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

Masterarbeit

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 1: Prof. Dr. rer. nat. Leen Lambers
Betreuer 2: Prof. Dr. rer. nat. Gerd Wagner
Gutachter: M.Sc Lucas Sakizloglou

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

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 Overfechting/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 PProperty-basedTests automatisch getestet werden können. Property-based"bzeieht 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.

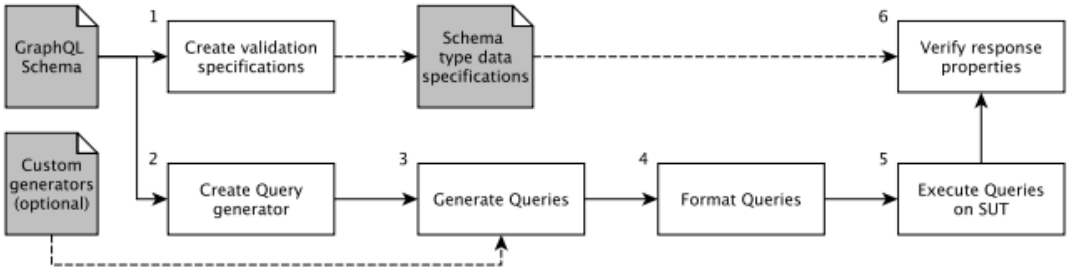
2 Einleitung

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] , QuickREST [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 umsetzten 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. Darauffolgend 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 Beweisführug 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 angewandet 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.

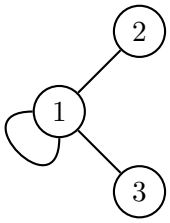
3.1 Graphentheorie

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

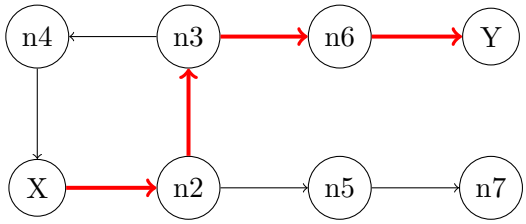
Eine gerichtete Kante besitzt im Gegensatz zu den ungerichteten Kanten eine Orientierung. Dies bedeutet, dass die Kante (x, y) einen Startknoten und einen Endknoten definieren muss. Gekennzeichnet wird dies für den Startknoten durch $init()$ und für den Endkonten durch $end()$. Eine gerichtete Kante 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 erlaubt 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 Eingansgrad 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 Knote sind.

