

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 Sakizoglou

Eidesstattliche Erklärung

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

.....
Ort, Datum

.....
Unterschrift

Inhaltsverzeichnis

1	Zusammenfassung	1
2	Einleitung	3
2.1	Motivation	3
2.2	Umsetzung	5
3	Grundlagen	6
3.1	Graphentheorie	7
3.1.1	Graph	7
3.1.2	gerichteter Graph	8
3.1.3	gewichteter Graph	8
3.1.4	Pfad	9
3.1.5	Kreis	9
3.1.6	Zusammenfassung	10
3.2	APIs	11
3.3	GraphQL	12
3.3.1	Schema & Typen	12
3.3.2	Query	14
3.3.3	Resolver	14
3.4	Zusammenhang Graphentheorie und GraphQL	16
3.4.1	Schema als Graph	16
3.4.2	Abfragen im Graphen	17
3.5	Testen	19
3.5.1	Typen von Tests	19
3.5.2	Arten von Tests	20
3.5.3	Testabdeckung	23
4	Graphüberdeckung	25
4.1	Graphüberdeckung allgemein	25
4.2	Graphüberdeckung Kriterien	26
4.2.1	Node Coverage	26
4.2.2	Edge-Coverage	26
4.2.3	Edge-Pair Coverage	27
4.2.4	Prime-Path Coverage	28
4.2.5	Complete-Path Coverage	29
4.2.6	abschließender Vergleich der Coverage-Kriterien	29
4.3	Graphcoverage für Code	30

4.4	Graphcoverage für GraphQL	31
4.4.1	Node-Coverage für GraphQL	32
4.4.2	Edge-Coverage für GraphQL	32
4.4.3	Edge-Pair-Coverage für GraphQL	32
4.4.4	SimplePath-Coverage für GraphQL	33
4.4.5	Prime-Path Coverage für GraphQL	33
4.4.6	Complete-Path Coverage für GraphQL	33
4.4.7	Fazit	33
5	verwandte Arbeiten	34
5.1	Property Based Testing	34
5.2	heuristisch suchenbasiertes Testen	35
5.3	Deviation Testing	36
5.4	Query Harvesting	36
5.5	Vergleich der Arbeiten	37
5.6	Andere Arbeiten	37
5.6.1	Empirical Study of GraphQL Schemas	38
5.6.2	LinGBM Performance Benchmark to Build GraphQL Servers . . .	38
5.6.3	GraphQL A Systematic Mapping Study	38
6	Testentwurf	39
6.1	Schema in Graph abbilden	39
6.2	Pfade aus Graph bilden	45
6.3	Coverage-Pfade ermitteln	46
6.3.1	filternder Ansatz	46
6.3.2	generativer Ansatz	47
6.4	Query aus Pfad ermitteln	48
6.5	Test ausführen & Testauswertung	52
6.5.1	Positive Tests	52
6.5.2	Falsch-Negative Tests	55
6.5.3	Negative Tests	55
6.6	Zusammenfassung der Methode und struktureller Vergleich mit Property-based Testing	56
7	Testautomatisierung	58
7.1	Tool- / Dependencyauswahl	58
7.1.1	NetworkX	58
7.1.2	Faker	59
7.1.3	PyTest	60
7.2	Umsetzung der Methode	60
7.2.1	Schema in Graph abbilden	60
7.2.2	Pfade aus Graph bilden	62
7.2.3	Querys aus Pfad ermitteln	63
7.2.4	Test ausführen / Testfile generieren	64

7.2.5	Testauswertung	65
7.3	Zusammenfassung der Implementation	67
8	Auswertung / Experiment / Vergleich mit Property-based Testing	68
8.1	Vergleichsmetriken	68
8.1.1	Metriken aus Property-based Testing	68
8.1.2	Fehlerfindungskapazitäten	68
8.1.3	Neue Metrik	69
8.2	Threats to Validity / Limitierungen	70
8.2.1	Argumentgeneratoren	70
8.3	Fehlerfindungskapazitäten	71
8.3.1	GraphQL-Toy	71
8.3.2	Auswertung GraphQL-Toy	73
8.3.3	GitLab	74
8.3.4	Auswertung GitLab	75
8.4	Schema-Abdeckung	77
8.4.1	GraphQL-Toy Schema Coverage	77
8.4.2	GitLab Schema Coverage	78
8.5	Zusammenfassung der Experimente	79
9	zukünftige Arbeit	39
10	Fazit	40
11	Glossar	83
12	Anhang	84
	Literaturverzeichnis	108

1 Zusammenfassung

Im Zuge der digitalen Transformation nimmt die Anzahl von Softwareanwendungen rasant zu [1]. Insbesondere durch das Internet of Things und die generelle fortschreitende Vernetzung diverser Geräte nimmt Netzwerklast stark zu. [2] Bisheriger Standard für Kommunikation von Geräten über das Internet waren REST-APIs [3, vgl. Introduction] diese haben jedoch gewisse Limitierungen wie zum Beispiel: Ineffizienz durch Overfetching/Underfetching, Anzahl an Requests, Versionierung, Komplexität und vieles mehr. [3] 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 [4] und dabei auch nur die wirklich gewünschten Daten überträgt. [3, vgl. Advantages of GraphQL APIs] 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. [5, S. 16] 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. [6, vgl. Introduction] Im Rahmen der internationalen Konferenz für Automatisierung von Softwaretests IEEE/ACM 2021 wurde ein Paper veröffentlicht, dass eine Methode vorstellt wie GraphQL-APIs mithilfe von Property-based Tests automatisch getestet werden können. Property-based bezieht sich darauf, dass die Eigenschaften eines Objektes genutzt werden, um diese zu testen. Diese Methode generiert, der Datenstruktur angepasste, zufällige Tests und bietet so eine Möglichkeit, Fehler zu entdecken, ohne ein tiefgreifendes Wissen des zu testenden System zu besitzen. [6, vgl. Proposed Method] Die zufallsbasierte Testgenerierung weist allerdings einige Schwachstellen auf. So kann Sie nicht garantieren, dass die generierten Tests zu jeder Zeit eine gute Abdeckung der GraphQL-API haben, denn es können einzelne Routen der API komplett ausgelassen werden. Es sind Testszenarios denkbar, die sehr viele false-positives durchlassen und somit die Qualität der Software nicht ausreichend sicherstellen können. GraphQL ermöglicht außerdem einen potenziell unendlichen Suchraum für die Tests. Um den potenziell unendlichen Suchraum einzugrenzen wurde ein Rekursionslimit eingeführt, dass die Testlänge limitiert. Diese Limitierung führt dazu, dass die Testabdeckung nur bis zu einem bestimmten Komplexitätsgrad des zu testenden System ausreichend ist. Mit dieser Arbeit wurde ein anderer Ansatz für die automatisierte Testgenerierung untersucht, um die Testabdeckung verlässlich zu verbessern. Ein Schwerpunkt der Arbeit lag hierbei darin, zuerst die theoretische Verknüpfung von GraphQL mit der Graphentheo-

rie herzustellen. Das gewonnene Wissen wurde verwendet um zu analysieren, welches Graphüberdeckungskriterium sich ideal eignen würde für einen Prototyp. Die erlangten Kenntnisse halfen bei der Entwicklung eines Prototypes für die Testentwicklung. In zwei Experimenten konnte gezeigt werden, dass die entwickelte Methode in der Lage ist, Fehler in GraphQL-APIs zu finden. Insgesamt war es möglich, einen Punkt aus [6, VI. Future Work] zu erweitern, indem für eine bessere Graphabdeckung gesorgt wurde.

2 Einleitung

In diesem Kapitel führen wir an das Thema heran und stellen unsere Motive dar. Wir definieren, welche Ziele wir in dieser Arbeit verfolgen und geben abschließend eine grobe Übersicht über die Kapitelstruktur.

2.1 Motivation

Mit einer steigenden Nutzung von GraphQL wird es immer wichtiger, Tests für GraphQL-APIs zu entwickeln damit eine gute Softwarequalität sichergestellt werden kann. Die Entwicklung von Tests kann manuell oder automatisch geschehen. Bei Unit-Tests, also Tests für einzelne Funktionen, kann ein Programmierer selbst entscheiden, ob er diese selbst schreiben will oder von einem Tool automatisch generieren lassen will. Integrations-Tests, also Tests die Kombinationen einzelner Interaktionen von Funktionen miteinander testen, hingegen haben sehr oft einen sehr großen Testraum, sodass ein manuelles Schreiben dieser Tests fehleranfällig und langwierig ist. Für REST-APIs existieren schon automatische Integrationstesttools wie zum Beispiel: EvoMaster [7], QuickREST [8] oder RESTTESTGEN [9]. GraphQL-APIs haben leider noch einen Mangel an solchen automatischen Testtools. Im Rahmen der internationalen Konferenz für Automatisierung von Softwaretests IEEE/ACM 2021 wurde mit *Automatic Property-based Testing of GraphQL APIs* [6] eine Methode vorgestellt, die diesen Mangel angehen soll. Es wurde eine Methode entwickelt die aus dem GraphQL-Schema, also der Beschreibung der Datenstruktur der API, Tests zufällig generiert und damit versucht Fehler in der Programmierung zu finden. Die entwickelte Methode arbeitet nach folgendem Prinzip:

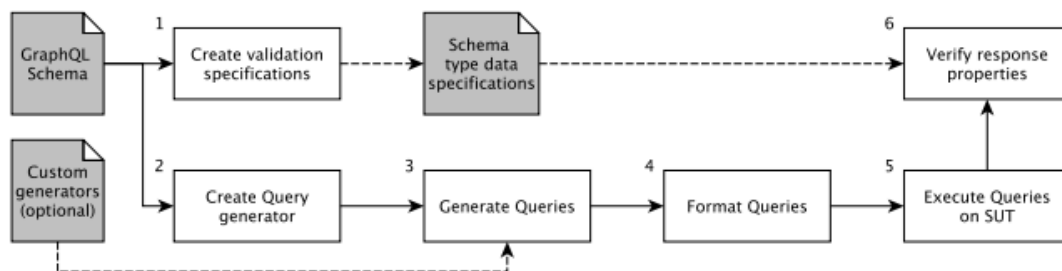


Abbildung 2.1: Methode von [6]

Es wird aus einem GraphQL-Schema ein Testgenerator entwickelt. Dieser kann aus der Typspezifikation, die GraphQL vorgibt, valide GraphQL-Querys entwickeln und diese mit verschiedenen Argumenten anreichern. Die generierten Querys stellen die entwickelten Tests dar. Das Besondere an GraphQL ist jedoch, dass es, wie der Name schon andeutet, einen Graphen umsetzt. Ein sehr einfaches Schema lässt sich beispielhaft in diesen Graphen übersetzen:

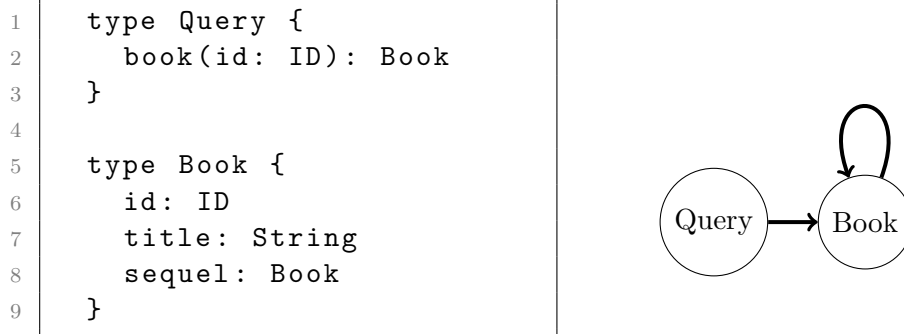


Abbildung 2.2: GraphQL-Schema als Graph

Generell starten alle GraphQL-Querys im Query-Type. Erlaubte Anfragen sind dann alle Pfade mit Ursprung im Query Knoten wobei Limitierungen implementiert werden können. Property-based Testing nimmt nun die definierten Felder im Query-Type und geht die Pfade welche sich im Graphen ergeben zufällig ab. Nach einer bestimmten Anzahl an zufälligen Iterationen wird die Query aus dem erlangten Pfad generiert und ausgeführt. Nutzen wir nun jedoch ein wesentlich größeres Schema, zum Beispiel die GraphQL-API von GitLab [10] so erkennt man schnell, dass der Graph so komplex wird, dass eine zufällige Pfadgenerierung zu unzuverlässig und ineffizient ist um eine große Struktur zuverlässig zu testen. Einen ähnlichen Sachverhalt findet man in der Testgenerierung für Programmcode. Dieser kann sehr komplex werden und es müssen Strategien gefunden werden um diese effizient und zuverlässig zu testen. Ein häufig verwendeter Ansatz ist es, den Code in einen Kontrollflussgraphen zu überführen, bei dem die Knoten Anweisungen oder Operationen darstellen und die Kanten den möglichen Pfaden entsprechen, die während der Ausführung des Programms genommen werden können. Hierbei wurde schon erhebliche Arbeit geleistet und diverse Kriterien entwickelt wie man eine gute Testabdeckung entwickelt. An dieser Stelle sei insbesondere an *Introduction to Software Testing* [5] verwiesen. In [5] wird die Graphenüberdeckung erarbeitet und praktisch gezeigt, wie sie helfen kann um Tests zu generieren. Es werden verschiedene Kriterien vorgestellt, die den Graphen auf unterschiedliche Art und Weise betrachten. Wir wollen zeigen, dass das in [5] erarbeitete Wissen nicht nur für Testentwicklung von Programmcode zielführend ist, sondern auch für die Testgenerierung von anderen Graphstrukturen verwandt werden kann, in unserem Fall für die Testgenerierung für GraphQL-APIs. Unser Fokus wird sich auf die PrimePath-Überdeckung [5, vgl. Criterion 2.4] richten da wir uns erhoffen, dass diese einen guten Mittelweg zwischen Testgenauigkeit, Fehlerfindung und Effizienz bietet.

2.2 Umsetzung

Zuallererst wird in dieser Arbeit die grundlegende Theorie in Kapitel 3 definiert und in Bezug zueinander gesetzt. Wir beginnen mit der Definition einiger Konzepte aus der Graphentheorie in Abschnitt 3.1. Daraufgehend sehen wir uns GraphQL präziser in Absatz 3.3 an und verbinden dann beide Absätze miteinander, indem wir einen konkreten Zusammenhang zwischen beiden Themen herstellen. Abschließend für die grundlegende Theorie führen wir in Absatz 3.5 umfassend in das Thema Softwaretests ein. Die zuvor erarbeitete Theorie wird dann im Kapitel Graphüberdeckung 4 erweitert durch Theorien und Definitionen. In diesem Kapitel zeigen wir außerdem, inwiefern die zuvor definierten Überdeckungskriterien hinreichend für Testgenerierung für Programmcode und GraphQL sind. Bevor wir unsere Methode entwickeln besprechen wir im Kapitel verwandte Arbeiten 5 wie der aktuelle Stand der Forschung zum Thema automatisiertes Testen von GraphQL ist. Mithilfe der zuvor entwickelten Theorie erarbeiten wir dann im Kapitel Testentwurf 6 ein Konzept, wie GraphQL automatisiert getestet werden kann. Dieses Konzept wird im Kapitel Testautomatisierung 7 eine praktische Umsetzung in einem Prototyp finden. Um nachzuweisen, dass der entwickelte Prototyp fähig ist, Fehler zu finden werden einige Experimente ausgeführt und mit der *Property-based* Methode [6] verglichen im Kapitel 8. Enden wird die Arbeit mit einem kleinen Ausblick in Kapitel ?? und einem Fazit über unsere erreichten Verbesserungen.

