "name": "books",
231         "description": null,
232         "args": [],
233         "type": {
234           "kind": "LIST",
235           "name": null,
236           "ofType": {
237             "kind": "OBJECT",
238             "name": "Book",

```

```
239         "ofType": null
240     }
241 },
242     "isDeprecated": false,
243     "deprecationReason": null
244 }
245 ],
246     "inputFields": null,
247     "interfaces": [],
248     "enumValues": null,
249     "possibleTypes": null
250 }
251 ]
252 ]
253 }
254 }
255 }
```

Literaturverzeichnis

- [1] *Evo Master*, <https://github.com/EMResearch/EvoMaster>, zuletzt besucht: 15.06.2023.
- [2] S. Karlsson, A. Causevic und D. Sundmark, *QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs*, 2019. arXiv: 1912.09686 [cs.SE].
- [3] *Rest Test Gen*, <https://github.com/SeUniVr/RestTestGen/>, zuletzt besucht: 15.06.2023.
- [4] D. S. Stefan Karlsson Adnan Causevic, “Automatic Property-based Testing of GraphQL APIs,” *International Conference on Automation of Software Test*, 2021.
- [5] R. Diestel, *Graphentheorie*. 2000, ISBN: 3-540-67656-2.
- [6] *GraphQL-Specification*, <https://spec.graphql.org/June2018/>, zuletzt besucht: 16.06.2023.
- [7] *GraphQL is the better REST*, <https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>, zuletzt besucht: 15.06.2023.
- [8] P. A. J. Offutt, *Introduction to Software Testing*. 2008, ISBN: 978-0-521-88038-1.
- [9] *State of GraphQL 2022 survey*, <https://blog.graphqleditor.com/state-of-graphql-2022>, zuletzt besucht: 15.06.2023.
- [10] *Serene - Clojure.Spec from GraphQL Schema*, <https://github.com/paren-com/serene>, zuletzt besucht: 15.06.2023.
- [11] *Clojure Spec - Data structure definition*, <https://clojure.org/guides/spec>, zuletzt besucht: 15.06.2023.
- [12] *Mali - Data Driven Specification Library for Clojure*, <https://github.com/metosin/malli>, zuletzt besucht: 15.06.2023.
- [13] A. Belhadi, M. Zhang und A. Arcuri, *White-Box and Black-Box Fuzzing for GraphQL APIs*, 2022. arXiv: 2209.05833 [cs.SE].
- [14] *NetworkX - Network Analysis in Python*, <https://networkx.org>, zuletzt besucht: 06.07.2023.
- [15] *NetworkX - Network Analysis in Python*, <https://github.com/networkx/networkx>, zuletzt besucht: 06.07.2023.
- [16] *Faker - Faker is a Python package that generates fake data for you.* <https://github.com/joke2k/faker>, zuletzt besucht: 06.07.2023.
- [17] *Explaining GraphQL Connections*, <https://www.apollographql.com/blog/graphql/explaining-graphql-connections/>, zuletzt besucht: 18.06.2023.

- [18] *Faker*, <https://faker.readthedocs.io/en/master/>, zuletzt besucht: 06.07.2023.
- [19] *Pytest: Helps you write better programs*, <https://docs.pytest.org/en/7.4.x/>, zuletzt besucht: 06.07.2023.

Onlineressourcen wurden am 9. Juli 2023 auf ihre Verfügbarkeit hin überprüft.