Pfad / Weg

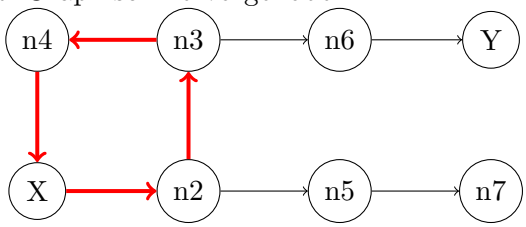
Ein Pfad, oft auch Weg genannt, ist eine Sequenz von Knoten die nachfolgend durch Kanten miteinander verbunden sind. [5, vgl. S. 7 0.3] Der Pfad von einem Knoten x zu einem Knoten y über Knoten n_1 bis n_7 hat dann zum Beispiel diese Struktur $p = \{(x, n2), (n2, n3), (n3, n6), (n6, y)\}$ wobei hierbei gilt, dass die Kanten $\{(x, n2), (n2, n3), (n3, n6), (n6, y)\} \in E$. 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 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 $= \{(X, e1), (e2, e3), ..., (e_n, X)\}$ 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 ungerichtete 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 Konkurenten 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 exisitieren. 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 bedeuet, 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 Authoren. 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 Einschänkungen 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 nich 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 müss 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 beeinhaltet 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
    createBok(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 Authors
    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 permantente 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 Rückgabetyp. 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 Resolvern 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
}
```

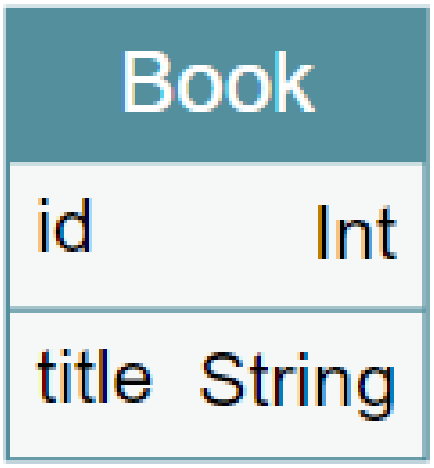
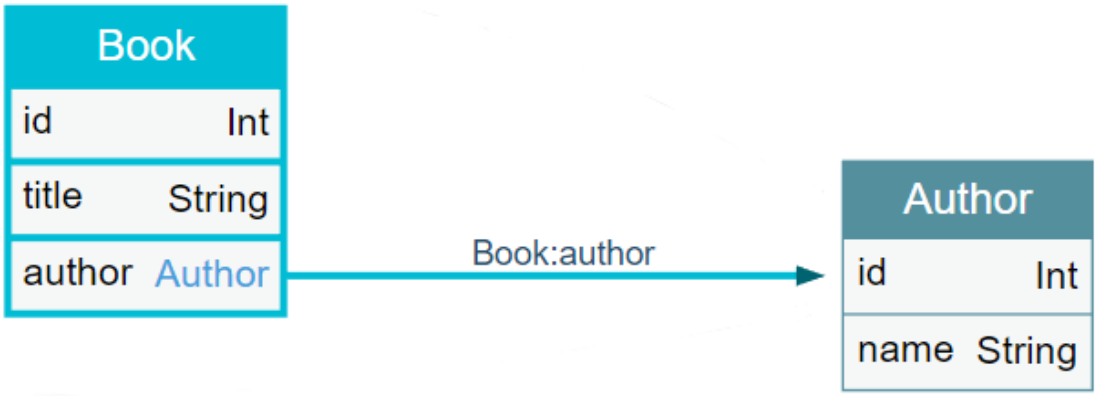


Abbildung 3.1: Book als Knoten

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 Autoren 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 Typen von Tests

Es gibt verschiedene Sichtweisen auf das zu testende System. Die Sichtweisen können intern oder extern bestimmt sein. So sind Faktoren wie möglicher Zugriff auf Quellcode oder Architekturdetails des Systems essentiell wichtig um zu entscheiden, welche Art von Tests angewandt werden. Es gibt zwei generelle Typen von Sichtweisen und eine Mischform. Wir betrachten hierbei das zu testende System als Box. Diese Box kann entweder transparent gesehen werden, halbtransparent oder untransparent. Der Transparente Ansatz nennt sich WhiteBox-Testing, der halbtransparente GreyBox-Testing und der Untransparente BlackBox-Testing. Die Ansätze unterscheiden sich vorallem in den zur Verfügung stehenden Informationen des Systems. Wir vernachlässigen hierbei das GreyBox-Testing da dieses keine Relevanz für das hier zugrunde liegende Problem besitzt.

White-Box Testing

Im WhiteBox-Testing stehen alle Informationen über das System zur Verfügung. Der Tester hat Zugriff auf Code, Architekturdetails und besitzt Kenntniss über alle möglichen Details des Systems. Somit kann der Tester auf alle möglichen Informationen über das System zugreifen und damit seine Tests generieren. Die erstellten Tests fundieren dann auf einem soliden Niveau, dass begründet wird durch das Domänenwissen über das System.

Black-Box Testing

Der Idealfall des Testens ist zwar der WhiteBox-Test allerdings ist es durch verschiedene Limitierungen nicht immer möglich oder auch gewollt, auf alle Informationen zuzugreifen. Im BlackBox-Testing hat der Tester keinen Zugriff auf interne Funktionsweisen der Software. Schwerpunkt des Testens ist es, dass die Software das tut was sie soll. Wir interessieren uns hier nicht dafür was intern im System abläuft sondern nur, ob Eingabe zu Ausgabe passt. In dieser Arbeit werden wir uns vor allem auf das BlackBox-Testing beziehen. Dies folgt einerseits aus dem methodischen bezug zu *Property-based Testing*[4] als auch aus dem Fakt, dass das Testtool unabhängig von Implementierungsdetails nutzbar sein soll.

3.4.2 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 etwaiigen 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 Granularitätseben 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 werden 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 getest 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.

3.4.3 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 Integern 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. Coveragekriteren 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, das 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 Divison 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 Grapcoverage 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 x N. Hierbei ist die Menge als initx x 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

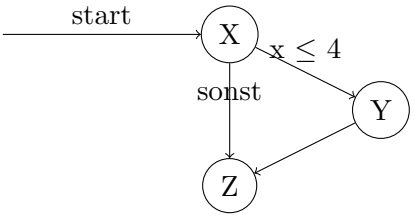
Null, der in einem Knoten N_0 startet und in einem Knoten N_f endet, nennt sich Testpfad [8, vgl. S. 28] Ziel ist es nun mit hilfe von Coverage-Kriterien Testpfade zu ermitteln die unser Problem ausreichend testen - was hierbei ausreichend ist, ist von Fall zu Fall unterschiedlich.

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

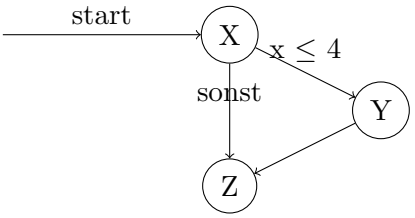
Node-Coverage ist ein Coveragekriterium, dass alle Knoten, die von N_0 erreichbar sind, in einem Graphen abdecken soll. Definieren wir folgenden, sehr einfachen Graphen:



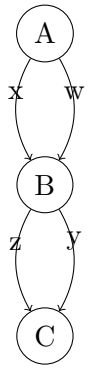
So wäre die Node-Coverage mit einem Test einzigen Test erfüllbar. Dieser ist der Pfad $X \to Y \to Z$. Es ist auch denkbar, dass wir zwei Pfade oder mehr nutzen allerdings erfüllt dieser Pfad schon unser Kriterium daher geben wir uns vorerst zufrieden. Wir sehen schnell, dass dieser Ansatz noch Lücken aufweist, da der Pfad $X \to Y \to Z$ das Kriterium erfüllt, allerdings wird eine Kante $X \to Z$ 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 $X \to Y \to Z$ und $X \to Z$ überdeckt. Edge-Coverage hat allerdings auch Probleme Graphen vollständig zu überdecken. Man nehme folgendes Beispiel:

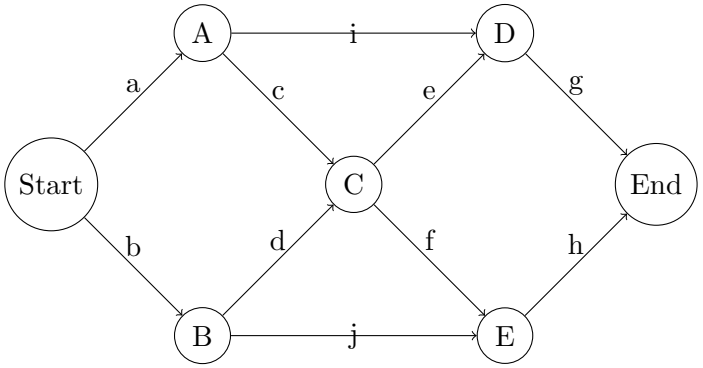


Pfade die laut Edge-Coverage ausreichen um den Graphen zu überdecken wären:
$x \to z$
$w \to y$
Hierbei wird allerdings außer acht gelassen, dass in x auch Änderungen passieren können die Auswirkungen im Programm haben können. So sind die Routen $x \to y$ und $w \to z$ in der Edge-Coverage nicht berücksichtigt. 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:
$x \to z$
$w \to y$
$x \to y$
$w \to z$

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.



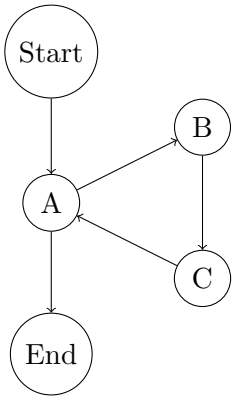
Nach der Definition von Edge-Pair Coverage ermitteln wir erstmal alle Pfadkombinationen der Länge "bis zu 2" Dies wären:
$Start \to A \to C$
$Start \to A \to D$
$Start \to B \to C$
$Start \to B \to E$
$A \to C \to D$
$A \to C \to E$
$B \to C \to D$
$B \to C \to E$
$A \to D \to End$
$B \to E \to End$
$C \to D \to End$
$C \to E \to End$

Hierdurch ergeben sich dann diese Testpfade:
$Start \to A \to C \to D \to End$
$Start \to A \to C \to E \to End$
$Start \to B \to C \to D \to End$
$Start \to B \to C \to E \to End$

allerdings fehlen auch hier wieder Pfade.
Zum Beispiel der Pfad $Start \to A \to D \to End$ 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:

$Start \to A \to End$
$Start \to A \to B \to C$
$A \to B \to C \to A$
$B \to C \to A \to B$
$C \to A \to B \to C$
$B \to C \to A \to End$

und schon zwei Testpfade würden ausreichen um die Prime-Path Coverage zu erfüllen. Die Testpfade die das Testrequirement erfüllen sind:
$Start \to A \to End$
$Start \to A \to B \to C \to A \to B \to C \to A \to End$

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

Wir haben nun verschiedene Coverage-Kriterien kennengelernt und teilweise schon Probleme benannt die einzelne Kriterien haben. Im Sinne einer zufriedenstellenden Testung eines Systems wollen wir nun die einzelnen Kriterien noch einmal zentral gegenüber stellen und sehen, welche Kriterien sich eignen würden für unseren, zu entwickelnden Prototypen. Unser Vergleich wird 4 verschiedene Kriterien beachten, diese sind Granu-

larität, Redundanz, Abdeckungspotential und Komplexität beziehungsweise Effizienz. Obwohl alle Überdeckungskriterien einen einzigartigen Blick auf den Graphen bieten, so lässt sich feststellen, dass einige Überdeckungskriterien geeigneter sind als andere.

Granularität Dieses Kriterium setzt seinen Fokus darauf, welche Teile des Graphens im Überdeckungskriterium Anwendung finden. Hierbei schauen wir insbesondere darauf, wie die Struktur des Graphens überdeckt wird.

Redundanz Damit wir nicht immer wieder die selben Tests ausführen vergleichen wir, wie stark die Redundanz innerhalb des Überdeckungskriteriums ist. Wir wollen möglichst effizient testen da der Testprozess bei großen Systemen durchaus einige Zeit in Anspruch nehmen kann.

Abdeckungspotential Wir untersuchen hier, wie stark das Potential des Kriteriums ist Tests zu generieren, die verlässlich Fehler finden. Hierbei ist vor allem wichtig, wie die Pfade strukturell aufgebaut sein werden die das Überdeckungskriterium erfüllen.

Komplexität / Effizienz Mit der Komplexität und Effizienz wird untersucht, wie viele Pfade generiert werden und wie gut der mögliche Testraum durch diese abgebildet werden kann.

Kriterium	Granularität	Redundanz	Abdeckungstiefe	Komplexität
Node	Knoten	Keine	Oberflächlich	Niedrigste, s
Edge	Kanten	Minimal	Ein bisschen tiefer	Mittlere, e
Edge-Pair	Kantenpaare	Überlappung	Tiefer	Erhöhte, e
SimplePath	Pfade ohne Wiederholungen	Keine	Ziemlich tief	Hohe, lar
PrimePath	Hauptpfade	Kann enthalten	Tief	Sehr hoch,
CompletePath	Alle möglichen Pfade	Höchste	Am tiefsten	Extrem hoch, s

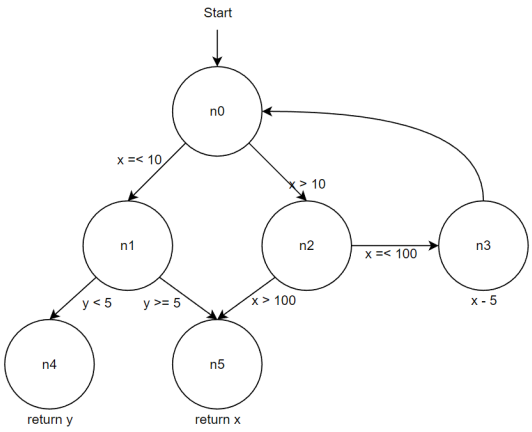
Tabelle 4.1: Vergleich der Graphabdeckungskriterien

4.3 Graphcoverage für Code

Im Kontext der Testentwicklung erlaubt die Graphcoverage es uns, einen systematischen Ansatz zur Testgenerierung zu verfolgen. Beziehen wir die Graphcoverage auf die Generierung von Tests für Code so müssen wir uns zuerst fragen, wie wir Code als Graphen darstellen können. Da dieser Abschnitt eher der generellen Einordnung dient, werden wir uns an dieser Stelle kurz halten. Im Allgemeinen muss der Code zuerst in einen Kontrollflussgraphen überführt werden. (Quelle) Ein Kontrollflussgraph ist ein gerichteter Graph mit, wie in 4.1 auch definiert, einer Menge Knoten, Anfangskonten und Endknoten sowie

Kanten. Wer mehr über die Umwandlung von Code in Kontrollflussgraphen lernen will, sei an [8, Kapitel 2.] verwiesen.