3 Grundlagen

Das automatisierte Testen von GraphQL-APIs erfordert ein spezifisches Domänenwissen in verschiedenen Teilbereichen der Informatik und Mathematik, insbesondere die Graphentheorie und das Softwaretesten. Dieses Domänenwissen wird in den folgenden Abschnitten auf Grundlage zweier Lehrbücher [5], [11] erarbeitet und in Kontext gesetzt. Wir benötigen die Graphentheorie um die Struktur von GraphQL auf einer abstrakten Ebene besser verstehen zu können. Außerdem setzen die in [5] vorgestellten Überdeckungskriterien ein grundlegendes Wissen über Graphentheorie und Softwaretests voraus.

3.1 Graphentheorie

Da GraphQL es ermöglicht, dass komplexe Beziehungen innerhalb eines Datenmodells in Form von Graphen modelliert werden [12, vgl. Modelling with Graph(QL)] benötigen wir die Graphentheorie, da diese Methoden liefert, um Graphen zu definieren und analysieren. Desweiteren sind die Testabdeckungskriterien die wir nutzen eng mit der Graphentheorie verbunden. Die folgenden Absätze werden eher theoretisch gehalten. Die Zusammenhänge zwischen der Graphentheorie und Testen von GraphQL-APIs werden sich jedoch später erschließen.

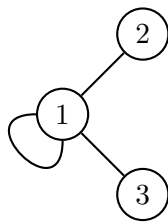
3.1.1 Graph

Ein Graph ist ein mathematisches Modell. Es kann dazu verwendet werden, Beziehungen zwischen Objekten darzustellen. Nach [11] ist ein Graph wie folgt definiert:

Definition 1 *Ein Graph ist ein Paar $G = (V, E)$ zweier disjunkter Mengen mit $E \subseteq V^2$ [11, vgl. S.2 0.1 Graphen]*

Die Elemente der Menge V nennt man Knoten (Vertices). Verbindungen zwischen den Knoten sind Elemente der Menge E und diese nennt man Kanten (Edges). In dieser Definition spielt die Ordnung der Elemente von E keine Rolle daher nennt man solche Graphen auch ungerichtete Graphen. Um Graphen darzustellen gibt es verschiedene Ansätze. Der geläufigste Ansatz ist es, Knoten als Punkte und Kanten als Verbindungslinien zu zeichnen. Häufig wird auch eine Adjazenzmatrix genutzt, bei dieser wird mit 0, 1 aufgeschlüsselt, welche Knoten eine Verbindung haben. Bei 0 existiert keine Kante zwischen den Knoten und bei 1 existiert eine.

Beispiel 1 *Ein Graph sei definiert mit $V = \{1, 2, 3\}$ und $E = \{(1, 1), (1, 2), (1, 3)\}$ Mögliche Darstellungen des Graphen sind:*



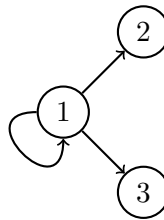
	1	2	3
1	1	1	1
2	0	0	0
3	0	0	0

3.1.2 gerichteter Graph

Gerichtete Graphen sind die Grundlage vieler Überdeckungskriterien. [5, vgl. 2.1 Overview]. Daher definieren wir sie hier.

Definition 2 Ein gerichteter Graph ist ein Paar $G = (V, E)$ zweier disjunkter Mengen mit zwei Funktionen $init: E \rightarrow V$ und $ter: E \rightarrow V$, die jeder Kante e eine Anfangsecke $init(e)$ und eine Endecke $ter(e)$ zuordnen. [11, S.26 0.10 Verwandte Begriffsbildungen]

Bei einem gerichteten Graphen ist somit die Sortierung der Kantenpaare wichtig. Die Funktionen $init$ und ter können am einfachsten durch die Sortierung der Kantenpaare erreicht werden. Hierbei ist das erste Element des Kantenpaares die Anfangsecke und das zweite Element ist die Endecke. Die Kanten in einem gerichteten Graphen werden mit einem Pfeil gezeichnet. Dabei zeigt der Pfeil stets in Richtung Endecke. Der in Beispiel 1 definierte Graph sieht als gerichteter Graph so aus:



3.1.3 gewichteter Graph

Für die spätere Entwicklung unseres Testentwurfs ist es wichtig, gewichtete Graphen zu definieren. Zuvor wurden eine Kante in Definition ?? als ein Tupel (x, y) eingeführt. Ein gewichteter Graph weist jeder Kante nun ein Kantengewicht zu, dies ist im allgemeinen eine positiv, reelle Zahl [13, vgl. S. 251].

Definition 3 Ein Graph $G = (V, E)$ mit einer Abbildung $g: E \rightarrow \mathbb{R}_{>0}$ heißt gewichteter Graph. Die Abbildung g heißt Gewichtsfunktion. Für $e \in E$ heißt $g(e)$ das Gewicht von e . Das Gewicht von G ist die Summe der Gewichte aller Kanten, $g(G) = \sum_{e \in E} g(e)$. [13, vgl. Definition 6.1 S. 251]

Für unsere späteren Anwendungszwecke muss die Definition jedoch ein wenig allgemeiner gefasst werden. Wir wollen später die GraphQL-Typen, als Gewichte nutzen, um unsere Tests zu entwerfen. Dafür verallgemeinern wir die Definition 3 indem die Gewichtsfunktion angepasst wird. Wir nennen dies vorerst allgemein gewichteter Graph.

Definition 4 Ein Graph $G = (V, E)$ mit einer Abbildung $g: E \rightarrow X$ heißt allgemein gewichteter Graph. Die Menge X ist frei wählbar.

Wir werden später auf Definition 4 verweisen wenn der konkrete Testentwurf entwickelt wird.

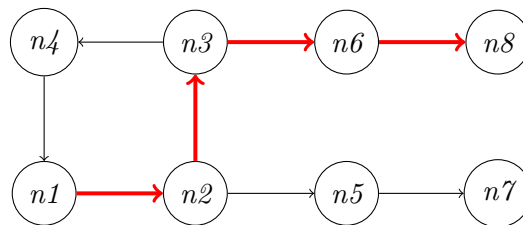
3.1.4 Pfad

Ein Pfad, oft auch Weg genannt, ist eine Sequenz von Knoten die nacheinander durch Kanten miteinander verbunden sind. [11, vgl. S. 7 0.3]

Definition 5 Ein Weg ist ein nicht leerer Graph $P = (V, E)$ der Form $V = x_0, x_1, \dots, x_k$ und $E = x_0x_1, x_1x_2, \dots, x_{k-1}x_k$ wobei die x_i paarweise verschieden sind. [11, vgl. S. 7]

Ein Weg wird oft durch die Folge seiner Knoten beschrieben also $P = x_0x_1 \dots x_k$ [11, vgl. S.7] Die Länge eines Weges ist die Anzahl der Kanten die dieser besucht. [11, vgl. S. 7] In gewichteten Graphen ist das Gewicht eines Pfades die Summe aller Gewichte der einbezogenen Kanten. [13, vgl. 7.2 kürzeste Wege]

Beispiel 2 Definieren wir einen Graphen G mit $V = \{n1, n2, n3, n4, n5, n6, n7, n8\}$ und $E = \{(n1, n2), (n2, n3), (n3, n4), (n4, n1), (n2, n5), (n3, n6), (n5, n7), (n6, n8)\}$ so wäre ein möglicher Pfad von Knoten $n1$ zu $n8$ der Pfad $p = \{(n1, n2), (n2, n3), (n3, n6), (n6, n8)\}$ Graphisch würde dies wie folgt aussehen (der Pfad in Rot markiert):



3.1.5 Kreis

Ein Kreis in einem Graphen ist ein Weg bei dem gilt: *Anfangsknoten = Endknoten* [11, vgl. S. 8] Die Größe eines Kreises ist die Länge des Weges die dieser Kreis bildet. Der kürzeste Kreis eines Graphens nennt sich *Tailenweiteng*(G) und der längste Kreis ist der Umfang [11, vgl. S.8] Der Graph aus Beispiel 2 hat einen Kreis der Länge 4 und ist in Abbildung 3.1.5 rot eingezeichnet.

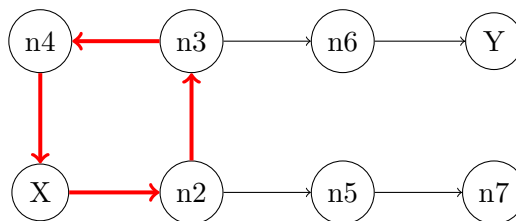


Abbildung 3.1: Ein zyklischer Graph

In unserem Kontext des Testentwurfs sind Kreise besonders interessant da diese für einen potentiell unendlich großen Testsraum sorgen. In einem azyklischen gerichteten

Graphen, einem gerichteten Graphen der keinen Kreis besitzt, ist die Menge aller möglichen Pfade endlich. Bei einem Graphen mit Zyklen ist die Menge aller möglichen Pfade unendlich. Dies folgt aus der Tatsache, dass jeder Pfad der den Kreis beinhaltet, diesen Kreis ein weiteres Mal ablaufen kann und somit stets ein neuer Pfad generiert wird.

3.1.6 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 vermehrt arbeiten. Insbesondere, wenn wir die Überdeckungskriterien im Bezug des Testens einführen benötigen wir das hier erarbeitete Wissen.

3.2 APIs

Im folgenden wird häufig die Rede sein von APIs. Daher soll hier kurz geklärt werden, was eine API ist und wie diese im Allgemeinen funktionieren. API steht kurz für Application Programming Interface (Programmierschnittstelle) [14, vgl.]. Eine API ermöglicht es, dass unabhängige Anwendungen miteinander kommunizieren und Daten austauschen können [14]. Im allgemeinen funktioniert diese Kommunikation mit dem HTTP-Protokoll über das Internet. Dabei stellt ein System per HTTP eine Anfrage und die API liefert eine Antwort. Es ist nicht festgelegt, auf welche Art das System die Antwort für die API erzeugt. Dies hängt stark von der zugrundeliegenden Implementierung ab.

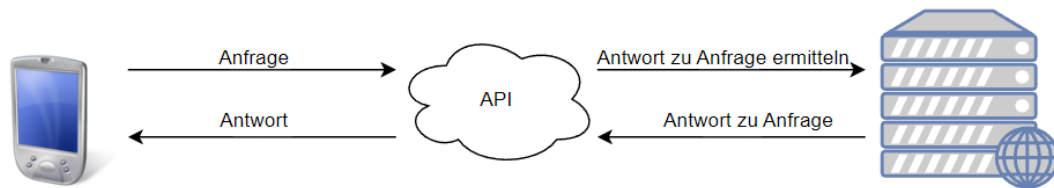


Abbildung 3.2: simple API-Kommunikation

Es gibt verschiedene Entwurfsmuster, die man für ein API Design nutzen kann. Die verschiedenen Muster werden durch Programmcode umgesetzt und sind somit fehleranfällig. In dieser Arbeit wird die Umsetzung von GraphQL-APIs betrachtet und der Programmcode hinter der GraphQL-API soll automatisiert getestet werden, sodass die Softwarequalität von der Programmierschnittstelle verifiziert werden kann.

3.3 GraphQL

GraphQL [15] ist eine Open-Source Query-Language (Abfragesprache) und Laufzeitumgebung die von Facebook entwickelt wurde [15, vgl. Introduction]. Die Besonderheiten von GraphQL sind, dass man mit nur einer einzelnen Anfrage mehrere Ressourcen gleichzeitig abfragen kann [16, vgl. No More Over- and Underfetching] und die Daten in einem Schema durch einen Typgraphen definiert sind [16, 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. GraphQL erleichtert außerdem die Kommunikation von Schnittstellen, indem die gewünschten Felder schon in der Query definiert werden und direkt den erwarteten Datentyp zusichern. Hier liegt auch der große Vorteil im Vergleich zum direkten technologischen Konkurrenten REST API [4, vgl. Welche REST-Einschränkungen versucht GraphQL zu überwinden?]. Bei REST-APIs sind nämlich für verschiedene Ressourcen auch jeweils eine eigene Anfrage nötig [16, vgl. No More Over- and Underfetching] und die Typsicherheit ist nicht so stark gegeben wie bei GraphQL-APIs [4, vgl. Zusammenfassung der Unterschiede: REST vs. GraphQL]. Diese beiden großen Vorteile sorgen dafür, dass GraphQL an Popularität gewinnt und zunehmend eingesetzt wird [17, vgl. Continued growth and the road ahead]. GraphQL ist eine Abfragesprache und Spezifikation, dies bedeutet, dass GraphQL selbst keine konkrete Implementierung für eine Schnittstelle ist. Implementierungen von der GraphQL-Spezifikation sind in GraphQL-Servern umgesetzt die in verschiedenen Programmiersprachen existieren. Eine umfassende Auswahl verschiedenster Implementierungen findet sich in [18]. Besonderer Beliebtheit erfreuen sich ApolloServer [19], Express GraphQL [20] und HyGraph [21]. Da jedoch alle Server die GraphQL-Spezifikation umsetzen müssen und unsere Tests aus GraphQL Anfragen bestehen, ist es für uns irrelevant welche konkrete GraphQL Serverimplementierung das zu testende System verwendet. Im Kontext dieser Arbeit ist ein tiefgreifendes, technologisches Verständniss von GraphQL essenziell, deshalb wird hier eine tiefgreifende Erklärung von GraphQL folgen.

3.3.1 Schema & Typen

Grundlage einer jeden GraphQL-API ist ein GraphQL-Schema. [15, vgl. Core Concepts] Das Schema definiert exakt, wie die Daten in der API aufgebaut sind und welche Informationen existieren [15, vgl. 3.2 Schema]. Ein GraphQL-Schema ist eine Sammlung von einzigartigen Typen und definiert die Einstiegspunkte der API. Es gibt drei Einstiegspunkte, die *query* zum Daten abfragen, *mutation* zum Daten verändern und *subscription* um über Datenänderungen informiert zu werden [15, vgl. 3.2.1 Root Operation Types] *Query* ist dabei als einziger verpflichtend, die anderen sind optional. [15, vgl. 3.2.1] Die drei Einstiegspunkte sind als Typen definiert. Ein Typ ist die fundamentale Einheit eines jeden GraphQL Schemas. [15, vgl. 3.4 Types].