```
1  function example(x, y) {
2      if (x > 10) {
3          if(x > 100){
4              return x;
5          }
6          example(x - 5, y);
7      } else {
8          if (y < 5) {
9              return y;
10         }
11         return x;
12     }
13 }
```



Der Code der Funktion *example* sei nun zu testen. Mithilfe der zuvor definierten Coveragekriterien können wir Pfade nun Pfade ermitteln und aus diesen Pfaden tests generieren. Nehme wir zum Beispiel die Node-Coverage. So müssen wir Pfade finden, sodass jeder Knoten mindestens einmal in einem Test vorkommt. Mit NodeCoverage sind die Pfade $(N_0, N_2, N_3, N_1, N_5)$ und (N_0, N_1, N_4) ausreichend. Jeder Knoten wird durch einen Pfad repräsentiert. Erreicht werden können diese Pfade durch die Variablenauswahl x = 15, y = 10 für Pfad 1 und x = 5, y = 10 für Pfad 2. Somit haben wir die Tests example(15,10) und example(5,10) welche NodeCoverage erfüllen. Wir sehen auf anhieb, dass die Node-Coverage nicht ausreichend ist um Code gut zu testen. Es ist wünschenswert, dass jede Zeile Code im Testprozess mindestens einmal ausgeführt wird (Quelle TODO). Wir sehen aber, dass z.B. Zeile 4 nie ausgeführt wird durch unsere Tests. Da die Tests jedoch die Node-Coverage erfüllen müssen wir schlussfolgern, dass die Node-Coverage nicht ausreichend ist. Somit muss ein stärkeres Coverage-Kriterium her um Code ideal zu testen. Der interessierte Leser sei an dieser Stelle an *Introduction to Software Testing Kapitel 2.3.1* [8] verwiesen. Wir wollten hier lediglich zeigen, dass Überdeckungskriterien speziell für den Anwendungsfall auszuwählen sind.

4.4 Graphcoverage für GraphQL

Wie wir zuvor, in 3.3 feststellen konnten, lässt sich GraphQL in einen Graphen übersetzen. Aus dieser Tatsache folgt, dass wir Coveragekriterien auf diesem Graphen nutzen können. Im Unterschied zum Code ist die Zugrundeliegende Struktur direkt ein Graph und wir müssen keine großen Umwandlungen vornehmen. Alle Informationen über den Graphen sind in seinem Schema kodiert. Die zu ermittelnden Pfade ergeben damit auch direkt unsere Tests. Im folgenden wollen wir untersuchen, welches Coveragekriterium denn am geeignetsten wäre um es für Testgenerierung zu nutzen. Wir betrachten zuerst jedes Coveragekriterium für sich, betrachten seine Fähigkeiten aber auch Limitierungen. Abschließend ziehen wir ein Fazit, welches Coveragekriterium sich in unserem Kontext am

ehesten eignen würde für eine Testgenerierung. Wir starten vom grobgranularsten Kriterium und verfeinern die Granularität immer weiter.

4.4.1 Node-Coverage für GraphQL

Die Node-Coverage zielt darauf ab, dass jeder Knoten in mindestens einem Test Berücksichtigung findet. In GraphQL sind Knoten als Type definiert. Jeder Type definiert seine eigenen Resolver wobei dies im Endeffekt Funktionen sind die es zu Testen gilt. Nutzen wir nun Node-Coverage, so ignorieren wir unsere Maxime, dass wir möglichst alle Funktionen wenigstens einmal aufrufen wollen. So kann ein Type mehrere Felder definieren, ist dieser Type aber in einem Pfad nur einmal mit einer Funktion vertreten, so gilt er als ausreichend abgedeckt. Dadurch, dass die Kanten also eben so wichtig sind, ist Node-Coverage ein ungeeignetes Coveragekriterium für GraphQL Testgenerierung.

4.4.2 Edge-Coverage für GraphQL

Zuvor wurde deutlich, dass die Abdeckung aller Kanten essentiell ist um GraphQL gut zu testen. Mit der Edge-Coverage zielen wir genau darauf ab, dass jede Kante mit mindestens einem Test abgedeckt wird. Im Sinne unserer Maxime, dass jede Funktion zumindest einmal ausgeführt werden sollte, haben wir hier ein geeigneteres Kriterium gefunden. Die Edge-Coverage findet auch Anwendung in *Propert-based Testing*[4, vgl. D-RQ1] Hierbei muss gesagt werden, dass Edge-Coverage durchaus unseren Zweck erfüllt, dass jede Funktion einmal ausgeführt wird. Jedoch bezieht unser Kontext sich insbesondere auf das Integrationstesten. Wir wollen sicherstellen, dass die Funktionen miteinander ideal funktionieren. So ergibt sich, dass Edge-Coverage eine nicht zufriedenstellende Komplexität liefert da nur beachtet wird, dass alle Kanten einmal getest werden. Es wird nicht beachtet, dass auch die Hintereinanderreihung der Kanten funktionieren muss, da in GraphQL eine Funktion tiefer in der Query immer auf die Ergebnisse einer Funktion höher in der Query zurückgreift. Um zu validieren, dass diese Zusammenarbeit klappt, brauchen wir noch speziellere Überdeckungskriterien.

4.4.3 Edge-Pair-Coverage für GraphQL

In der Edge-Pair Coverage betrachten wir alle Kantenpaare. Dadurch erlangen wir eine bessere Abdeckung der Funktionen indem sichergestellt wird, dass jede Kante mit jeder darauffolgenden Kante einmal ausgeführt wird. Dieses Kriterium stellt eine Verbesserung der Edge-Coverage dar, allerdings werden eben nur aufeinanderfolgende Kantenpaare abgedeckt. GraphQL kann aber wesentliche tiefere und komplexere Strukturen abbilden. Somit ergibt sich, dass die Abdeckung mit Edge-Pair noch nicht ausreichend ist da insbesondere stark verschachtelte Anfragen hier nicht als Test generiert werden obwohl diese wahrscheinlich besonders interessant für Tests sind.

4.4.4 SimplePath-Coverage für GraphQL

SimplePath berücksichtigt alle Pfadkombinationen im Schema die keine Wiederholungen enthalten. Mit diesem Coveragekriterium haben wir ein Kriterium gefunden, dass zumindest die einfachen Pfade, ohne Kreise gut abdeckt. Wir können theoretisch also folgern, dass dieses Kriterium gut geeignet wäre. Praktisch angewandt zeigt sich jedoch schnell, dass dieses Kriterium sehr viele redundante Tests erzeugt. Was per-se nicht schlecht ist jedoch keinen großen Informationsgewinn bringt.

4.4.5 Prime-Path Coverage für GraphQL

Mit der PrimePath Coverage elimieren wir die Redundanz aus der SimplePath Coverage und stellen sicher, dass vor allem sehr relevante Pfade gefunden werden die wir zur Testabdeckung nutzen. Wir erhöhen so die Effizienz der Tests indem wir unrelevante Tests filtern und dennoch eine zufriedenstellende Abdeckung beibehalten.

4.4.6 Complete-Path Coverage für GraphQL

Mit der Complete-Path Coverage haben wir als Ziel, dass wir alle Pfade, die möglich sind, auch generieren und in unseren Tests berücksichtigen. Da GraphQL jedoch Zyklen erlaubt ist die Anzahl an potentiellen Pfaden unendlich. Hierdurch folgt, dass auch der Testraum unendlich werden würde mit der Complete-Path Coverage. Somit ist Complete-Path Coverage nicht umsetztbar da wir im Allgemeinen nicht ausschließen können und wollen, dass GraphQL keine Zyklen haben kann. Dies wäre ein zu großer Einschnitt in GraphQL die von einem Testtool nicht erwartet werden sollte.

4.4.7 Fazit

Die beiden geeignetsten Coveragekriterien sind SimplePath-Coverage und PrimePath Coverage wobei PrimePath Coverage hierbei als Verfeinerung bzw. Einschränkung von SimplePath gesehen werden kann. Da jeder PrimePath auch ein SimplePath ist. Es hängt nun vom SUT ab welches Kriterium zu nutzen ist. Im Allgemeinen lässt sich sagen, dass wir SimplePath-Coverage nutzen sollten für kleinere, einfachere Schemas da wir somit eine gute Testabdeckung erreichen und die Offside der großen Redundanz noch nicht so schnell zu tragen kommt. Sollte das Schema nun stark wachsen und insbesondere viele Zyklen aufweisen, so empfiehlt sich die PrimePath Coverage da sie viel Redundanz entfernt.

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

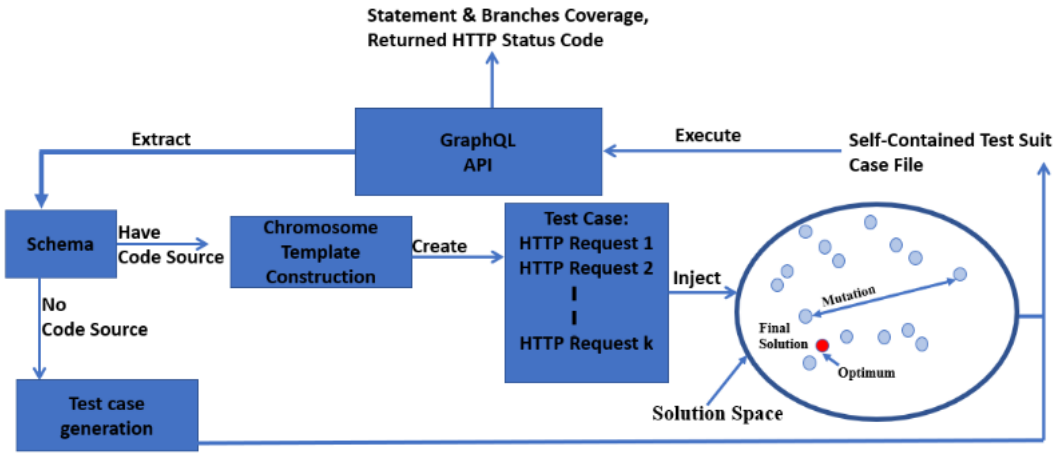
[[4][vgl. 3. Proposed Method]]

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-Bilbiothek Serene[10] Clojure.Specs[11] generiert und diese Clojure.Specs[11] dann nutzt um mit der Clojure-Bilbiothek 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 druchaus 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 beeinhaltet, 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 berücksichtig 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.

5.5 Vergleich der Arbeiten

Folgender Vergleich soll die verwandten Arbeiten noch einmal kurz einordnen.

Arbeit / Kriterium	**Property Based Testing**	**heuristisch suchen-basiertes Testen**	**Deviation-Testing**	**Query Harvesting**
Generierungsart	Zufallsbasierte Routenge-nerierung	Heuristische Suche	Erweiterung von beste-henden 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 geschriebe-nen Tests	stark Abhängig von User-requests
Orakel	simples Raten	mit Source Code: Analyse	Aus entwickel-ten 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

5.6 Andere Arbeiten

Hier ist eine kurze Übersicht über andere Arbeiten, dieses Kapitel ist sehr unwahrschein-lich in einer Abgabeversion. 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 Ünter der Haubefunktioniert

6 Testentwurfsprozess

Wie zuvor gesehen kann ein GraphQL-Schema mit Graphcoverage-Kriterien überdeckt werden. Wir wollen diese Gegebenheit 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:

```
{
  "kind": "",
  "name": "",
  "description": "",
  "fields": [],
  "inputFields": [],
  "interfaces": [],
  "enumValues": [],
  "possibleTypes": []
}
```

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 Defintion 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 $fields$ beeinhaltet Objekte folgender Struktur:

```
{
  "name": "",
  "description": "",
  "args": [],
  "type": {},
  "isDeprecated": "",
  "deprecationReason": ""
}
```

Listing 6.3: Type-Field

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

```
{
    "kind": "",
    "name": "",
    "ofType": null
}
```

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:

```
type Query {
  book(id: ID): Book
  author(id: ID): Author
  publisher(id: ID): Publisher
}

```

```
type Book {
  id: ID
  title: String
  author: Author
  publisher: Publisher
}

type Author {
  id: ID
  name: String
  books: [Book]
}

type Publisher {
  id: ID
  name: String
  books: [Book]
}
```

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*.

```
{
```

```
      "name": "book",
      "description": null,
      "args": [
        {
          "name": "id",
          "description": null,
          "type": {
            "kind": "SCALAR",
            "name": "ID",
            "ofType": null
          },
          "defaultValue": null
        }
      ],
      "type": {
        "kind": "OBJECT",
        "name": "Book",
        "ofType": null
      },
      "isDeprecated": false,
      "deprecationReason": null
    }
```

Listing 6.6: book Field

```
    {
      "name": "author",
      "description": null,
      "args": [
        {
          "name": "id",
          "description": null,
          "type": {
            "kind": "SCALAR",
            "name": "ID",
            "ofType": null
          },
          "defaultValue": null
        }
      ],
      "type": {
        "kind": "OBJECT",
        "name": "Author",
        "ofType": null
```

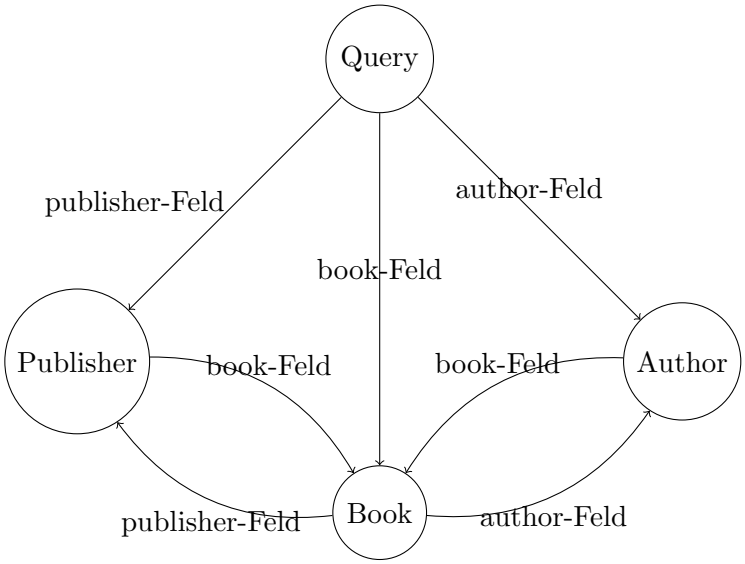
```
      },
      "isDeprecated": false,
      "deprecationReason": null
    }
```

Listing 6.7: author Field

```
    {
      "name": "publisher",
      "description": null,
      "args": [
        {
          "name": "id",
          "description": null,
          "type": {
            "kind": "SCALAR",
            "name": "ID",
            "ofType": null
          },
          "defaultValue": null
        }
      ],
      "type": {
        "kind": "OBJECT",
        "name": "Publisher",
        "ofType": null
      },
      "isDeprecated": false,
      "deprecationReason": null
    }
```

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 generierien, 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 Bedinungen 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 Startpunk 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 iterriert 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

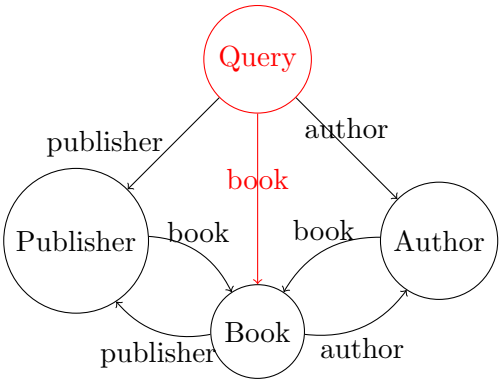
Da wir nun die Pfade, je nach Coveragekriterium ermittelt haben müssen wir nun eine Query aus diesen Pfaden bilden. Hierbei nutzen wir diverse Hinweise die im GraphQL-Schema gegeben werden und einen Teil der Arbeit aus [4, Property-Based Testing]. Jeder ermittelte Pfad startet im Query-Knoten. Da wir 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 berücksichtig 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. Ermittel ob die Kante $(Query, Book)$ Argumente benötigt.
3. Ermittel alle $SCALAR$ Types die $Book$ definiert.
4. Ermittel die Kante von $Book$ zu $Publisher$
5. Ermittel ob die Kante $(Book, Publisher)$ Argumente benötigt.
6. Ermittel alle $SCALAR$ Types die $Publisher$ definiert. (Pfad zuende daher kein Folge-Typ)
7. Baue Query-String aus den erlangten Informationen

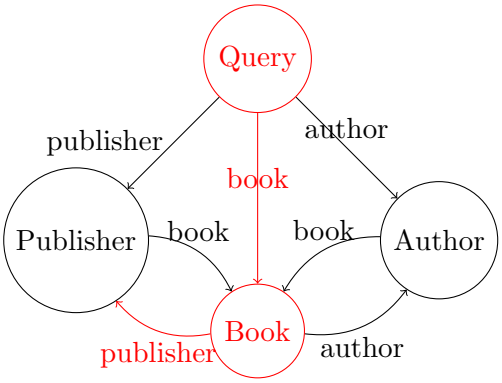
Wir werden nun Schritt für Schritt alle Schritte einzeln durchgehen und eine Query generieren.

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



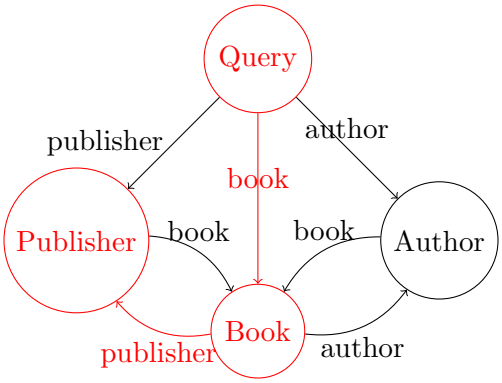
Wir betrachten die Kante (Query, Book). Durch unsere vorherige Definition des Graphens wissen wir, dass diese Kante das Attribut *book* trägt und uns somit hinweist, welches Feld wir benötigen um diese Kante in unserer Query abzubilden. Dieses entspricht *Listing*6.6 und wir erkennen, dass das Book-Field ein Argument *id* vom Typ *ID* besitzt. Daher wissen wir, dass wir dem Book-Feld 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 6.1: Schritte 1 - 3



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*.

Abbildung 6.2: Schritte 4 - 6



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:

```
{
}
```

Als Erstes fügen wir das gewählte Feld "Bookmit 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:

```
{
    book(id: RandomString){
        id
        title
    }
}
```

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:

```
{
    book(id: RandomString){
        id
        title
        publisher {}
    }
}
```

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

```
{
    book(id: RandomString){
        id
        title
        publisher {
            id
            name
        }
    }
}
```

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

Abbildung 6.3: Schritt 7

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 exisitenter 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

```
{
    book(id: RandomString){
        id
        title
        publisher {
            id
            name
        }
    }
}
```

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

```
{
    "data": {
```

```
                "book": {
                    id: "1",
                    title: "Moby Dick"
                    publisher: {
                        id: "1",
                        name: "Testverlag"
                    }
                }
            }
        }
```

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 Eingabevariabeln 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:

```
        {
            "data": {
                "book": null
            }
        }
```

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

```
{
    "data": {
        "book": {
            id: "1",
            title: "Moby Dick"
            publisher: {
                id: "1",
                name: "Testverlag"
            }
        }
    }
}
```

Listing 6.10: vollständige Response

Und die leere Antwort hätte eine Tiefe von 1

```
{
    "data": {
        "book": null
    }
}
```

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 geteste 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 $Schritt7.$ 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.

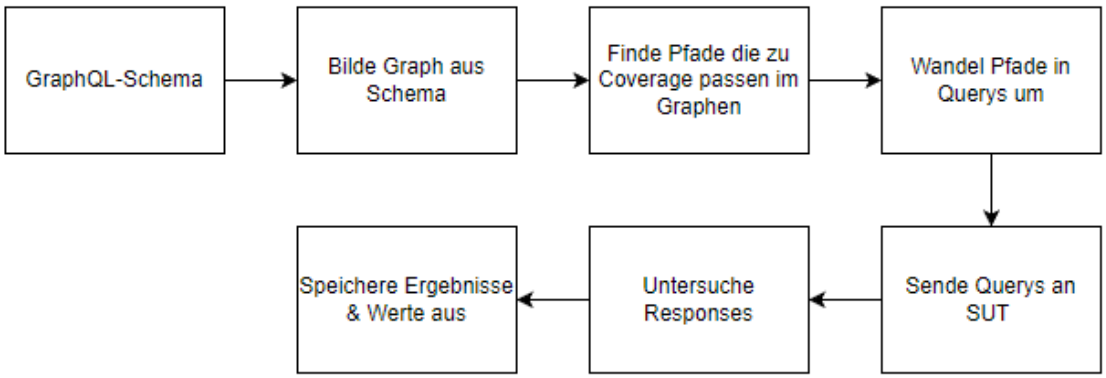
6.5.3 Negative Tests

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 Testautomatisierung

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. Vorraussetzungen 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-Biblitohek 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 einem 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 Bilbiothek als NetworkX. Faker ist eine Bilbiothek 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 Einzeilers 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 Zeitpunk 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.

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

```
def buildGraph(graph, type_name, type_dict):
    if type_name.startswith(nonSchemaTypePrefix) or
        type_name in baseDatatypes:
        pass
    else:
        for adjacentNode in type_dict[type_name]['fields']:
            if graph.has_edge(type_name, adjacentNode['type'
                ]['name']):
                return
            else:
                if adjacentNode['type']['name'] and
                    adjacentNode['type']['name'] not in
                    baseDatatypes:
                    graph.add_edge(type_name, adjacentNode['
                        type']['name'])
                    graph[type_name][adjacentNode['type']['
                        name']]["data"] = adjacentNode
                    buildGraph(graph, adjacentNode['type']['
                        name'], type_dict)
                if adjacentNode['type']['kind'] == 'LIST'
                    and adjacentNode['type']['ofType']['name'
                    ] not in baseDatatypes:
                    graph.add_edge(type_name, adjacentNode['
                        type']['ofType']['name'])
                    graph[type_name][adjacentNode['type']['
                        ofType']['name']]["data"] =
```

```
                    adjacentNode
16                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 *generare_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 *generare_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     pfadliste = []
3     generate_paths(startknoten, [], pfadliste, g)
4     return pfadliste
```

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,
                    g)
6       if is_prime_path(path, pathList):
7           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 Querys aus Pfad ermitteln

Die entwickelten Pfade werden in diesem Schritt nun in Querys umgewandelt, sodass diese an die GraphQL-API gestellt werden können. Eine Query beginnt in GraphQL immer im Query-Knoten und so beginnen auch alle unseren ermittelten Pfade in diesem Knoten. Da wir uns bei der Argumentgenerierung stark an *Property-based Testing* [4] halten erstellen wir pro Pfad n Querys. Standardmäßig ist $n = 5$. Um Querys aus einem Pfad zu erstellen wurde der *querygenerator* entwickelt. Der Querygenerator hat die Methode *pathToQuery* welche den Pfad, ein Typedict und den Graphen erwartet. Das Typedict ist ein Python-Dict, dass alle Informationen über das Schema enthält. Die Funktion erstellt nun eine Query indem der angegebene abgelaufen wird. Diese Funktionalität setzt die rekursive Funktion $resolvePathTillOnlyScalarTypesOrEnd(path, typedict, graph, query =$) um. Die Funktion arbeitet dabei so, dass sie für jeden Ausgangspunkt einer Kante alle $SCALAR$ Felder hinzufügt. So wird sichergestellt, dass alle einfachen Felder eines Objektes abgefragt werden. Es kann so validiert werden, dass das Objekt übereinstimmt mit der Schema-Definition. Sind alle $SCALAR$ Types hinzugefügt, so wird geprüft, welches Feld vom Type $OBJECT$ hinzugefügt werden muss um die Kante zum nächsten Knoten abzubilden. Sollte diese Kante Einträge im $args$ Feld besitzen, so werden die Argumente generiert. Da Argumente nur $SCALAR$ Types oder Aggregationen von $SCALAR$ Types sein können, benötigen wir Datengeneratoren für die Standarddatentypen. Hierbei kann es allerdings auch vorkommen, dass Argumente als Listen vorkommen. In allen Fällen sind die Argumente jedoch immer nur Varianten von Standarddatentypen. Die Auflösung der Argumente wird erledigt durch die Funktion $resolveArg$. Benötigt ein $OBJECT$ Feld nun Argumente, so werden diese der Query hinzugefügt. Anschließen wird die Kante aus der Liste des Pfades entfernt und die Funktion wieder rekursiv aufgerufen. Abbruchbedingung ist, dass der Pfad keine Kanten mehr besitzt. Wenn der Pfad keine Kanten mehr besitzt, so ist die entwickelte Query ein Test der den Pfad abdeckt.

Es sei unbedingt angemerkt, dass durch die zufällige Argumentgenerierung keinesfalls garantiert ist, dass bei der Testausführung dann der volle Pfad getestet wird. Sollte zum Beispiel ein Pfad direkt am Anfang Argumente benötigen, diese aber jedoch zu keinen Daten des SUT passen, so ist es sehr wahrscheinlich, dass der Test erfolgreich sein wird, ohne dass der eben entwickelte Pfad wirklich komplett getestetet wird.

7.2.4 Test ausführen / Testfile generieren

In der Testausführung halten wir uns auch relativ nah an die vorgestellte Methode in *Property-based Testing* [4]. Wir gehen hierbei davon aus, dass eine GraphQL-API die zu testen ist, genau so reagiert wie in 6.5 vorgestellt. Dies bedeutet, dass wir mittels HTTP-POST unsere generierte Query an die zu testende API senden und die Antwort im nächsten Schritt untersuchen. Eine exakte Umsetzung hierfür existieren in zwei verschiedenen Versionen. Einerseits werden on-the-fly die Tests im Programmdurchlauf ausgeführt.

```
1     queryResults = []
2     for testQuery in primePathQueries:
3         r = requests.post(testUrl, json={'query': testQuery
              })
4         response_as_dict = json.loads(r.text)
5         queryResults.append([testQuery, r, measurement])
```

Andererseits werden die Querys auch in einer $test_graphQL.py$ abgespeichert indem wir aus der Query einen Test in *PyTest* generieren.

```
1 for testQuery in primePathQueries:
2     f.write(pytestgenerator.generateTestFromQuery(testQuery,
          testUrl))
```

Der Test ist so definiert, dass er im Endeffekt auch ein HTTP-POST ausführt und direkt die Testauswertung enthält. Zur Testauswertung jedoch im folgenden mehr.