Es gibt 6 verschiedene Typdefinitionen, diese sind:

Typ	Beschreibung
ScalarTypeDefinition	Primitive Datentypen die keine Verbindungen zu anderen Typen haben dürfen. (Strings, Integer, ...)
ObjectTypeDefinition	Komplexere Datentypen die Verbindungen untereinander haben und in Feldern ihre Verbindungen definieren
InterfaceTypeDefinition	Ein abstrakter Datentyp der die Struktur für andere Typen vorgibt
UnionTypeDefinition	Ein Typ der die Vereinigung verschiedener Typen ist
EnumTypeDefinition	Ein Typ der nur eine feste Anzahl an vorher festgelegten Werten hat
InputObjectTypeDefinition	Zusammensetzung von ScalarTypes um komplexere Eingabeargumente zu bilden

GraphQL Typen [15, vgl. 3.4 Types]

Ein Typ hat einen einzigartigen Namen und definiert alle Informationen über sich, hierbei wird für jede Information ein Feld angelegt. Das Feld setzt dann wieder jeweils einen Typen um (InputObjectTypeDefinition ist dabei ausgeschlossen). Ein sehr einfaches Schema wäre zum Beispiel die Beziehung zwischen Büchern und Autoren. Ein Buch hat einen Titel und einen Author. Ein Author hat einen Namen und ein Geburtsdatum. Zugehöriges Schema für dieses Beispiel sähe wie folgt aus:

```

1      type Buch {
2          title: String
3          author: Autor
4      }
5      type Autor{
6          name: String
7          geburtsdatum: Date
8      }
```

Abbildung 3.3: minimales Schema mit zwei Types

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. [15, vgl. 3.6 Objects]

Hierbei gelten folgende Einschränkungen für die Elemente des Tupels:

Feldname ein eindeutiger Feldbezeichner

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

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

3.3.2 Query

Da der Query-Type der Einstiegspunkt in alle Abfragen ist, welche es später zu testen gilt, soll dieser hier noch einmal nähere Betrachtung finden. Sämtliche valide Abfragen beginnen im Query-Type. Abfragen können mit und ohne Eingabeparameter angegeben werden. Informationen darüber finden sich im Schema. Die definierten Anfragen haben, wie jeder Typ, einen eindeutigen Bezeichner, welcher dann auch in der zustellenden Abfrage benutzt wird. Die Felder der Antwort hängen hierbei vom Feldtypen ab. Ist das Ergebnis der Query-Definition ein `ScalarType`, so wird es direkt ausgegeben. Ist es ein anderer definierter Typ, so muss näher bestimmt werden welche Felder erwartet werden, dabei muss die Abfrage stets mindestens ein `ScalarType` enthalten. Nutzen wir das Beispiel der Bücher & Autoren weiter, könnte man wie folgt einen Query-Type definieren:

```
1      type Query{
2          # liste aller b cher
3          getBooks: [Book]
4          # ein zuf lliges buch
5          getBook: Book
6          # author zum jeweiligen Buch
7          getBookByTitle(String title): Author
8      }
```

Abbildung 3.4: Query Type für Buch und Autor

Eine solche API wäre in der Lage, 3 verschiedene Anfragen zu beantworten. Man muss beachten, dass die Anfrage `getBookByTitle("Beispieltitel"){ author }` zwingend einen `Scalar` Typen aus dem Typ `Author` ausweisen muss. Die Anfrage wäre somit erst in dieser Form valide: `getBookByTitle("Beispieltitel"){ author{ name } }`

3.3.3 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 [19, vgl.]. 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. Sie setzen die Schnittstellenprogrammierung wie in Kapitel 3.2 angesprochen um. Für jeden Typ im Schema, muss ein Resolver implementiert werden, dies umfasst insbesondere die Query, Mutation und Subscription Typen aber auch alle selbstdefinierten Typen [19, vgl.]. 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);
    },
  },
};
```

Abbildung 3.5: ein einfacher Resolver

Wobei zu beachten ist, 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 Fehlerhaft sein kann. Ein Klassiker für GraphQL ist es, einzelne Attribute in einem Resolver zu vergessen. Ein Entwickler sollte für jeden Resolver, der eine Funktion darstellt, einen beziehungsweise mehrere Tests zur Verfügung stellen. Da sich die einzelnen Resolver aber auch untereinander aufrufen können, kann sich ein teils riesiger Testraum ergeben. Hierzu mehr in Kapitel 3.4.2 Um sicherzustellen, dass der Kombinationsaufruf der Resolver fehlerfrei ist, wird im Folgenden eine Methode entwickelt die gewährleistet, dass die möglichen Kombinationen ausreichend durch Tests abgedeckt sind und so die Qualität und Zuverlässigkeit von GraphQL-APIs erhöht wird.

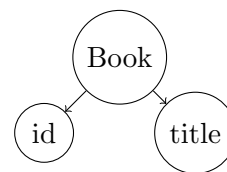
3.4 Zusammenhang Graphentheorie und GraphQL

Da wir nun ein grundlegendes Wissen über Graphentheorie und GraphQL erlangt haben, muss noch gezeigt werden, dass Graphentheorie auch anwendbar ist auf GraphQL. Wir nutzen diese Verknüpfung dann später, um Algorithmen die für Graphen gedacht sind für GraphQL zu nutzen.

3.4.1 Schema als Graph

Das GraphQL-Schema ist wie in Kapitel 3.3.1 gezeigt eine Komposition von Typen. Ein Typ definiert Felder in sich. Jedes Feld eines Typens zeigt seinerseits wieder auf ein anderes Feld [12, vgl. Modelling with GraphQL]. Somit wird jedes Feld eines Typens zu einer ausgehenden Kante. Definieren wir nun einen simplen Typen *Buch* welcher nur zwei Skalare Typen hat.

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



Dann hat der Typ *Buch* zwei Kanten, zu den jeweils beiden definierten Feldern *id* und *title*. In Kapitel 3.3.1 haben wir festgelegt, dass Skalare Typen keine eigenen Beziehungen haben. Daher können die Felder *id* und *title* selbst keine ausgehenden Kanten haben. Das generelle Prinzip der Graphbildung ist:

1. Erstelle Knoten aus jedem Typen und den definierten Feldern des Knotens
2. Ziehe für jeden Typ seine Kanten, indem alle Felder des Typens zum entsprechenden Knoten verbunden werden

Nach diesem Prinzip kann nun aus einem beliebig großen Schema ein gerichteter Graph gebildet werden.

3.4.2 Abfragen im Graphen

Wie behandelt GraphQL nun Anfragen intern und wie können wir dies in unserer Graphstruktur visualisieren? Für die Erklärung definieren wir zuerst ein Schema als Beispiel:

```
1 type Query {  
2     buch: Buch  
3     autor: Autor  
4 }  
5 type Buch {  
6     id: Int  
7     title: String  
8     autor: Autor  
9     verleger: Verlag  
10 }  
  
1 type Author {  
2     id: Int  
3     name: String  
4     schrieb: [Buch!]  
5 }  
6 type Verlag {  
7     name: String  
8 }
```

Abbildung 3.6: Schemadefinition

Dieses Schema wird durch folgenden Graphen repräsentiert:

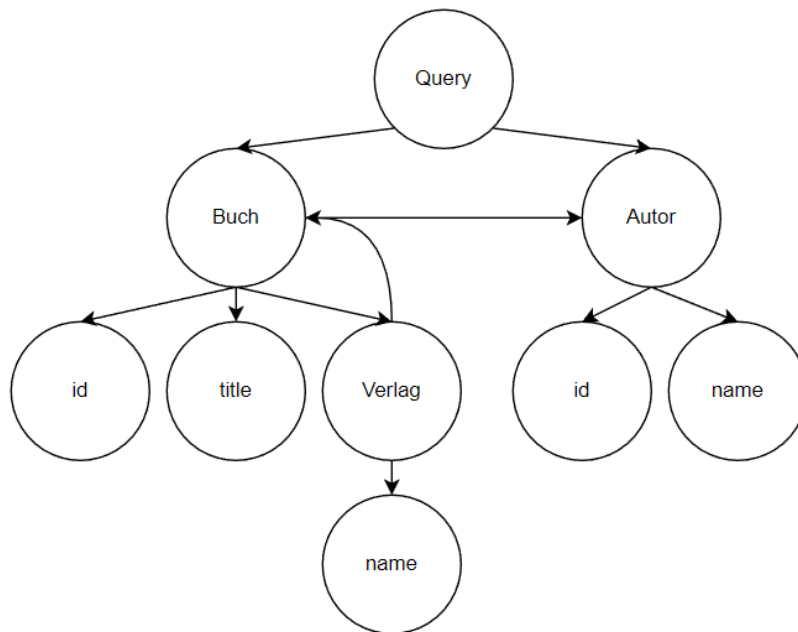


Abbildung 3.7: Graph für Schemadefinition aus Abbildung 3.4.2

Valide Anfragen an eine GraphQL-API sind nun alle Pfade, die mit Startknoten *Query* beginnen und mindestens einen Knoten beinhalten, der keine ausgehenden Kanten hat (also ein Scalar-Type) [12, vgl. Modelling with GraphQL]. Auf jeder Ebene des Graphens

werden Resolver ausgeführt welche für die Datenbereitstellung verantwortlich sind. Wollen wir nun einen GraphQL-Server mit dem zuvor definierten Schema ein Buch mit id, Titel und Verlag mit Verlagsnamen abfragen, so können wir diese Query nutzen: `{ buch { id title verleger { name }}}.`

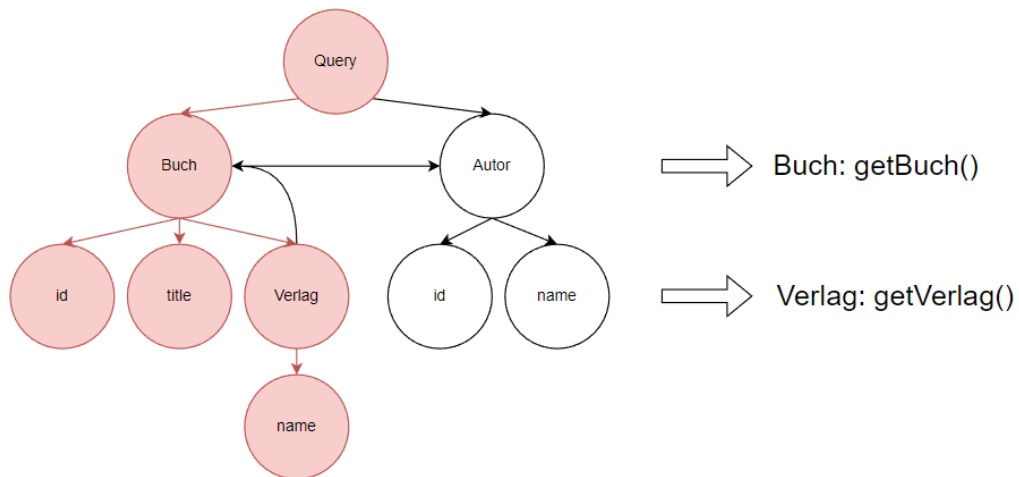


Abbildung 3.8: Graph für Schemadefinition nach [12]

Mit Ausführung dieser Query wird initial der Resolver `getBuch()` ausgeführt. Liefert dieser ein valides Ergebnis zurück, also gibt es ein solches Buch, dann wird der nächste Resolver `getVerlag()` ausgeführt [12, vgl. Resolver]. Dadurch, dass die Typen *Buch* und *Autor* aufeinander verweisen, ist der Graph des Schemas zyklisch und die Pfadmenge, das heißt die validen Abfragen die gestellt werden können, unendlich.

3.5 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. [5, vgl. Introduction] Um all dies sicherzustellen sind systematische Tests von Software nötig. Ziel ist es sicherzustellen, dass die Software den definierten Anforderungen und Spezifikation entspricht. Hierbei werden diverse Techniken und Ansätze verfolgt, die im folgenden kurz vorgestellt werden.

3.5.1 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 essenziell wichtig um zu entscheiden, welche Art von Tests angewandt werden. Es gibt zwei generelle Sichtweisen und eine Mischform. Das zu testende System wird als Box betrachtet. Diese Box kann aus verschiedenen Sichten gesehen werden, die wir im Folgenden näher erläutern. Die Ansätze unterscheiden sich vor allem in den zur Verfügung stehenden Informationen über das System.

White-Box Testing

Im WhiteBox-Testing stehen alle Informationen über das System zur Verfügung [22, vgl. 1.4.2 Code-Based Testing]. Der Tester hat Zugriff auf Code, Architekturdetails und besitzt Kenntnisse über alle möglichen Details des Systems [22, vgl. 1.4.2 Code-Based Testing]. 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. Verschiedene Techniken zur Analyse des Domänenwissens wurden entwickelt um die Informationen für die Testentwicklung zu nutzen. Wir werden anschließend in Kapitel 4 eine dieser Techniken näher untersuchen.

Black-Box Testing

Im BlackBox-Testing hat der Tester keinen Zugriff auf interne Funktionsweisen der Software. Schwerpunkt des Testens ist es, dass die Software das tut, was in den Anforderungen gefordert ist [22, vgl. Specification-Based Testing]. Da der Quellcode nicht einsehbar ist muss man sich darauf verlassen, dass die Anforderungen treffend formuliert wurden [23, vgl.]. Das BlackBox-Testing hat einen methodischen Bezug zu Property-based Testing [6].

Grey-Box Testing

Das Grey-Box Testing ist eine Mischform von White und BlackBox Tests [23, vgl.]. Es sind in dieser Sicht Teile der Software bekannt aber man hat keinen umfassenden Einblick

wie im White-Box Testing. Es werden funktionale als auch strukturelle Testansätze verfolgt, je nachdem wie viel Informationen über das System tatsächlich verfügbar sind [24, vgl.]. Das System wird aus der Sicht des Endbenutzers getestet [23, vgl.], jedoch mit zusätzlichem Wissen über Teile des internen Aufbaus.

3.5.2 Arten von Tests

Neben verschiedenen Transparenzen auf das zu testende System gibt es verschiedene Granularitätsebenen der Tests. Tests können von abgeleitet werden von Anforderungen, Spezifikationen, Designartifikaten und dem Programmcode [5, vgl. 1.1.1 Testing Levels Based on Software Activity] Dabei können verschiedene Level an Tests definiert werden, diese sind eng verbunden mit den Entwicklungsaktivitäten einer Software [5, vgl. 1.1.1].

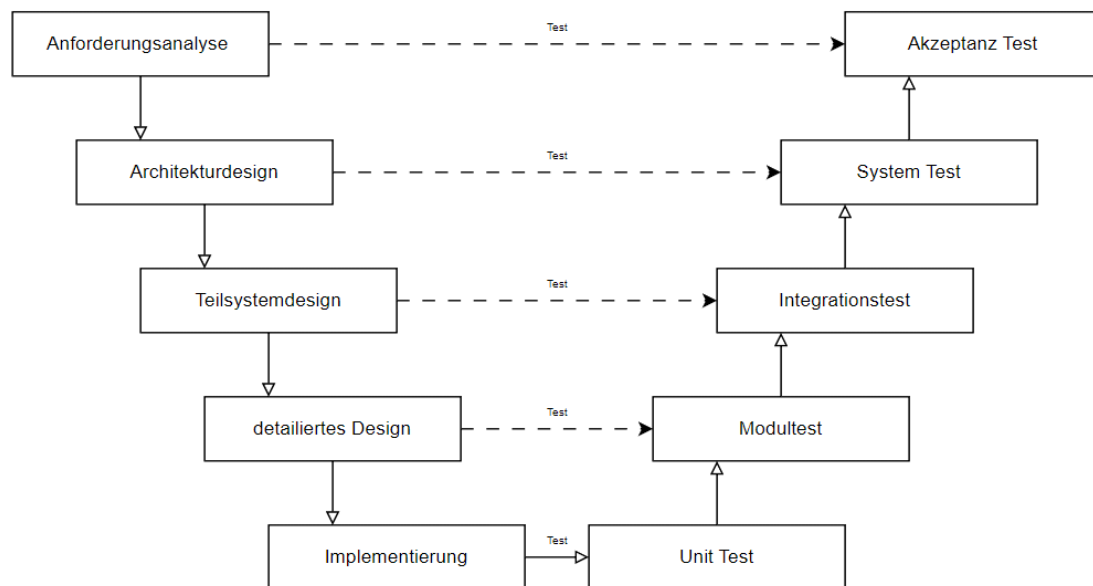


Abbildung 3.9: Softwareentwicklung und Test-Levels im V-Modell [5, vgl. Figure 1.2]

Die verschiedenen Testebenen sollten schon im Designprozess Beachtung finden, denn die Formulierung von Test kann dabei helfen Designfehler zu finden noch bevor die Software entwickelt wird. Die verschiedenen Ebenen der Tests spiegeln auch verschiedene Ansichten in den Tests wieder.

Azeptanz Test - Betrachten der Software hinsichtlich der Anforderungen

System Test - Betrachten der Software hinsichtlich der Architektur

Integrationstest - Betrachten der Software hinsichtlich der Teilsysteme

Modultest - Betrachten der Software hinsichtlich detailliertem Design

Unit Test - Betrachten der Software hinsichtlich konkreter Implementierung

[5, vgl. 1.1.1]

Im folgenden gehen wir noch einmal kurz auf jede einzelne Testebene etwas präziser ein. Dabei ist die Reihenfolge von feingranular zu grobgranular.

Unit Test

Als feingranularste Testebene ist das Ziel des Unit-Testings den entwickelten Code zu testen. Eine einzelne *Unit* ist in Objekt-Orientierter Programmierung eine Funktion oder Methode [22, vgl. Unit Testing]. Ein einzelner Unit Test konzentriert sich dabei auf eine Funktion oder Methode.

```
1      def add(a, b):  
2          return a + b
```

Abbildung 3.10: Eine einfache Python-Funktion

```
1      def test_add_positive():  
2          self.assertEqual(add(3, 5), 8)  
3  
4      def test_add_negative():  
5          self.assertEqual(add(-3, -5), -8)  
6  
7      def test_add_mixed():  
8          self.assertEqual(add(5, -3), 2)
```

Abbildung 3.11: Drei Unit Tests für die add-Funktion

Er prüft, ob die gegebene Einheit bei bekannten Eingaben die erwarteten Ausgaben liefert. Für diese Tests werden häufig Testframeworks genutzt die bei der Entwicklung und Ausführung der Tests helfen [5]. In Abbildung 3.10 wurde eine Funktion definiert und in Abbildung 3.11 sind drei UnitTest mit PyTest [42] definiert. Die Tests führen verschiedene Methoden aus und prüfen, dass das Ergebnis mit der Erwartung übereinstimmt. Das *assertEqual* übernimmt dabei die Auswertung.

Modul Test

Eine Granularitätsebene höher ist das Modul Testen. Ein Modul ist eine Sammlung von Units [5, vgl. S. 6]. Ziel ist es, die Interoperabilität der einzelnen Units in einem Modul sicherzustellen [5, vgl. S. 6].

```
1      def add(a, b):  
2          return a + b  
3      def sub(a, b):  
4          return a - b  
5      def mul(a, b):  
6          return a * b  
7      def quo(a, b):  
8          return a / b
```

Abbildung 3.12: Eine Python Rechenmodul

Ein Modul wird nun getestet, indem geprüft wird, dass die einzelnen Funktionen untereinander richtig miteinander interagieren.

```
1      def test_rechenmodul():  
2          testResult = (mul(add(2,3), sub(3,2)))  
3          self.assertEqual(testResult, 5)
```

Abbildung 3.13: Eine Modul Test

Anzumerken ist jedoch, dass die Interoperabilität im Modul-Test nur innerhalb eines Moduls getestet wird [5, vgl. S. 6].

Integrations Test

Das Integrationstesten übernimmt das Testen von Interoperabilität zwischen verschiedenen Modulen [5, vgl. S. 7]. Es wird dabei davon ausgegangen, dass die einzelnen Module zuvor korrekt getestet worden sind und die einzelnen Module korrekt arbeiten [5, vgl. S. 7]. Testobjekt sind vor allem die Schnittstellen der einzelnen Module und damit auch die Kommunikation zwischen den Modulen.

```
1      def nnwUser(name, gb, email):  
2          pw = generateRandomPw()  
3          return new User(name, gb, email, pw)
```

Abbildung 3.14: Modul 1

```
1 def saveUser(User user):
2     db.save(user)
3 def getUser(name):
4     db.findUser(name)
```

Abbildung 3.15: Modul 2

Ein Integrationstest für beide Module wäre nun, zu testen, ob ein neu angelegter Nutzer ordentlich gespeichert wird und ob dabei die zugewiesenen Daten auch passend bleiben.

```
1 def testAddNewUserAndSaveUserAndGetUser():
2     user = newUser("Peter", "01.04.1980", "p@test.de")
3     saveUser(user)
4     self.assertEqual(getUser("Peter"), user)
```

Abbildung 3.16: Integrationstest zwischen Modul 1 und Modul 2

Im Kontext dieser Arbeit werden wir untersuchen, wie solche Tests automatisiert für GraphQL erstellt werden können. Hinter jedem verschiedenen Typen von GraphQL steckt ein Resolver, welcher als ein eigenes Modul gesehen werden kann. Die Interoperabilität dieser Module gilt es im folgenden dann automatisiert zu testen.

System Test

Um zu testen, ob nun nicht nur einzelne Teile des Systems gut miteinander funktionieren, sondern auch das ganze System als solches, nutzt man die System-Tests [5, vgl. S. 6]. Die Tests werden auf Grundlage der Spezifikation des Systems erstellt. Es wird davon ausgegangen, dass die einzelnen Module hier wie erwartet funktionieren. Im Vordergrund dieser Testebene steht, dass die Software die Spezifikation, also die Erwartungen an sich, erfüllt.

Akzeptanz Test

Die finale Ebene des Testprozesses ist der Akzeptanz Test. In dieser Ebene wird die Software aus Sicht des Endnutzers geprüft, oft ist der Endnutzer auch Teil dieses Prozesses [5, vgl. S.6]. Ziel ist es, zu verifizieren, dass die Analyse und Umsetzung des Problems erfolgreich ist und der Nutzer mit der entwickelten Lösung zufrieden ist [5, vgl. S.6].

3.5.3 Testabdeckung

Wir kennen nun verschiedene Ebenen und Sichtweisen des Testens. Allerdings ist noch nicht klar, wann ausreichend getestet wurde und ob wir überhaupt genügend testen

können. Hier zu nutzen wir formale Abdeckungskriterien wie sie in [5] eingeführt werden. Sie helfen uns, sinnvolle Testfälle zu entwickeln und zu entscheiden, wann ausreichend getestet wurde. Sollte man nämlich davon ausgehen, dass man einfach alle möglichen Eingaben an eine Funktion testen kann, so kommt man sehr schnell an die Erkenntnis, dass dies unmöglich ist. Als Beispiel sei hier eine simple Addition von 2 64-bit Integer genannt. Für eine komplette Testung dieser simplen Addition gibt es $2^{64} - 18$ Trillionen Kombinationen. Mit einem 3GHz Prozessor wäre eine vollständige Testung nach $2^{64}/3.000.000.000 \approx 6.149.571$ Sekunden (69 Tage) erledigt. Betrachten wir weiterhin einen Java-Compiler so ist der Eingaberaum von Programmen die zum Test stehen effektiv unendlich und somit nicht testbar [5, vgl. 1.3 Coverage Criteria for Testing].

Abdeckungskriterien

Wie gezeigt ist ein "vollständiges Testen also ein ausprobieren aller Möglichkeiten einfach unmöglich. Hierdurch sind wir gezwungen einen anderen Ansatz zu verfolgen. Abdeckungskriterien liefern hierbei einen Ansatz, die einem dabei helfen können, sinnvolle Tests zu entwickeln und zu entscheiden, wann genug Tests entwickelt wurden. Im folgenden sehen wir die Abdeckungskriterien auch als Testanforderungen [5, vgl. 1.3 Coverage Criteria for Testing]

Definition 6 *Eine Testanforderung ist ein spezifisches Element eines Softwareartikates das einen Testfall erfüllen muss. [5, Def. 1.20]*

Dabei soll ein Abdeckungskriterium erfüllt sein, wenn alle seine Testanforderungen erfüllt sind.

Definition 7 *Ein Abdeckungskriterium ist eine Regel oder eine Sammlung von Regeln, die Testanforderungen an eine Menge von Testfällen stellen [5, Def. 1.21]*

Um zu messen, wie gut ein Abdeckungskriterium durch eine Menge an Tests umgesetzt wird, führen wir die Abdeckung ein. Einerseits ist es manchmal sehr schwierig ein Abdeckungskriterium vollständig zu erreichen, andererseits können wir so messen, ob das Kriterium erfüllt wurde [5, vgl. S. 18].

Definition 8 *Gegeben sei eine Menge an Testanforderungen TR für ein Abdeckungskriterium C . Eine Menge an Testfällen T erfüllt C wenn gilt, dass für jede Testanforderung mindestens ein Test existiert der diese Testanforderung erfüllt. [5, vgl. Def. 1.22]*

Es gibt diverse Abdeckungskriterien die auf verschiedenen Annahmen beruhen. Ist das Ziel, dass alle Entscheidungen in einem Programm abgedeckt sein soll (Branch-Coverage), so führt jede Entscheidung zu zwei Testanforderungen, eine Anforderung für den positiven und eine für den negativen Entscheidungsfall [5, vgl. S. 17]. Soll jede Methode mindestens einmal aufgerufen werden (Call-Coverage), so führt jede Methode zu einer Testanforderung um diese Methode abzudecken [5, vgl. S. 17]- Im folgenden Kapitel 4 konzentrieren wir uns auf Abdeckungskriterien für Graphen.

4 Graphüberdeckung

Wie zuvor gesehen, existieren verschiedene Coverage-Kriterien um Testabdeckung zu prüfen. Graphcoverage ist hierbei eine Herangehensweise um graphenbasierte Datenstrukturen zu überdecken. Graphen können nämlich ähnliche Probleme aufweisen wie vorheriges Beispiel der Addition. Die Addition zweier 64-bit Integer ist wenigstens endlich, Graphenstrukturen haben sogar unter Umständen unendliche Testräume. Umso wichtiger ist es hier, dass Überdeckungskriterien formuliert werden können, die diesen möglicherweise unendlichen Suchraum stark verkleinern und dennoch eine ausreichende Testung ermöglichen. Da wir vorher ergründet haben, dass GraphQL sich als gerichteten Graph darstellen lässt, können wir nun die Graphcoverage nutzen, um Tests mithilfe der Graphcoverage zu erstellen. Wie genau die Coverage erstellt wird und daraus Tests resultieren, werden im folgenden geklärt. Gerichtete Graphen sind die Grundlage für viele Coverage-kriterien, wobei die Grundidee hierbei ist, Sachverhalte als Graphen zu modellieren und dann eine ausreichende Überdeckung zu finden. [5, 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 Graphüberdeckung allgemein

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

Menge N von Knoten

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

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

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

[5, 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

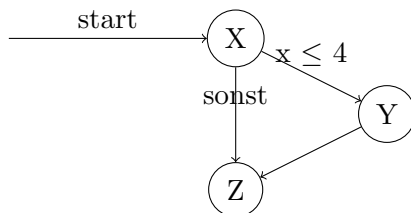
Null, der in einem Knoten N_0 startet und in einem Knoten N_f endet, nennt sich Testpfad [5, vgl. S. 28] Ziel ist es nun mit Hilfe von Coverage-Kriterien Testpfade zu ermitteln die unser Problem ausreichend testen - was hierbei ausreichend ist, ist von Fall zu Fall unterschiedlich.

4.2 Graphüberdeckung Kriterien

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

4.2.1 Node Coverage

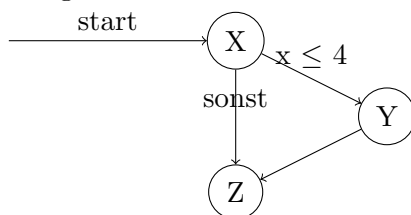
Node-Coverage ist ein Coveragekriterium, dass alle Knoten, die von N_0 erreichbar sind, in einem Graphen abdecken soll. Definieren wir folgenden, sehr einfachen Graphen:



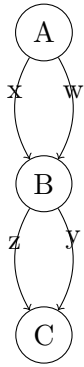
So wäre die Node-Coverage mit einem Test einzigen Test erfüllbar. Dieser ist der Pfad $X \rightarrow Y \rightarrow Z$. Es ist auch denkbar, dass wir zwei Pfade oder mehr nutzen allerdings erfüllt dieser Pfad schon unser Kriterium daher geben wir uns vorerst zufrieden. Wir sehen schnell, dass dieser Ansatz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

4.2.2 Edge-Coverage

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



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



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

$x \rightarrow z$

$w \rightarrow y$

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

4.2.3 Edge-Pair Coverage

Das Edge-Pair Coveragekriterium ist eine Erweiterung der Edge-Coverage, indem hier auch die Beziehungen von einzelnen Kanten untereinander berücksichtigt werden um das zuvor aufgetretene Problem zu lösen. Nach [5, Introduction to Software Testing] ist Edge-Pair Coverage: "Alle erreichbaren Pfade von Länge bis zu 2 im Testgraphen". Ziel dieses Coverage-Kriteriums ist es, dass alle möglichen Kantenpaare abgedeckt sind. Eben definiertes Beispiel hätte mit Edge-Pair Coverage eine Überdeckung mit:

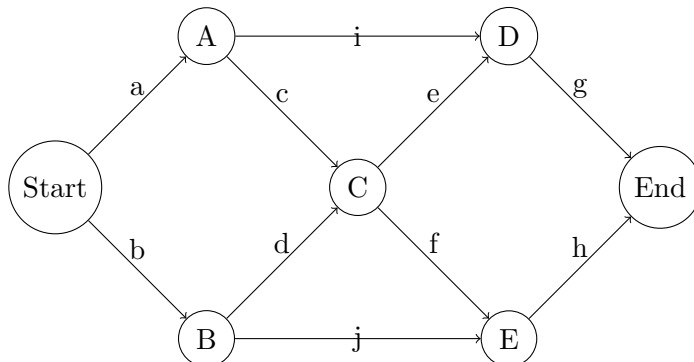
$x \rightarrow z$

$w \rightarrow y$

$x \rightarrow y$

$w \rightarrow z$

Dieses simple Beispiel wird durch Edge-Pair Coverage gut abgedeckt. Edge-Pair Coverage neigt allerdings dazu, extrem große Suchräume zu erzeugen und nur Pfadkombinationen bestimmter Länge zu betrachten. [5, vgl. S. 35] Hierdurch werden bestimmte Kombinationen von Pfaden immer noch nicht berücksichtigt.



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

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

Hierdurch ergeben sich dann diese Testpfade:

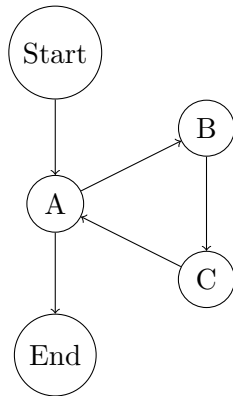
$Start \rightarrow A \rightarrow C \rightarrow D \rightarrow End$
 $Start \rightarrow A \rightarrow C \rightarrow E \rightarrow End$
 $Start \rightarrow B \rightarrow C \rightarrow D \rightarrow End$
 $Start \rightarrow B \rightarrow C \rightarrow E \rightarrow End$

allerdings fehlen auch hier wieder Pfade.

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

4.2.4 Prime-Path Coverage

Die Prime-Path Coverage verlangt, dass jeder (Prime)Primärpfad durch mindestens einen Testpfad abgedeckt sein muss. Ein Primärpfad ist definiert als ein einfacher Pfad, der nicht vollständig als zusammenhängender Teil in einem anderen einfachen Pfad enthalten ist. [5, vgl. S. 35] Hierbei ist ein einfacher Pfad dann ein Pfad, in dem keine Kanten und keine Knoten wiederholt werden, mit Ausnahme möglicherweise des ersten und letzten Knotens (wenn sie gleich sind, handelt es sich um einen Kreis). In diesem Graphen:



Sind dies alle Prime-Paths:

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

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

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

4.2.5 Complete-Path Coverage

Als letztes Coveragekriterium wollen wir die Complete-Path Coverage einführen. Ziel dieses Coveragekriteriums ist es, dass jeder mögliche Pfad mit einem Test abgedeckt werden kann. Wir brauchen dieses Kriterium nicht ausführlich definieren, da in zyklischen Graphen eine Complete-Coverage nicht möglich ist. Kreise in Graphen führen zu einer unendlich großen Menge an Pfaden und wir können keine unendlich große Menge behandeln. Die Complete-Path Coverage ist sinnvoll wenn wir Kreise verbieten und der Testgraph nicht zu groß ist. Um unsere Methode jedoch von [6, 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
Node	Knoten	Keine	Oberflächlich
Edge	Kanten	Minimal	Ein bisschen tiefer
Edge-Pair	Kantenpaare	Überlappung	Tiefer
SimplePath	Pfade ohne Wiederholungen	Keine	Ziemlich tief
PrimePath	Hauptpfade	Kann enthalten	Tief
CompletePath	Alle möglichen Pfade	Höchste	Am tiefsten

Komplexität
 Niedrigste, s
 Mittlere, e
 Erhöhte, e
 Hohe, lar
 Sehr hoch,
 Extrem hoch, s

Tabelle 4.1: Vergleich der Graphabdeckungskriterien

4.3 Graphcoverage für Code

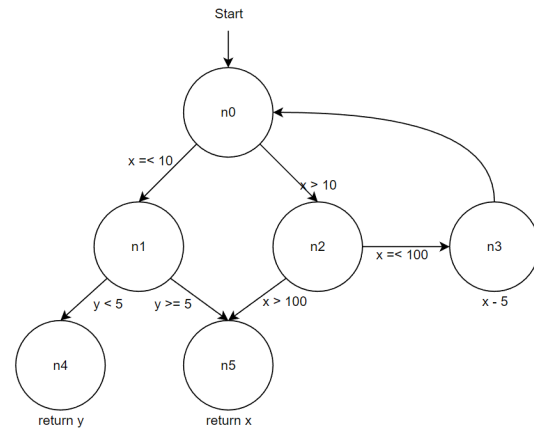
Im Kontext der Testentwicklung erlaubt die Graphcoverage es uns, einen systematischen Ansatz zur Testgenerierung zu verfolgen. Beziehen wir die Graphcoverage auf die Generierung von Tests für Code so müssen wir uns zuerst fragen, wie wir Code als Graphen darstellen können. Da dieser Abschnitt eher der 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, Anfangsknoten und Endknoten sowie

Kanten. Wer mehr über die Umwandlung von Code in Kontrollflussgraphen lernen will, sei an [5, 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 Pfade 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 antrieb, 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* [5] 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

ehosten 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 *Property-based Testing*[6, 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 getestet 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 eliminieren 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 umsetzbar 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 verwandte Arbeiten

Da GraphQL eine stetig wachsende Beliebtheit verzeichnet [17][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" [6] wird der Ansatz des Property-based Testing verfolgt, um Integrationstests zu erstellen. Property-based Testing ist laut dem Paper heute Synonym mit Random Testing" [6][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:

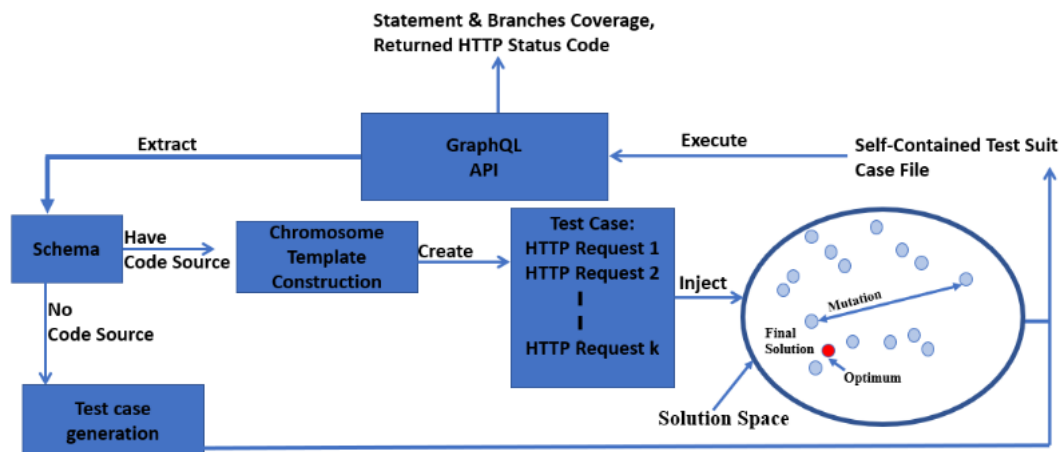
1. Vom Schema, generiere Typ-Spezifikationen
2. Generiere einen Generator der zufällig eine Liste an Query-Objekten erstellen kann
3. Generiere n Querys
4. Transformiere die Queries in GraphQL-Format
5. Führe die Queries auf dem SUT (system under test) aus
6. Evaluiere die Ergebnisse auf ihre Properties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

Insbesondere Punkt 2. und Punkt 6. weisen Verbesserungsbedarf auf. Punkt 2 wird auch der Hauptunterscheidungspunkt beider Arbeiten sein, denn hier sind dann zwei gänzlich unterschiedliche Konzepte am Werk. Dieser ist beim Property-Based Testing nämlich ein Query-Generator der mithilfe der Clojure-Bibliothek Serene[25] Clojure.Specs[26] generiert und diese Clojure.Specs[26] dann nutzt um mit der Clojure-Bibliothek Malli[27] 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[27] 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.[6][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"[6][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 suchenasbasiertes Testen

EvoMaster[7] 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"[28] 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[28]. Im Vergleich mit unserer geplanten Arbeit mittels des Prime-Path-Algorithmus ergeben sich einige Unterschiede, diese sind unter anderem: Nutzung eines evolutionären Algorithmus Many-Independent-Objective (MIO). Im Paper selbst wird davon ausgegangen, dass andere evolutionäre Algorithmen unter Umständen passender wären als der MIO Algorithmus für die Testgenerierung. Jedoch ist ein evolutionärer Algorithmus auch immer ein stochastisch, heuristisch sich dem Optimum annähernder Algorithmus. (Beleg

hierfür) Im Gegensatz dazu ist der Ansatz dieser Arbeit ein iterativer Algorithmus der ideale Überdeckungen auf direkte Art bietet und im ersten Durchlauf direkt sein ideales Ergebnis ermittelt. Die ideale Lösung bezieht sich hierbei auf bestimmte Code-Coverage Kriterien die durch unseren Algorithmus erfüllt werden. Inwiefern der evolutionäre Algorithmus diese Kriterien erfüllt bleibt offen, es ist jedoch davon auszugehen, dass er sich einer idealen Lösung dieser Kriterien nur annähert da er eben ein stochastischer Algorithmus ist. (beleg oder Quelle)

5.3 Deviation Testing

Da GraphQL dynamisch auf Anfragen reagiert und es somit möglich ist, in seiner Anfrage einzelne Felder mit einzubeziehen oder auch auszuschließen ist dies im Grunde genommen ein einzelner Test-Case. Im Paper "Deviation Testing: A Test Case Generation Technique for GraphQL APIs" wird diese Gegebenheit benutzt und aus einer selbstdefinierten Query werden hier einzelne Test-Cases gebildet. Ein solcher Test macht je nach Implementierung der GraphQL-Resolver durchaus Sinn, da im Backend Felder durchaus zusammenhängen können und es Bugs geben kann wenn Resolver fehlerhaft definiert sind. z.B. könnte folgende Definition zu solchen Fehlern führen:

(hier BSP mit Code einfügen)

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

5.4 Query Harvesting

Klassisches Testen von Anwendungen beinhaltet, dass möglichst das komplette System getestet wird bevor es verwendet wird. Im Paper "Harvesting Production GraphQL Queries to Detect Schema Faults" wird ein gänzlich anderer Ansatz verfolgt. Hierbei ist es nicht wichtig die gesamte GraphQL-API vor der Veröffentlichung zu testen sondern echte Queries die in Production ausgeführt werden zu sammeln. Der Ansatz der hierbei verfolgt wird begründet sich daraus, dass ein Testraum für GraphQL potentiell unendlich sein kann und es sehr wahrscheinlich ist, dass nur ein kleiner Teil der API wirklich intensiv genutzt wird, sodass auch nur dieser Teil wirklich stark durch Tests abgedeckt werden muss. AutoGraphQL läuft hierbei in zwei Phasen wobei in der ersten Phase alle einzigartigen Anfragen geloggt werden. In der zweiten Phase werden dann aus den geloggten Anfragen Tests generiert. Hierbei wird für jede geloggte Query genau ein Test-

Case erstellt. Bei dieser Art des Testens wird insbesondere darauf Wert gelegt, dass es keine Fehler im GraphQL Schema gibt. Dies ist ein wichtiger Teil um GraphQL-API's zu testen allerdings noch kein vollständiger Test denn hier wird außer Acht gelassen, dass eine Query konform zum GraphQL-Schema sein kann aber trotzdem falsch indem zum Beispiel falsche Daten zurückgegeben werden durch falsche Referenzierung oder ähnlichem. In dem zu entwickelndem Tool sollten alle Querys die von AutoGraphQL geloggt werden auch berücksichtigt werden da sie durch den Prime-Path Algorithmus auch ermittelt werden. Es kann allerdings sinnvoll sein AutoGraphQL als Monitoring-Software mitlaufen zu lassen und weitere etwaige Fehler hiermit zu loggen und automatisch daraus Test-Cases erstellen zu können damit zukünftig keine Fehler dieser Art mehr passieren.

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

5.6 Andere Arbeiten

Hier ist eine kurze Übersicht über andere Arbeiten, dieses Kapitel ist sehr unwahrscheinlich 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 Unter der Haubefunktioniert

6 Testentwurf

Wie zuvor gesehen kann ein GraphQL-Schema mit Graphcoverage-Kriterien überdeckt werden. Wir wollen diese Gelegenheit nun nutzen um eine Methode zur Generierung von Integrationstests für GraphQL zu entwickeln. Hierbei werden wir uns in einigen Teilen an der Methode von [6, 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 [6, Property-based Testing of GraphQL-APIs]

6.1 Schema in Graph abbilden

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

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

Listing 6.1: Schema-Response

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

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

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

Listing 6.2: Type-Field

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

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

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

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