```
1 def generateTestFromQuery(testQuery, url):
2     testQueryAsValidString = testQuery.replace('"', '\\"')
3     testString = "def testQuery" + str(uuid.uuid4()).replace
          ('-', '') + "(caplog):" + "\n"
4     testString = testString + "    caplog.set_level(logging.
          WARNING)" + "\n"
5     testString = testString + "    response = requests.post
          (\"" + url + "\"," "json={\'query\':\"" +
          testQueryAsValidString + "\"})" + "\n"
6     testString = testString + "    response_as_dict = json.
          loads(response.text)"  + "\n"
7     testString = testString + "    measurement = queries.
          compareQueryResults(response_as_dict, " + "\"" +
          testQueryAsValidString + "\"" + ")" + "\n"
```

```
8       testString = testString + "    if measurement[\"
            expectedPathLength\"] > measurement[\"
            pathLengthFromResult\"]:" + "\n"
9       testString = testString + "        logging.warning(\"
            Test hat nicht 100% Abdeckung \")" + "\n"
10      testString = testString + "    assert response.
            status_code == 200" + "\n"
11      testString = testString + "\n" + "\n"
12      return testString
```

7.2.5 Testauswertung

Die Testauswertung erfolgt in zweierlei Maß. Einerseits werden Tests ad-hoc ausgeführt und ausgeführt andererseits wird ein Datei mit PyTests generiert. Diese dient insbesondere dafür, dass etwaiige fehlerhafte Tests zu einem späteren Zeitpunkt mittels PyTest noch einmal ausgeführt werden können. Für die Auswertung der Tests halten wir uns in beiden Beispielen an die HTTP-Statuscodes der Antwort sowie an die zuvor vorgestellte Messmethode der Pfadlängen.

Ad-Hoc Tests

Die Ad-Hoc Testausführung erfolgt durch ein simples HTTP-POST and das SUT. Zuerst sammeln wir dann alle Ergebnisse der Tests und werten sie zu einem späteren Zeitpunkt genauer aus.

```
1  queryResults = []
2  for testQuery in primePathQueries:
3      r = requests.post(testUrl, json={'query': testQuery})
4      response_as_dict = json.loads(r.text)
5      measurement = queries.compareQueryResults(
           response_as_dict, testQuery)
6      queryResults.append([testQuery, r, measurement])
```

Wir berechnen in Place dann die Pfadlängen von Response und Erwartung. Alles wird dann in einer Liste mit den QueryResults gespeichert. In der späteren Auswertung sortieren wir die Tests dann in 4 verschiedene Kategorien.

```
1  for queryResult in queryResults:
2      testCount = testCount + 1
3      if any(substring in queryResult[1].text for substring in
           ["GRAPHQL_PARSE_FAILED", "GRAPHQL_VALIDATION_FAILED"
       ]):
4      own_failure = own_failure + 1
5      elif "INTERNAL_SERVER_ERROR" in queryResult[1].text or "
           error" in queryResult[1].text:
```

```
6       print(queryResult[1].text)
7       server_failures = server_failures + 1
8       elif "data" in queryResult[1].text and queryResult[2]["
            expectedPathLength"] > queryResult[2]["
            pathLengthFromResult"]:
9       successfull = successfull + 1
10      elif "data" in queryResult[1].text and queryResult[2]["
            expectedPathLength"] == queryResult[2]["
            pathLengthFromResult"]:
11      perfect = perfect + 1
```

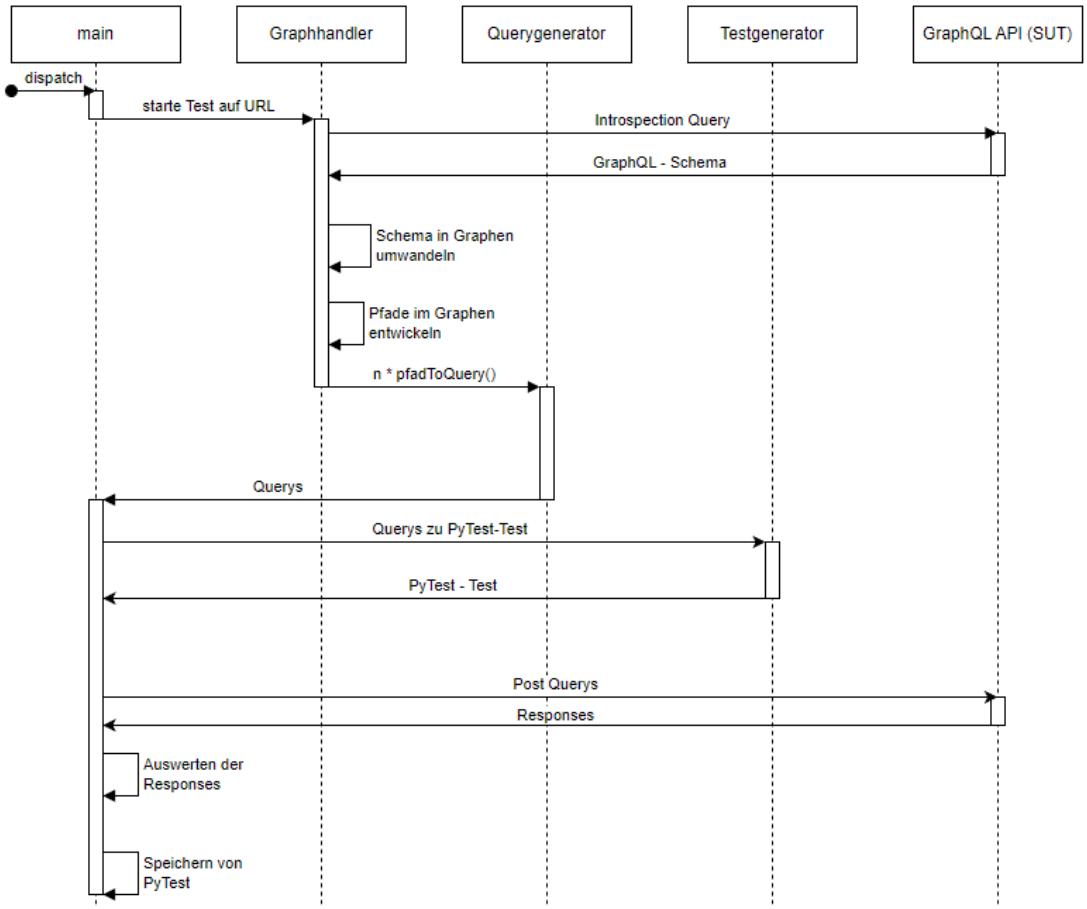
Die Kategoriesierungen sind $Good\_Test$, $Perfect\_Test$, $malformed\_Test$ und $confirmed\_failed\_Test$ Ein Good_Test ist ein Test, der keinen fehler erzeugt hat allerdings die Pfadlänge von Erwartung != Response. Perfect_Tests sind Tests, die keinen Fehler erzeugen und Pfadlänge Erwartung = Response. Malformed_Tests sind Tests, die fehlerhaft sind aber allerdings aufgrund von Generierungsfehler unsererseits. Dies sind also Tests die vernachlässigt werden müssen als False-Negatives. Confirmed_Failed_Tests sind Tests bei denen wir tatsächlich einen Fehler bekommen. Die Fehlerhaften Tests werden dann im folgenden Ausgegeben mit der konkreten Fehlerbeschreibung, sodass eine Fehleranalyse möglich wird.

PyTest Tests

Die erstellten Tests werden, wie in 7.2.4 vorgestellt, in einer Datei Namens $test\_GraphQL.py$ abgespeichert. Wenn PyTests installiert ist so lassen sich diese alle ausführen mit dem Befehl *pytest*. Diese Datei dient insbesondere zur Validierung von Codeänderungen. Man stelle sich vor, dass die fehlerhaften Code-Teile ausgebessert wurden. Da wir nun mit Random-Argumenten arbeiten können wir jedoch nicht garantieren, dass die selben Tests generiert werden. Daher speichern wir sie in der Datei ab um sie später noch einmal nutzen zu können und zu validieren, dass etwaiige Fehlerbehebungen wirksam waren und der spezielle Test nicht mehr scheitert.

7.3 Zusammenfassung der Implementation

Die entwickelte Methode lässt sich in folgendem Sequenzdiagramm gut abbilden:



Durch die Generierung der Tests auf zwei Arten erreichen wir, dass einerseits die Tests direkt ausgeführt werden, sodass das Tool direkt Feedback zurückgibt. Andererseits verbessern wir die Überprüfbarkeit indem eine Pytest-File generiert wird die sicherstellt, dass die Tests erneut ausgeführt werden können. Limitierungen der Methode liegen insbesondere in den Argumentgeneratoren. Diese haben teilweise eine sehr schlechte Generierungskapazität im allgemeinen. Dies liegt daran, dass wir den BlackBox-Testing Ansatz verfolgen und wir somit kein Domänenwissen über die zugrunde liegenden Daten besitzen. Insgesamt lässt sich sagen, dass die Methode gut funktioniert und fähig ist, fehler zu finden in GraphQL-APIs. Hierzu jedoch im folgenden, insbesondere bei den Experimenten, mehr.

8 Auswertung / Experiment / Vergleich mit Property-based Testing

8.1 Vergleichmetriken

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 nocheinmal 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 exisitiert, starten wir mit einem sehr einfachen und intuitiven Ansatz* [4, vgl. B. Measuring Schema Coverage]. In der Tat ist das vorgestellte Coveragekriterirum 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:

```
{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) ¿ 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 Vorraussetzungen 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 genutz wird. Eine beispielhafte Anpassung wäre hier nun, dass wir den Generator für den Type *ID* so

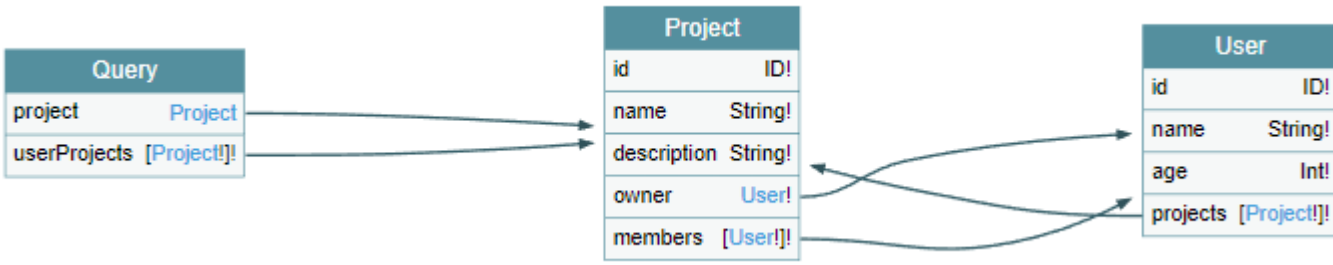
anpassen, dass er nur Argumente für ID zurückliefert die ein Zahlenwert sind in einem gewissen Bereich der durch das SUT abgebildet wird.