```

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

```

Listing 6.3: Type-Field

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

```

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

```

Listing 6.4: Type-Field

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

```

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

```

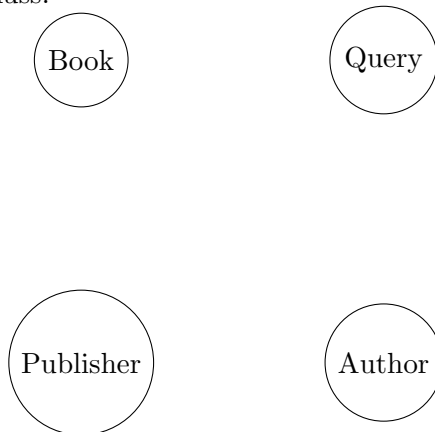
```

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

```

Listing 6.5: Schema Definition

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



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

1

```
{
```

```

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

```

Listing 6.6: book Field

```

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

```



```

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

```

Listing 6.7: author Field

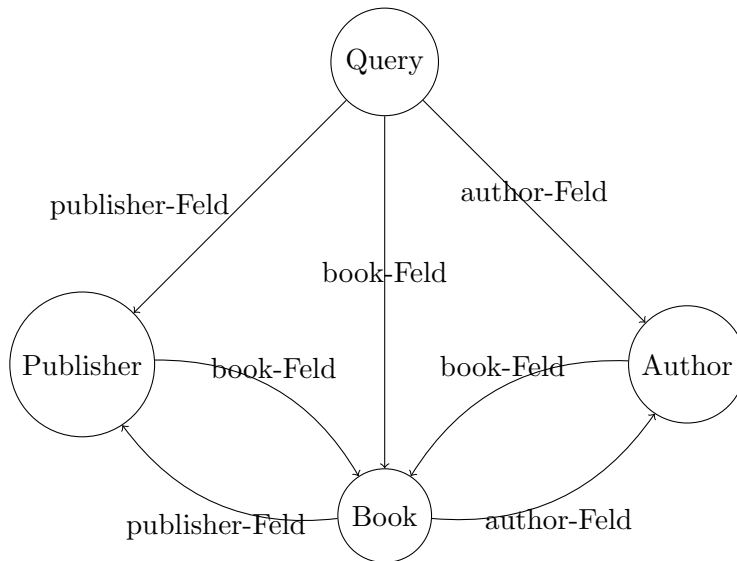
```

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

```

Listing 6.8: publisher Field

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



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

6.2 Pfade aus Graph bilden

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

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

6.3 Coverage-Pfade ermitteln

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

6.3.1 filternder Ansatz

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

```
Input: alle_Pfade
```

```
prime_paths = []
```

```
Für alle_Pfade:
```

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

```
        prime_paths += Pfad
```

```
    Sonst verwerfe Pfad
```

```
return prime_paths
```

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

```
    Für pfad in alle_Pfade:
```

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

```
            return False
```

```
    return True
```

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

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

```
        return False
```

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

```
        Wenn TeilPfad = möglicherPrimePath:
```

```
            return True
```

```
    return False
```

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

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

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

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

6.3.2 generativer Ansatz

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

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