8.3 Fehlerfindungskapazitäten

Zuerst wollen wir die Fehlerfindungskapazitäten des Testtool auf die Probe stellen. Hierfür nutzen wir die beiden, zuvor benannten, Testsyteme GraphQL-Toy (eine experimentelle GraphQL-Implementierung) und GitLab in der Version 12.6.3. Ziel ist es mindestens die Fehler zu finden die vom *Property-based Testtool*[4] gefunden wurden. Idealerweise wollen wir jedoch sogar mehr Fehler finden.

8.3.1 GraphQL-Toy

Das Testssystem GraphQL-Toy hat ein sehr simples Schema in dem nur drei $OBJECT$ Typen exisitieren. Diese sind $Query$, $Project$ und $User$. Das Schema hat folgende Struktur:



Entwickelt wurde dieses System mit dem Hintergrund, dass bekannte Bugs im Code eingebracht werden und überprüft werden kann ob das Testtool diese findet. Insgesamt wurden 15 verschiedene Buggs vorgestellt welche in verschiedene Kategorien fallen wie Syntaxfehler, falsche Rückgabedaten, falsche Datenstrukturen etc. Einige der Bugs werden im folgenden kurz vorgestellt. Eine Liste aller eingebauten Bugs lässt sich im Appendix unter *GraphQL-Toy Implementation mit Bugs* 12 finden.

Bug 1 - SyntaxFehler

Einfache Syntaxfehler wurden an verschiedenen Stellen eingebaut. Dies bedeutet, dass jeder Funktionsaufruf dieser Funktion garantiert scheitern wird. Somit kann jede Request diesen Fehlerfall entdecken, solange die Request auch das Feld hinter der Funktion mit dem Syntaxfehler abfragt. Ein einfacher Syntaxfehler wäre zum Beispiel folgender Code:

```
    const resolvers = {
        Query: {
            project: (_, {id}, context, info) => {
                // Example bug 1 - Syntax mistake
                return db.projects.find(project => project.
                    id ===);
            }
        }
    }
```

Hierbei fehlt der Wert mit dem die *project.id* verglichen werden soll. Ein jeder Aufruf dieser Funktion mit egal welcher *ID* führt zu einem Fehler.

Bug 2 - Falscher Objekttyp

Objektfehler sind ein wenig unoffensichtlichere Fehler. Hierbei gibt der Code ein Objekt zurück, dass nicht der definierten Struktur im Schema entspricht. GraphQL wird hierfür dann einen Fehler erzeugen da die Daten eben nicht zum Schema passen.

```
    const resolvers = {
        Query: {
            project: (_, {id}, context, info) => {
                // Example bug 5 - wrong type "error"
                return { ...db.projects.find(project =>
                    project.id === id), name: ["a", "b"] };
            }
        }
    }
```

Um diesen Fehler ausführen zu können ist es wichtig, dass das Feld auch abgefragt wird. Sollte das Feld ein Argument benötigen, so muss dieses passen, sodass auch wirklich ein Objekt abgefragt wird und dann der falsche Type zrückgegeben wird.

Bug 3 - Typfehler in der Eingabe

Felder wie *ID* sind im GraphQL-Standard als einzigartige Strings definiert. Im allgemeinen wird der *ID* Type jedoch von diversen Entwicklern als Zahlenstring genutzt. Eine Funktion wandelt diesen String dann in eine Zahl um die z.B. genutzt wird um einen bestimmten Eintrag eines Arrays zu bekommen. Inputvalidierung ist also von Nöten.

```
    const resolvers = {
        Query: {
            project: (_, {id}, context, info) => {
                // Example bug 3 - Input type validation bug
                return db.projects[id];
```

```
            }
        }
    }
```

Es ist hier möglich, ohne jegliche Prüfung einen Key anzugeben. Ist ein Resolver wie heir implementiert so ist es erlaubt in der Query jeglichen String anzugeben. Es fällt also sehr leicht, dass z.B. ein IndexOutOfBound Fehler auftreten kann.

Mit dem Testtool nach [4, Property-based Testing] konnten 73% der Fehler, also 11 der 15 Fehler gefunden werden. Unser entwickeltes Testtool schaffte auf der selben API auch eine Entdeckung von 11 Fehlern. Wir konnten also dieselbe Abdeckung erreichen wie das Property-based Tool. Bemerkenswert hierbei ist allerdings, dass das Property-based Tool hierfür wesentlich mehr Queries benötigte, um eine zufriedenstellende Coverage zu erreichen. Das Property-based Tool benötigte 30 Durchläufe, die jeweils bis zu 100% Edge-Coverage liefen um alle Fehler zu finden. Im Kontrast dazu konnte unsere hier entwickelte Methode mithilfe von nur 2 PrimePfaden eine PrimePath Coverage erreichen. Hierzu wurden für jeden Pfad 5 Testquerys entwickelt. Es war somit möglich, alle 11 Fehler mit nur 10 Querys zu finden. Mehr noch, im allgemeinen haben, durch die spezielle Struktur des SUT's rein theoretisch nur 2 Querys ausgereicht, welche gut geeignet Argumente aufwiesen. Eine Query ist gut geeignet wenn gilt, dass die erwartete Pfadlänge = wirkliche Pfadlänge. Das Testtool fand diese beiden Querys hierfür:

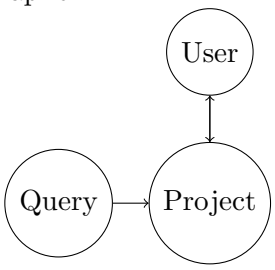
```
{ project(id: "2", ) {  id  name  description   owner {
    id  name  age   }  }  }
```

```
{ userProjects(id: "1") { name owner { id name age
   projects { name description id } } } }
```

Mithilfe dieser beiden Querys konnte jeder der 11 entdeckten Fehler gefunden werden. Dies liegt auch daran, dass der Argumentgenerator entsprechend angepasst wurde und nur valide IDs produziert hat. So war es sehr wahrscheinlich, dass eine ID die Generiert wird mindestens in einer der 5 erstellten Querys zur unterliegenden Datenstruktur gepasst hat und wir somit die eine tatsächliche Testausführung haben und nicht nur einen initialen *null* Wert der die Query sofort erfolgreich sein lässt. Die 4 nicht gefundenen Fehler sind die selben Fehler wie diese, die *Property-based Testing* [4, vgl. RQ.2] nicht finden konnte. Dies sind die Felder, in denen ein falscher Wert eines Objektes genutzt wurde um ein anderes Objekt zu erlangen. Hierbei verhindert der Black-Box Ansatz, dass der Fehler gefunden wird. Hierzu jedoch später mehr.

8.3.2 Auswertung GraphQL-Toy

Von den 15 injezierten Bugs wurden von beiden Tools jeweils 11 gefunden. Die injizierten Bugs klassifizierten sich in Syntax-Bug, Validierungs-Bugs, Filtering-Bug und Wrong-Type-Bug. [4, vgl. RQ 2 Type Errors]. In *Property-based Testing* fand der entwickelte Prototyp 11 Bugs. Es wurden alle Bugs gefunden die nicht in der Kategorie Filtering-Bug waren. Bugs der Kategorie Filtering-Bug wurden nicht gefunden da diese Bugs valide 200er Statuscodes zurückgeben aber oft leere Felder. Da durch den Black-Box Ansatz jedoch keine Information existiert, die uns verlässlich sagen kann, dass Daten falsch sind, werden diese Fehler akzeptiert und es wird kein Fehler erkannt. Diese Limitierung existiert in unserem hier entwickelten Prototyp auch. Wir haben exakt die selben Fehler finden können wie das Property-based Tool. Da wir in unserem Prototypen auch den Ansatz BlackBox-Testing gewählt haben, haben wir mit diesem Problem auch zu kämpfen.
Unser hier entwickelter Prototyp hat jedoch für die allgemeine Fehlerfindung wesentlich weniger Tests benötigt. Das Schema des GraphQL-Toy umfasste einen sehr simplen Graphen.



Unsere Methode ermittelte zwei Pfade als ausreichend für die Testgenerierung. Pro Pfad wurden dann 5 Tests erstellt. Aus diesen insgesamt 10 ermittelten Tests ergaben sich dann Tests die ausreichend waren ebenso alle Fehler zu finden. Bemerkenswert ist hier, dass das Tool nur ein einziges Mal seine Tests generieren musste. Es wurden somit 10 Tests generiert, die fähig waren, alle Fehler zu finden. In diesem kleinen Beispiel hat unser hier entwickelter Prototyp die selben Fähigkeiten wie der *Property-based Prototyp*. Die ausgeführten Experimente befinden sich im GitHub **github-toy-experimente**.

8.3.3 GitLab

8.3.4 Auswertung GitLab

8.4 Schema-Abdeckung

9 zukünftige Arbeit

9.1 BlackBox-Testing in WhiteBox-Testing umwandeln

9.2 Adaptive Generierung

10 Fazit

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

GraphQL-Toy Implementation mit Bugs

```
1
2  const {ApolloServer, gql, ApolloError} = require('apollo-
       server');
3
4  const typeDefs = gql`
5      type User {
6          id: ID!
7          name: String!
8          age: Int!
9          projects: [Project!]!
10     }
11
12     type Project {
13         id: ID!
14         name: String!
15         description: String!
16         owner: User!
17         members: [User!]!
18     }
19
20     type Query {
21         project(id: ID!): Project
22         userProjects(id: ID!): [Project!]!
23     }
24 `;
25
26 const db = {
27     projects: [
28         {
29             id: "1",
30             name: "Project 1",
31             description: "Awesome project!",
32             owner: "100",
33             members: ["100", "200"],
```

```
        },
        {
            id: "2",
            name: "Project 2",
            description: "Not an awesome project!",
            owner: "200",
            members: ["200"],
        },
    ],
    users: [
        {
            id: "100",
            name: "Burt",
            age: 23,
            projects: ["1", "2"],
        },
        {
            id: "200",
            name: "Earnie",
            age: 32,
            projects: ["2"],
        },
    ],
};

const resolvers = {
    Query: {
        project: (_, {id}, context, info) => {

            // Example bug 1 - Syntax mistake
            // return db.projects.find(project => project.id
                ===);

            // Example bug 2 - Give "foo", input validation
            // return db.projects[parseInt(id)];

            // Example bug 3 - Input type validation bug
            //return db.projects[id];

            // Example bug 4 - Using the wrong field
            // return db.projects.find(project => project.
               name === id);

            // Example bug 5 - wrong type "error"
```

```
76              // return { ...db.projects.find(project =>
                   project.id === id), name: ["a", "b"] };
77
78              // Example bug 6 - IndexOutOfBounds
79              // return db.projects[parseInt(id)];
80
81              // Correct implementation
82              return db.projects.find(project => project.id
                   === id);
83          },
84          userProjects: (_, {id}, context, info) => {
85              const user = db.users.find(user => user.id ===
                   id);
86
87              // Example bug 1 - Syntax Error
88              // return db.projects.filter(project => user.
                   projects.includes());
89
90              // Example bug 2 - Using the wrong field
91              // return db.projects.filter(project => user.
                   projects.includes(project.name));
92
93              // Example bug 3 - wrong type "errors"
94              // return db.projects.filter(project => user.
                   projects.includes);
95
96              // Correct implementation
97              return db.projects.filter(project => user.
                   projects.includes(project.id));
98          },
99      },
100     Project: {
101         owner: (project) => {
102             // Example bug 1 - Syntax mistake
103             // return db.users.find(user => user.id ===);
104
105             // Example bug 2 - Using the wrong field
106             // return db.users.find(user => user.name ===
                   project.owner);
107
108             // Example bug 3 - wrong type "error"
109             // return { ...db.users.find(user => user.id ===
                    project.owner), name: ["a", "b"] };
110
```

```
111             // Correct implementation
112             return db.users.find(user => user.id === project
                    .owner);
113         },
114         members: (project) => {
115             // Example bug 1 - logic error
116             // return db.users.filter(user => project.
                    members.includes());
117 
118             // Example bug 2 - Using the wrong field
119             // return db.users.filter(user => project.
                    members.includes(user.name));
120 
121             // Example bug 3 - wrong type "errors"
122             // return db.users.filter(user => project.
                    members.includes);
123 
124             // Correct implementation
125             return db.users.filter(user => project.members.
                    includes(user.id));
126         },
127     },
128     User: {
129         projects: (user) => {
130             // Example bug 1 - Syntax Error
131             // return db.projects.filter(project => user.
                    projects.includes());
132 
133             // Example bug 2 - Using the wrong field
134             return db.projects.filter(project => user.
                    projects.includes(project.name));
135 
136             // Example bug 3 - wrong type "errors"
137             // return db.projects.filter(project => user.
                    projects.includes);
138 
139             // Correct implementation
140             return db.projects.filter(project => user.
                    projects.includes(project.id));
141         },
142     },
143 };
144 
145 const server = new ApolloServer({typeDefs, resolvers});
```

```
146
147 server.listen().then(({url}) => {
148     console.log(`Server ready at ${url}`);
149 });
```

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     }
```

```
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
    deprecationReason
  }
  possibleTypes {
    ...TypeRef
  }
}

fragment InputValue on __InputValue {
  name
  description
  type {
    ...TypeRef
  }
  defaultValue
}

fragment TypeRef on __Type {
  kind
  name
  ofType {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
```

```
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
                kind
                name
              }
            }
          }
        }
      }
    }
  }
}
```

minimale Schema Response

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": null,
      "subscriptionType": null,
      "types": [
        {
          "kind": "OBJECT",
          "name": "Query",
          "description": null,
          "fields": [
            {
              "name": "book",
              "description": null,
              "args": [
                {
```

```
                    "name": "id",
                    "description": null,
                    "type": {
                      "kind": "SCALAR",
                      "name": "ID",
                      "ofType": null
                    },
                    "defaultValue": null
                  }
                ],
                "type": {
                  "kind": "OBJECT",
                  "name": "Book",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "author",
                "description": null,
                "args": [
                  {
                    "name": "id",
                    "description": null,
                    "type": {
                      "kind": "SCALAR",
                      "name": "ID",
                      "ofType": null
                    },
                    "defaultValue": null
                  }
                ],
                "type": {
                  "kind": "OBJECT",
                  "name": "Author",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "publisher",
                "description": null,
```

```
            "args": [
              {
                "name": "id",
                "description": null,
                "type": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null
                },
                "defaultValue": null
              }
            ],
            "type": {
              "kind": "OBJECT",
              "name": "Publisher",
              "ofType": null
            },
            "isDeprecated": false,
            "deprecationReason": null
          }
        ],
        "inputFields": null,
        "interfaces": [],
        "enumValues": null,
        "possibleTypes": null
      },
      {
        "kind": "OBJECT",
        "name": "Book",
        "description": null,
        "fields": [
          {
            "name": "id",
            "description": null,
            "args": [],
            "type": {
              "kind": "SCALAR",
              "name": "ID",
              "ofType": null
            },
            "isDeprecated": false,
            "deprecationReason": null
          },
          {
```

```
              "name": "title",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "author",
              "description": null,
              "args": [],
              "type": {
                "kind": "OBJECT",
                "name": "Author",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "publisher",
              "description": null,
              "args": [],
              "type": {
                "kind": "OBJECT",
                "name": "Publisher",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
          "interfaces": [],
          "enumValues": null,
          "possibleTypes": null
        },
        {
          "kind": "OBJECT",
          "name": "Author",
```

```
          "description": null,
          "fields": [
            {
              "name": "id",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "ID",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "name",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "books",
              "description": null,
              "args": [],
              "type": {
                "kind": "LIST",
                "name": null,
                "ofType": {
                  "kind": "OBJECT",
                  "name": "Book",
                  "ofType": null
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
```

```
            "interfaces": [],
            "enumValues": null,
            "possibleTypes": null
          },
          {
            "kind": "OBJECT",
            "name": "Publisher",
            "description": null,
            "fields": [
              {
                "name": "id",
                "description": null,
                "args": [],
                "type": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "name",
                "description": null,
                "args": [],
                "type": {
                  "kind": "SCALAR",
                  "name": "String",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "books",
                "description": null,
                "args": [],
                "type": {
                  "kind": "LIST",
                  "name": null,
                  "ofType": {
                    "kind": "OBJECT",
                    "name": "Book",
                    "ofType": null
```

```
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
          "interfaces": [],
          "enumValues": null,
          "possibleTypes": null
        }

      ]
    }
  }
}
```

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 definiton*, 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 21. Juli 2023 auf ihre Verfügbarkeit hin überprüft.