```

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

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

```

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

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

Wir sehen diese sind identisch zum filternden Ansatz.

6.4 Query aus Pfad ermitteln

Da wir nun die Pfade, je nach Coveragekriterium ermittelt haben müssen wir nun eine Query aus diesen Pfaden bilden. Hierbei nutzen wir diverse Hinweise die im GraphQL-Schema gegeben werden und einen Teil der Arbeit aus [6, Property-Based Testing]. Jeder ermittelte Pfad startet im Query-Knoten. Da wir zuvor den Kanten jeweils die Feldattribute mitgegeben haben, können wir nun sehr einfach ermitteln welche Operationen wir ausführen müssen, um von einem Knoten zum nächsten zu kommen. Dies bedeutet,

dass die Kanteninformationen ausreichen, um eine Query zu bilden. Wir gehen hierbei den kompletten Pfad ab und geben einen Query-String zurück der für GraphQL zulässig ist. Eine valide GraphQL-Query beginnt mit `{}` und folgt dann mit der ersten Felddefinition. Eine Felddefinition kann vom Type *SCALAR* oder *OBJECT* sein. Ist ein Feld vom Type *SCALAR* so kann dieses einfach der Query hinzugefügt werden als erwartetes Feld. GraphQLs Spezialität ist es, dass nur wirklich angeforderte Felder in einer Response existieren. Im Kontext des Integrationstesten wollen wir aber alle möglichen Felder abfragen einfach um sicher zu gehen, dass diese korrekt installiert wurden. Es wäre denkbar auch Variationen der einzelnen Felder zu implementieren, dass nicht in jeder Query immer alle Felder eines Typen berücksichtigt werden, dies wird hier jedoch zuerst vernachlässigt und wir inkludieren alle Felder. Ist ein Feld vom Type *OBJECT* so bedeutet dies, dass dieses Feld nur angegeben werden muss, wenn der nachfolgende Knoten im Pfad exakt den Typen dieses Feldes hat! Das Feld wird dann wieder eingeleitet durch `{}`. Innerhalb dieser Felder ist nun wieder jedes Feld des Typen *SCALAR* hinzuzufügen und das Feld des nächsten Knotens zu suchen sowie hinzuzufügen. Sollte ein Feld vom Type *OBJECT* Argumente benötigen, so steht dies im Schema und wir können diese generieren. Input-Argumente können vom Type *INPUT – OBJECT* sein oder *SCALAR*. Sollte es ein Scalar Type sein, so bedienen wir uns der Methode aus [6, 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. Ermittelt ob die Kante (*Query, Book*) Argumente benötigt.
3. Ermittelt alle *SCALAR* Types die *Book* definiert.
4. Ermittelt die Kante von *Book* zu *Publisher*
5. Ermittelt ob die Kante (*Book, Publisher*) Argumente benötigt.
6. Ermittelt alle *SCALAR* Types die *Publisher* definiert. (Pfad zuende daher kein Folge-Typ)
7. Baue Query-String aus den erlangten Informationen

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

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

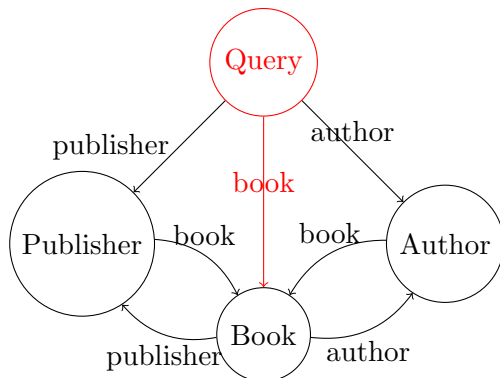


Abbildung 6.1: Schritte 1 - 3

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

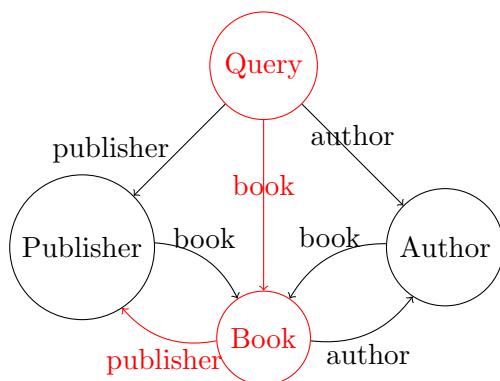
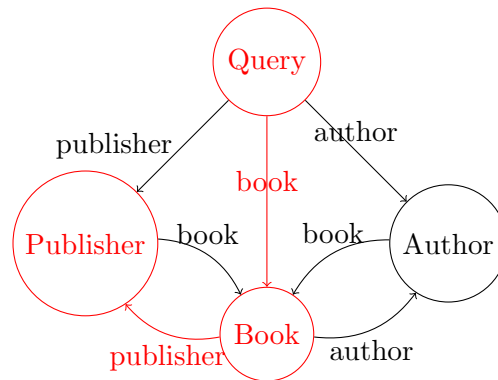


Abbildung 6.2: Schritte 4 - 6

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



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

```

1      {
2      }

```

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

```

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

```

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

```

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

```

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

```

1      {
2          book(id: RandomString){
3              id
4              title
5              publisher {
6                  id          51
7                  name
8              }
9          }
10     }

```

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

Generierung der Argumente für Querys auf [6, Property-based Testing] beziehen ist es unter Umständen möglich, dass Querys Argumente bekommen, die nicht zur zugrundeliegenden Datenstruktur passen. D.h. in der Standardmethode wird für ein zufälligen Integer-Wert direkt der gesamte Integer Wertebereich genutzt. Ist der Datenraum jedoch sehr klein, so kann es sein, dass kein einziger existenter Wert getroffen wird und die Query nur eine leere Antwort wiedergibt da das Element nicht gefunden wurde. Hier kommt es auf die spezielle Implementierung einerseits des SUT an andererseits auch auf die Programmierung der Argumentgeneratoren. Hierzu jedoch im Praxisteil mehr.

6.5 Test ausführen & Testauswertung

Für die Testausführung halten wir uns wieder ähnlich zum [6, 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 [6, Property-based Testing]. Zur Überprüfung der Querys stellen wir diese ganz einfach per HTTP-Post an den zu testenden GraphQL-API Endpunkt. In der Response erhalten wir dann einen StatusCode welcher üblicherweise zwischen 2XX und 5XX liegt. Wir werden nun die einzelnen StatusCodes aufbrechen, da diese inherent wichtig sind für unsere Testauswertung. Responses von 2XX werden in Kapitel *PositiveTests* behandelt, 4XX in Falsch-Negative Tests und 5XX in Negative Tests.

6.5.1 Positive Tests

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

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

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

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

```

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

```

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

```

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

```

Listing 6.9: mangelhafte Response

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

Unterschiede in den Pfadlängen

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

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

Demnach hätte folgende Response eine Tiefe von 2

```

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

```

Listing 6.10: vollständige Response

Und die leere Antwort hätte eine Tiefe von 1

```

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

```

Listing 6.11: mangelhafte Response

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

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

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

Zufallsgeneratoren der Argumente

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

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

Anzahl der Querys

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

6.5.2 Falsch-Negative Tests

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

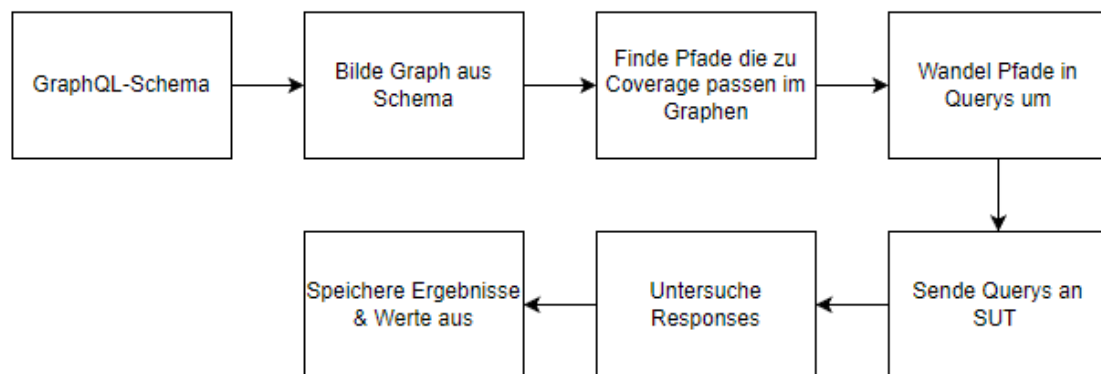
6.5.3 Negative Tests

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

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

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

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



Wie zu sehen, ist der ganz grobe Ablauf ähnlich zum [6, Property-based Testing] allerdings unterscheidet sich die Methode in einigen Teilen sehr stark vom [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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. Voraussetzungen zum Ausführen der Anwendung ist Python und diverse Dritt-Bibliotheken die in der Readme vermerkt sind.

7.1 Tool- / Dependencyauswahl

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

7.1.1 NetworkX

NetworkX ist eine Python-Bibliothek für *Erstellung, Manipulation und Untersuchung der Struktur, Dynamik und Funktionen komplexer Netzwerke* [29, vgl. Startseite] Mit einer Star-Anzahl von 12.8k[30] 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*[31]. Mit *16k*[31] Sternen auf GitHub ist Faker eine noch beliebtere Bibliothek als NetworkX. Faker ist eine Bibliothek die es sehr einfach macht Daten zu generieren. Da wir im Kontext von GraphQL Argumenten nur sehr einfache Datentypen als Argumente benötigen reicht uns diese Bibliothek komplett aus da sie es schafft uns schnell und unkompliziert Daten in genau dem Format zu generieren wie wir sie brauchen. Angenommen wir benötigen einen String der 10 Zeichen lang ist, so reicht eine Zeile:

```

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

```


Selbiges falls wir eine Zufallszahl benötigen zwischen 1 bis 1000

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

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

7.1.3 PyTest

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

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

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

7.2 Umsetzung der Methode

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

7.2.1 Schema in Graph abbilden

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

```

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

```

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

```

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

```

16

```

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

```

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

7.2.2 Pfade aus Graph bilden

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

```

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

```

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

```

1 def generate_prime_paths(startknoten, g):
2     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,
6                             g)
7         if is_prime_path(path, pathList):
            pathList.append(path)

```

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

```

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

```

7.2.3 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* [6] 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. Anschließend 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 getestet 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* [6]. 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         })
5     response_as_dict = json.loads(r.text)
6     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,
3         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
4         ('-', '_') + "(caplog):" + "\n"
5     testString = testString + "    caplog.set_level(logging.
6         WARNING)" + "\n"
7     testString = testString + "    response = requests.post
8         (\"" + url + "\", json={\"query\": \"" +
9         testQueryAsValidString + "\"})" + "\n"
10    testString = testString + "    response_as_dict = json.
11        loads(response.text)" + "\n"
12    testString = testString + "    measurement = queries.
13        compareQueryResults(response_as_dict, \"" + url + "\" +
14        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 etwaige 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

```

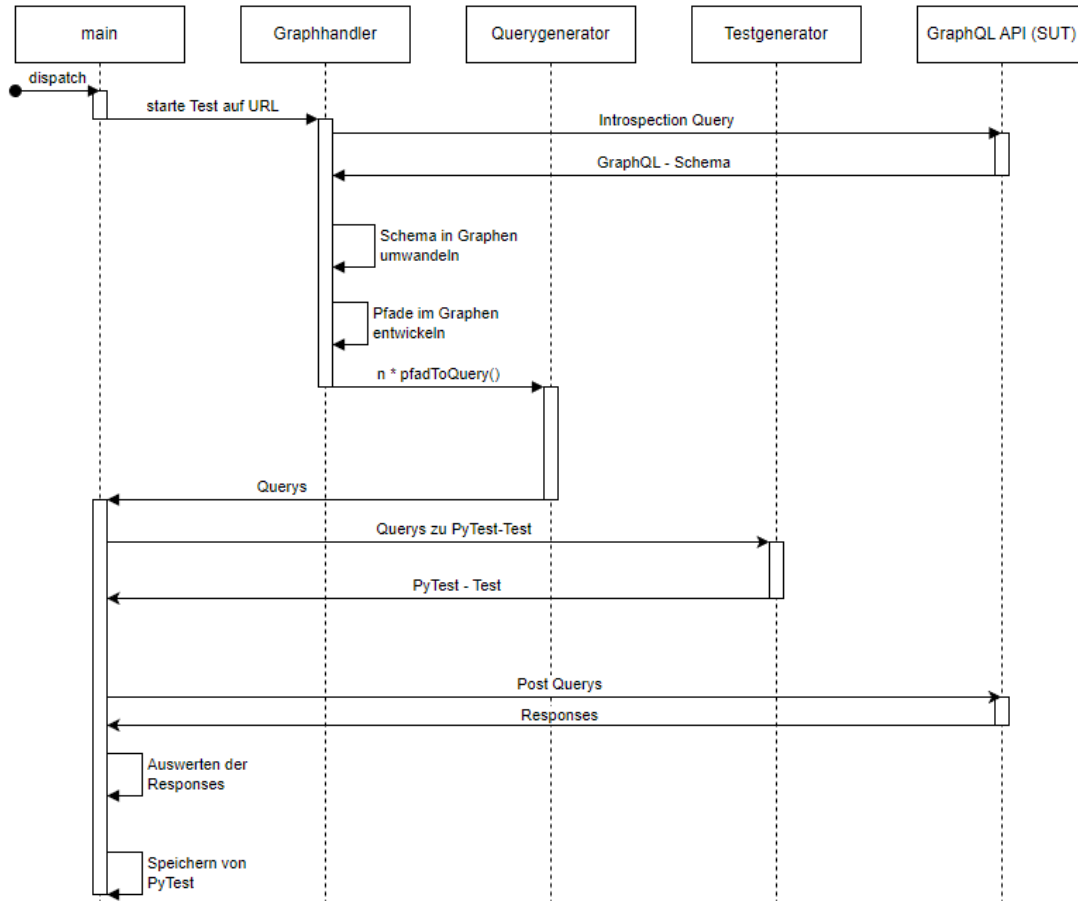
Die Kategorisierung sind *Good_Test*, *Perfect_Test*, *malformed_Test* und *confirmed_failed_Test*. Ein *Good_Test* ist ein Test, der keinen Fehler erzeugt hat allerdings die Pfadlänge von Erwartung != Response. *Perfect_Tests* sind Tests, die keinen Fehler erzeugen und Pfadlänge Erwartung = Response. *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 etwaige 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 Vergleichsmetriken

Bevor wir einen tatsächlichen Vergleich beider Methoden durchführen werden erst einmal die Metriken eingeführt in denen sich Verglichen wird. Hierdurch wird einfacher verständlich welche Punkte miteinander verglichen werden. Wir werden einige neue Metriken einführen aber auch Metriken nutzen die in *Property-based Testing*[6] 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? [6, vgl. RQ1]
2. Wie gut ist die Fehlerfindungskapazität der Methode? [6, 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 [6, 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. [6, vgl. A2] Unser entwickelter Prototyp soll in exakt dem gleichen Umfeld seine Tests generieren. Wir erwarten, dass wir möglichst die selben Fehler finden wie die ursprüngliche Methode und positiv wäre, wenn wir mehr und neue Fehler finden würden. Beide Forschungsfragen werden im folgenden noch einmal näher erläutert da diese ein wenig spezialisiert sind und Wissen über Methode ist wichtig um die Ergebnisse korrekt einordnen zu können.

8.1.2 Fehlerfindungskapazitäten

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

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

Schema Coverage / Schema (Ab)Überdeckung

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

Definition 9 *Für alle Objekte des Schemas: Bilde alle Tupel $\{Object, Field\}$. Ein Schema hat eine ideale Coverage wenn alle Tupel durch einen Test abgedeckt sind.*

Wie in Property-based Testing schon erwähnt: *da keine Coverage Metric für GraphQL Blackbox Test Auswertung existiert, starten wir mit einem sehr einfachen und intuitiven Ansatz* [6, vgl. B. Measuring Schema Coverage]. In der Tat ist das vorgestellte Coveragekriterium ein sehr einfaches Kriterium. Es lässt zum Beispiel die Beziehungen zwischen allen Knoten aus und beachtet nur, dass alle Knoten inbegriffen sind mit allen Feldern. Hiermit entspricht das definierte Coverage-Kriterium einer Kombination aus Edge- und Nodecoverage. Denn alle Knoten müssen abgedeckt sein und alle Kanten ausgehend von den Knoten. Wie zuvor gesehen ist ein solches Kriterium allerdings noch nicht ausreichend für eine ideale Testabdeckung, da zum Beispiel die verschiedenen Kantenkombinationen außer acht gelassen werden und sich somit doch noch Fehler im Code befinden können. Wesentlicher Unterschied beider Methoden ist insbesondere, dass *Property-based Testing* überprüfen muss ob es diese Coverage erreicht. Unsere vorgestellte Methode stellt sicher, dass diese Coverage erreicht ist bevor sie aufhört mit dem generieren. Die Überprüfung der Schema-Coverage in Property-based Testing geschah durch ausprobieren. Hierbei wurde ausprobiert wie viele Testgenerierungen benötigt wurden, um das definierte Kriterium zu erfüllen. Um ein 100% Coverage beim GitLab Schema zu erreichen waren verschiedene Anzahlen an Iterationen nötig bei verschiedenen Rekursionslimits. Eine 100% Coverage wurde bei GitLab nur erreicht wenn 10000 Tests mit Rekursionslimit 4 erstellt wurden. Die Berechnungszeit war hierbei 931 Sekunden. Dies ist zwar der schlechteste Wert in der gesamten Statistik und man könnte meinen, dass der Vergleich nun nicht genau wäre - allerdings ist dies auch der einzige Wert der verlässlich 100% Coverage geliefert hat. Unser Ziel ist es also, weniger als 10.000 Tests und 930 Sekunden zu benötigen um das hier definierte Coveragekriterium zu erfüllen.

8.1.3 Neue Metrik

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

Query sequentiell. Nutzen wir die zufällig generierte Query aus *Property based Testing* Fig. 9[6]. Die Query lautet:

1

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

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

Definition 10 Für alle Querys und dazugehörigen Responses wird die Pfadlänge bestimmt. Eine erfolgreiche Query hat dann zwei mögliche Szenarien:

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

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

8.2 Threats to Validity / Limitierungen

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

8.2.1 Argumentgeneratoren

Wie in 8.1.3 angesprochen ist es wichtig, dass GraphQL für jede Funktionen einen Wert ungleich *null* bekommt, sodass der Pfad weitergegangen werden kann und die Funktionen in diesem getestet werden. Um Bedingungen zu begünstigen werden die Argumentgeneratoren für jedes Experiment angepasst, sodass es sehr viel wahrscheinlich ist, dass die generierten Argumente auch zum SUT passen und die allgemeine Query-Qualität hierdurch besser wird. Es hängt dann explizit davon ab wie sehr die Argumentgeneratoren angepasst werden denn ein einfaches anpassen hat sich *Property-based Testing*[6] auch erlaubt. Hierbei sei zum Beispiel erwähnt, dass eine Type *ID* in GraphQL als String wert definiert ist, häufig in Implementierung jedoch als Zahlenstring genutzt wird. Eine beispielhafte Anpassung wäre hier nun, dass wir den Generator für den Type *ID* so

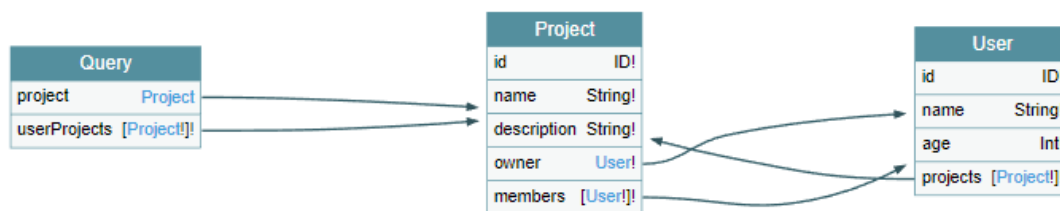
anpassen, dass er nur Argumente für *ID* zurückliefert die ein Zahlenwert sind in einem gewissen Bereich der durch das SUT abgebildet wird.

8.3 Fehlerfindungskapazitäten

Zuerst wollen wir die Fehlerfindungskapazitäten des Testtool auf die Probe stellen. Hierfür nutzen wir die beiden, zuvor benannten, Testsysteme 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*[6] gefunden wurden. Idealerweise wollen wir jedoch sogar mehr Fehler finden.

8.3.1 GraphQL-Toy

Das Testsystem GraphQL-Toy hat ein sehr simples Schema in dem nur drei *OBJECT* Typen existieren. 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:

```
1  const resolvers = {  
2    Query: {
```

```

3         project: (_, {id}, context, info) => {
4             // Example bug 1 - Syntax mistake
5             return db.projects.find(project => project.
6                 id ===);
7         }
8     }

```

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.

```

1     const resolvers = {
2       Query: {
3         project: (_, {id}, context, info) => {
4             // Example bug 5 - wrong type "error"
5             return { ...db.projects.find(project =>
6                 project.id === id), name: ["a", "b"] };
7         }
8     }

```

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 zurückgegeben 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.

```

1     const resolvers = {
2       Query: {
3         project: (_, {id}, context, info) => {
4             // Example bug 3 - Input type validation bug
5             return db.projects[id];
6         }
7     }
8 }

```

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

Mit dem Testtool nach [6, 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 PrimePfad eine PrimePath Coverage erreichen. Hierzu wurden für jeden Pfad 5 Testqueries 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:

1	<pre> { project(id: "2",) { id name description owner { id name age } } } </pre>
1	<pre> { userProjects(id: "1") { name owner { id name age projects { name description id } } } } </pre>

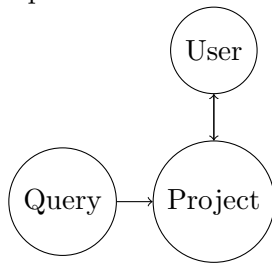
Mithilfe dieser beiden Querys konnte jeder der 11 entdeckten Fehler gefunden werden. Dies liegt auch daran, dass der Argumentgenerator entsprechend angepasst wurde und nur valide IDs produziert hat. So war es sehr wahrscheinlich, dass eine ID die Generiert wird mindestens in einer der 5 erstellten Querys 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* [6, 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 injizierten Bugs wurden von beiden Tools jeweils 11 gefunden. Die injizierten Bugs klassifizierten sich in Syntax-Bug, Validierungs-Bugs, Filtering-Bug und Wrong-Type-Bug. [6, 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.

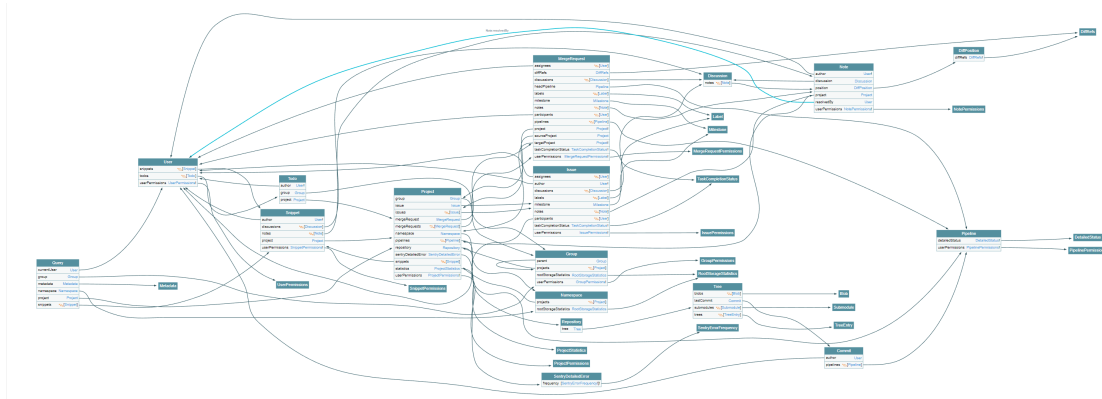
Unser hier entwickelter Prototyp hat jedoch für die allgemeine Fehlerfindung wesentlich weniger Tests benötigt. Das Schema des GraphQL-Toy umfasste einen sehr simplen Graphen.



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 [32].

8.3.3 GitLab

Das Testsystem GitLab wurde schon in Property-based Testing verwandt um an einem Industry-ready Projekt die Methode zu evaluieren. Wir wollen dies mit unserem Testsystem auch durchführen. GitLab stellt sowohl eine REST als auch GraphQL-API zur Kommunikation zur Verfügung. Mit GitLab wird ein komplexes Softwareprodukt zur Versionsverwaltung und DevOps-Anwendung getestet. Die Komplexität dieser Software wird deutlich, wenn wir uns das GraphQL-Schema von GitLab ansehen. Dieses ist sehr komplex und stark zyklisch.



8.3.4 Auswertung GitLab

Das *Property-based Testtool* fand insgesamt 4 Fehler die im Query-Bereich von GraphQL waren. Alle Fehler waren Fehler in der Validierung von Eingabevariablen. Hierbei lag der Fehler darin, dass die Resolver einen fehler verursachten wenn als Eingabe ein String mit leerem Zeichen kam. Dies bedeutet, ein leerer String "" wird richtig behandelt aber ein String mit leerem Zeichen führt zum Fehler: `e\u0000`. Die Fehler wurden gefunden durch folgende Querys:

```
1 {project(fullPath: "root/test-project") {sentryDetailedError
  (id: "") {count}}}
```

Listing 8.1: Fehler 1[33]

```
1 {project(fullPath: "e\u0000") {name fullPath}}
```

Listing 8.2: Fehler 2[34]

```
1 {namespace(fullPath: "e\u0000") {fullName name fullPath}}
```

Listing 8.3: Fehler 3[35]

```
1 {group(fullPath: "e\u0000"){fullName name fullPath }}
```

Listing 8.4: Fehler 4[36]

Alle vom Property-based Tool gefunden Fehler wurden durch unseren Prototypen gefunden mit den Querys

Query 1, Query 2, Query 3 und Query 4.

Getestet wurde auf dem offiziellen GitLab-Docker Image in der Version 12.6.3. Damit im GitLab auch Daten verfügbar sind wurde ein Population-Skript geschrieben, dass im GitLab 50 User anlegt und jedem User einige Projekte, Commit, MergeRequests etc. zuordnet. Das Population-Skript kann im [37, Github] gefunden werden. Da die Query-generierung stets im Query-Knoten beginnt und die PrimePaths gefunden werden sollen,

ergeben sich in diesem Schema sehr viele ähnliche Querys. Diese unterscheiden sich insbesondere am Ende der jeweiligen Query. Der zuvor vorgestellte Algorithmus errechnet für das Schema von GitLab eine Pfadanzahl von 41744 für eine PrimePath-Coverage des Schemas. Eine genaue Auflistung aller Pfade findet sich im [38, GitHub]. Mit der Maßgabe, dass wir pro Pfad 5 Tests erzeugen wollen wurden dann 208.720 Tests erzeugt. Diese Große Anzahl an Tests ergibt sich allerdings auch daraus, dass der Graph der von GraphQL erzeugt wird, einerseits wenige Einstiegspunkte hat. So hat der Query-Type nur 5 ausgehende Felder die für tiefere Pfadbildung relevant sind, 1 Feld hat nur MetaInformationen über die GitLab Software und erzeugt kein weiteren Pfade. Dies bedeutet, dass jeder Pfad auch mit einem dieser Felder beginnt da wir stets unsere Pfade aus dem Query-Type starten lassen müssen. Die zufällige Generierung von Argumenten limitiert hier dann dementsprechend, dass Tests sehr selten unser eingeführtes Measurement erfüllen. In nahezu allen Fällen haben die generierten Tests eine tatsächliche Pfadlänge die kleiner ist als die erwartete. In mehreren Durchläufen zeigte sich, dass im Schnitt nur ca. 20 Tests eine ideale Pfadlänge erreichen. Die Tests, die das erreichen sind im allgemeinen auch Tests, die einen sehr kurzen Pfad abbilden. Hier verringert sich einfach das Risiko, dass Eingabeargumente generiert werden, die keine zugrundeliegenden Daten haben und somit die Pfadausführung verhindern. Ein Beispiel hierfür ist der Pfad *Query- > Project- > MergeRequest- > Time*

```

1 { project(fullPath: "groupx_3/projectx_2_1") {
2   archived
3   avatarUrl
4   containerRegistryEnabled
5   ...
6   mergeRequest(iid: "1") {
7     allowCollaboration
8     createdAt
9     mergeStatus
10    ...
11  }
12 }
13 }
```

durch gutes Mocken des FullPath Argumentgenerators war es möglich, eine Query zu generieren, die somit passende Argumente generiert hat um für eine ideale Testausführung (d.h. Länge Ergebniss = Länge Testpfad) zu sorgen. Generell zeigt sich sehr schnell, dass bei unserer Methode ähnliche Limitierungen wie im Property-based Testing auftreten. So ist eine Anpassung an das Domänenwissen nötig. Für GitLab bedeutet dies unter anderem, dass die ID-Struktur um z.B. Projekte abzufragen in der Form `<user>/<project>` sein muss. [6, vgl. S.8] Da wir vom Prinzip her ähnliche Argumentgeneratoren verwendet haben wie in Property-based Testing, haben wir auch diese Limitierungen. Wir generieren zwar sehr viele Tests die auch PrimePath Coverage ideal erreichen würden. Wenn jedoch die Generierung der Tests auf Zufall basiert, so können wir auch nicht garantieren, dass unsere Eingabeargumente passend sind und ein valides

Ergebniss zurückgeben. Wird kein Ergebniss zurückgegeben, so folgert GraphQL, dass spätere Funktionen nicht ausgeführt werden müssen und somit werden diese auch nicht getestet. Es ist möglich, dieses Problem zu beheben, hierzu jedoch in FutureWork mehr.

8.4 Schema-Abdeckung

Für die Schemaabdeckung wurde im Property-based Testing die Edge-Coverage als zufriedenstellendes Coveragekriterium gewählt. Dieses Coveragekriterium findet allerdings keine Berücksichtigung in der Testgenerierung und ist in der Arbeit lediglich ein Measurement. Mit diesem soll gezeigt werden, wie viele Tests zufällig generiert werden müssen, damit im Durchschnitt eine zufriedenstellende Abdeckung erzielt wird. Dadurch wird die Überlegenheit bereits aus struktureller Sicht deutlich. Während im Property-based Testing nicht garantiert wird, dass eine zufriedenstellende Coverage erreicht wird, sichern wir durch unseren Generierungsalgorithmus zu, dass die generierten Tests der PrimePath-Coverage entsprechen. Dies stellt ein stärkeres Coveragekriterium dar als die Nodecoverage. Dennoch möchten wir untersuchen, wie gut die Coverage in der Praxis funktioniert. Dafür betrachten wir erneut die beiden zuvor benutzten Beispiele: einerseits das GraphQL-Toy als minimales Beispiel und andererseits GitLab als komplexes Beispiel. Wir betrachten die Abdeckung des Schemas eher auf theoretischer Seite. Die Limitierung der zufälligen Argumentgeneratoren behindern eine tatsächliche Umsetzung der hier vorgestellten Methode. Allerdings lösen sich diese auf, sobald das Problem der Argumentgeneratoren gelöst ist.

8.4.1 GraphQL-Toy Schema Coverage

Wie zuvor gesehen in 8.3.1, ist das Schema des GraphQL-Toys ein sehr simples. 3 Knoten, 5 Kanten bilden den ganzen Graphen. Das Property-based Tool nutzt zur Überdeckung des Graphens die Edge-Coverage. [6, vgl. Measuring Schema Coverage] Wie eingeführt in *Property-based Testing*[6] muss für eine zufriedenstellende Coverage die Tatsache erfüllt sein, dass jedes paar von (Type, objectField) abgedeckt sein. Dies bedeutet für das Schema, dass die Felder

- (Query, project)
- (Query, userProject)
- (Project, owner)
- (Project, members)
- (User, projects)

abgedeckt sein müssen um Edge-Coverage zu erfüllen. Da nur die beiden initialen Felder aus dem Query-Type Eingabeargumente benötigen ist die Querygenerierung relativ simpel. Wir können keine Aussage darüber machen ob die generierten Querys von

Property-based Testing [6] diese Coverage erfüllen denn bei der Querygenerierung spielt es keine Rolle ob dieses Measurement erreicht wird. Es gibt lediglich eine Messung die zeigt, dass der Prototyp mit hinreichender Wahrscheinlichkeit in der Lage ist, durch zufällige Querygenerierung Tests zu generieren, die Edge-Coverage erfüllen. [6, vgl. D.Results RQ1]. Im Gegensatz zum Property-based Testing hat der hier entwickelte Prototyp den Vorteil, dass die Pfadgenerierung nicht zufällig ist. Wir berechnen PrimePaths, diese sind ein stärkeres Coveragekriterium als die Edge-Coverage. Dadurch ergibt sich, dass die Tests die vom hier entwickelten Prototyp erstellt werden, stets diese Coverage erfüllen. Die Coverage muss nun also nicht mehr durch hinreichend viele Tests sichergestellt werden. Ein einziger Durchlauf reicht aus um sicherzustellen, dass die gewünschte Coverage erreicht ist. Natürlich bleibt offen ob die generierten Tests diese Coverage tatsächlich erreichen jedoch ist auch ein Problem im Property-based Testing. Dort wird auch nur geprüft, ob die Felder in der Anfrage existieren. Um dies eben messbar zu machen haben wir in 8.1.3 eine neue Metrik eingeführt die überprüfen soll ob die erwartete Coverage der Query mit der tatsächlichen Übereinstimmt.

8.4.2 GitLab Schema Coverage

Das GitLab Schema ist wesentlich komplexer. Im Gegensatz zum GraphQL Toy besteht das GitLab Schema aus 37 Knoten welche jeweils zahlreiche Kanten hinzufügen. Generell lässt sich sagen, dass das Schema sehr komplex und stark rekursiv ist. [6, vgl. Studied Cases 2] Da der Property-based Testing Ansatz ein Rekursionslimit benötigt stellt sich hier die Frage inwiefern überhaupt das Schema überdeckt werden kann. Laut Paper hat sich ein Rekursionslimit von 4 als hinreichend ausgezeichnet [6, vgl. Table 1] und wurde auch so im Code übernommen. Ein Rekursionslimit von 4 bedeutet, dass die maximale zu erreichende Pfadlänge des Testpfades 4 ist. Da das GitLab-Schema aber nun einen Graphen aufspannt, der durchaus wesentlich längere Pfade als 4 hat, ist es fragwürdig wie die 100% Schema-Coverage in [6, Table 1] berechnet wurden. Es seien hier einige Pfade beispielhaft genannt, die einzigartig sind, bei denen sich keine Kante doppelt und deren Länge 4 stark überschreitet:

```
Query -> User -> SnippetConnection -> SnippetEdge -> Snippet -> DiscussionConnection -> Discussion
Query -> Project -> Issue -> DiscussionConnection -> DiscussionEdge -> Discussion -> Note
Query -> Namespace -> ProjectConnection -> Project -> MergeRequestConnection -> MergeRequest
```

Somit gilt, dass in Property-based Testing der Graph des GraphQL-Schemas zum Coverage-Measurement beschnitten wurde, dass nur noch zum jeweils passenden Rekursionslimit die Pfadlänge berücksichtigt wird. Im Grunde lässt sich sagen, dass die genannten 100% Schema Coverage bedeuten, dass alle erreichbaren Felder mit Pfadlänge = Rekursionslimit in den Tests berücksichtigt sind. Dies ist ein sehr großer, struktureller Einschnitt und die in Property-based Testing genannten 100% Edge-Coverage sind keine 100% Edge-Coverage auf dem Schema im eigentlichen Sinne so wie es hier erarbeitet wurde. Die genannten 100% Coverage sind eine Edge-Coverage für einen reduzierten Graphen der auf Grundlage der Rekursionstiefe reduziert wird. Wie auch zuvor erwähnt, er-

zeugt unser hier entwickelter Prototyp Tests die in unserem Fall die PrimePath-Coverage umsetzen. Da PrimePath-Coverage ein stärkeres Coverage-Kriterium als Edge-Coverage darstellt, haben wir hier einen strukturellen Vorteil der uns garantiert, dass die Coverage besser ist. Außerdem reduzieren wir unseren Graphen nicht. Wir führen die Pfadgenerierung auf dem gesamten Graphen aus und erhalten, wie zuvor erwähnt, über 40.000 Pfade zurück die nötig sind um eine PrimePath-Coverage für das GitLab Schema zu erreichen. Hier zeigt sich auch ein direkter Unterschied. Während in Property-based Testing gesagt wird, dass 10.000 Tests mit einem Rekursionslimit von 4 ausreichen um ein 100% Edge-Coverage zu erreichen [6, vgl. Table 1] so sehen wir, dass 10.000 Tests nicht reichen können wenn allein schon über 40.000 PrimePaths existieren. Insbesondere sei hierbei angemerkt, dass die Pfad- & Testgenerierung auf dem GitLab-Schema keine alzu komplexe Aufgabe war. Die Berechnung der Querys geschah auf einem hardwaretechnischen ähnlichem Level wie in Property-based Testing verwandt. [6, vgl. Experimental Setup]. Hier wurde die Aussage getroffen, dass ein [6, Tiefensuchen Ansatz nicht skaliert und deswegen ein iterativer Ansatz zu präferieren ist]. Der hier entwickelte Prototyp zeigt das Gegenteil.

8.5 Zusammenfassung der Experimente

Wir konnten in beiden Experimenten zeigen, dass unser Prototyp die selben Fehler findet wie der Prototyp aus Property-based Testing. Einige Verbesserungen konnten wir eingehen aber wir leiden auch unter ähnlichen Problem wie Property-based Testing. Durch unseren Ansatz ist es möglich geworden, die Schema-Coverage der Tests nachweisbar zu erhöhen und wesentlich verlässlicher zu machen. So ist im Property-based Ansatz immer ein gewisser Zufall entscheidend wie gut die Tests sind. Die Verlässlichkeit der Abdeckung mit Tests ist hier nicht gegeben. Hier konnten wir mit unserer neu entwickelten Methode Fortschritte erzielen indem wir zeigten, dass es möglich ist aus einem GraphQL Schema einen Graphen zu bilden, auf diesem Coverage Algorithmen laufen zu lassen und aus den gewonnen Tests-Pfade zu ermitteln. Somit kann der Ansatz des Property-based Testings mit unserer Methode erweitert werden und die Testcoverage verlässlich und nachweisbar erhöhen. Limitierungen unseres Prototypen bestehen genau wie im Property-based Testing noch. Hierbei generiert unser Tool sehr komplexe und präzise Querys die das System gut testen würden. Allerdings werden Argumente nur zufällig generiert und verhindern somit oft, dass diese komplexen Querys vollständig ausgeführt werden. Einerseits kann dieses Problem auf manuelle Art und Weise verbessert werden indem man die Argumentengeneratoren anpasst und diesen hilft indem man z.B. ID-Generierungen anpasst. Andererseits lässt sich dieses Problem wahrscheinlich auch lösen wenn man sich vom BlackBox-Testing lösen kann und die Methode weiterentwickelt, sodass aus dieser ein Grey oder WhiteBox Testing wird. Hierzu jedoch in futureWork mehr.

9 Zukünftige Arbeit

Einige bereits angesprochene Punkte bieten Möglichkeiten für eine Erweiterung der hier geleisteten Arbeit. Während hier ein erheblicher Beitrag zur theoretischen Testabdeckung von GraphQL-APIs geleistet wurde, so ist nicht garantiert, dass die entwickelten Tests tatsächlich die ermittelte Abdeckung erreichen. Dies folgert sich aus der zufälligen Argumentgenerierung in den einzelnen Querys. Ziel ist es nun, den Zufall möglichst weit zu begrenzen oder aber die erlangten Ergebnisse intelligenter zu nutzen. Wir wollen im folgenden zwei Ansätze vorstellen, die eine praktische Testausführung zuverlässiger und präziser machen können.

9.1 BlackBox-Testing in WhiteBox-Testing umwandeln

Die bisherige Testgenerierung verfolgt den BlackBox-Ansatz bzw. im experimentellen Teil den GreyBox-Ansatz. Das Testsystem hat im ursprünglichen Sinn keinerlei Informationen über das SUT. Im experimentellen Ansatz haben wir den BlackBox Ansatz ein wenig abgeschwächt und zu einem GreyBox-Ansatz verändert, indem wir die Argumentgeneratoren an das jeweilige SUT angepasst haben, sodass die zufällige Argumentgenerierung mit höherer Wahrscheinlichkeit ein Argument liefert, dass dem Test eine bessere, tatsächliche Abdeckung liefert. Idealerweise wäre nun, dass die Testgenerierung auf einem WhiteBox-Ansatz fußt. Hierdurch ist spezifisches Domänenwissen über das SUT vorhanden. Insbesondere die zugrundeliegende Datenstruktur, Programmcode etc. Durch einen White-Box Ansatz wäre es nun möglich, die Argumentgeneratoren automatisch anzupassen, sodass sich diese am Schema und den zugrunde liegenden Daten orientieren. Außerdem wäre eine Code-Analyse möglich die dazu führen kann, dass Testcases noch präziser und exakter Fehler finden. Die von unserem Prototypen generierten Pfade könnten hierbei weiterhin als Basis dienen um eine gute Testabdeckung beziehungsweise Pfadabdeckung sicherzustellen. Mithilfe von optimierten Argumentgeneratoren ist es nun möglich unsere in 8.1.3 eingeführte Metrik weiter an die 100% zu bringen.

9.2 Adaptive Generierung

Die aktuelle Testgenerierung geschieht in einzelnen Phasen. Es werden erst aus einem Graphen die Pfade generiert, hieraus werden Tests erzeugt und diese werden dann an das SUT gestellt und ausgewertet. Alle generierten Tests werden dabei allerdings unabhängig voneinander generiert. Im Sinne einer besseren realen Testabdeckung wäre es jedoch wünschenswert, Tests, bei denen die Argumente gut generiert wurden in weitere Testgenerierung einzubeziehen. Eine Aufteilung wäre hierbei, dass die zu testenden

Pfade sich allmählich weiterentwickeln und somit die Testabdeckung erhöht wird. Ein allgemeiner Ablauf für ein solches Programm könnte wie folgt aussehen:

HIER SCHAUBILD TODO

Mithilfe einer solchen Generierung werden die ermittelten Pfade allmählich abgedeckt indem die Tests erst länger entwickelt werden wenn zuvor passende Argumente generiert wurden die eine tatsächliche Testausführung sicherstellen. Hierbei ist in jedem Generierungsabschnitt denkbar, dass sowohl passende als auch fehlerhafte Argumente generiert werden. In Kombination mit White-Box Testing ist auch eine Code-Analyse von Argumenthandlern denkbar.

10 Fazit

Unsere Behauptung, dass wir die Methode des *Property-based Testings* um einen besseren Pfadfindungsalgorithmus als ein simples, zufälliges, begrenztes raten verbessern können hat sich als wahr herausgestellt. Indem wir einen theoretischen Rahmen geschaffen haben, der zeigt, dass Graphkriterien, insbesondere Graphcoveragekriterien, auf ein GraphQL-Schema anwendbar sind, konnten wir eine Methode entwickeln, die für die Testentwicklung hinreichend ist. Hierbei haben wir eigene Teile eingebracht aber auch Methoden des *Property-based Testings* verwandt, wie zum Beispiel die Argumentgenerierung. Final entstand dann daraus ein Prototyp, der fähig ist, Fehler in GraphQL-APIs zu finden. Dies wurde an zwei Beispielen gezeigt und nachgewiesen, indem wir Fehler finden konnten.

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"],
```

```

34     },
35     {
36         id: "2",
37         name: "Project 2",
38         description: "Not an awesome project!",
39         owner: "200",
40         members: ["200"],
41     },
42 ],
43 users: [
44     {
45         id: "100",
46         name: "Burt",
47         age: 23,
48         projects: ["1", "2"],
49     },
50     {
51         id: "200",
52         name: "Earnie",
53         age: 32,
54         projects: ["2"],
55     },
56 ],
57 };
58
59 const resolvers = {
60   Query: {
61     project: (_, {id}, context, info) => {
62
63       // Example bug 1 - Syntax mistake
64       // return db.projects.find(project => project.id
65         ===);
66
67       // Example bug 2 - Give "foo", input validation
68       // return db.projects[parseInt(id)];
69
70       // Example bug 3 - Input type validation bug
71       //return db.projects[id];
72
73       // Example bug 4 - Using the wrong field
74       // return db.projects.find(project => project.
75         name === id);
76
77       // Example bug 5 - wrong type "error"

```

```

76         // return { ...db.projects.find(project =>
77             project.id === id), name: ["a", "b"] };
78
79         // Example bug 6 - IndexOutOfBounds
80         // return db.projects[parseInt(id)];
81
82         // Correct implementation
83         return db.projects.find(project => project.id
84             === id);
85     },
86     userProjects: (_, {id}, context, info) => {
87         const user = db.users.find(user => user.id ===
88             id);
89
90         // Example bug 1 - Syntax Error
91         // return db.projects.filter(project => user.
92             projects.includes());
93
94         // Example bug 2 - Using the wrong field
95         // return db.projects.filter(project => user.
96             projects.includes(project.name));
97
98         // Example bug 3 - wrong type "errors"
99         // return db.projects.filter(project => user.
100             projects.includes());
101
102         // Correct implementation
103         return db.projects.filter(project => user.
104             projects.includes(project.id));
105     },
106 },
107 Project: {
108     owner: (project) => {
109         // Example bug 1 - Syntax mistake
110         // return db.users.find(user => user.id ===);
111
112         // Example bug 2 - Using the wrong field
113         // return db.users.find(user => user.name ===
114             project.owner);
115
116         // Example bug 3 - wrong type "error"
117         // return { ...db.users.find(user => user.id ===
118             project.owner), name: ["a", "b"] };

```

```

111         // Correct implementation
112         return db.users.find(user => user.id === project
113             .owner);
114     },
115     members: (project) => {
116         // Example bug 1 - logic error
117         // return db.users.filter(user => project.
118             members.includes());
119
120         // Example bug 2 - Using the wrong field
121         // return db.users.filter(user => project.
122             members.includes(user.name));
123
124         // Example bug 3 - wrong type "errors"
125         // return db.users.filter(user => project.
126             members.includes);
127
128         // Correct implementation
129         return db.users.filter(user => project.members.
130             includes(user.id));
131     },
132 },
133 User: {
134     projects: (user) => {
135         // Example bug 1 - Syntax Error
136         // return db.projects.filter(project => user.
137             projects.includes());
138
139         // Example bug 2 - Using the wrong field
140         return db.projects.filter(project => user.
141             projects.includes(project.name));
142
143         // Example bug 3 - wrong type "errors"
144         // return db.projects.filter(project => user.
145             projects.includes);
146
147         // Correct implementation
148         return db.projects.filter(project => user.
149             projects.includes(project.id));
150     },
151 },
152 };
153
154 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     }
```

```

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

```

```

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

```

minimale Schema Response

```

1  {
2      "data": {
3          "__schema": {
4              "queryType": {
5                  "name": "Query"
6              },
7              "mutationType": null,
8              "subscriptionType": null,
9              "types": [
10                 {
11                     "kind": "OBJECT",
12                     "name": "Query",
13                     "description": null,
14                     "fields": [
15                         {
16                             "name": "book",
17                             "description": null,
18                             "args": [
19

```

```

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

```



```

64         "args": [
65             {
66                 "name": "id",
67                 "description": null,
68                 "type": {
69                     "kind": "SCALAR",
70                     "name": "ID",
71                     "ofType": null
72                 },
73                 "defaultValue": null
74             }
75         ],
76         "type": {
77             "kind": "OBJECT",
78             "name": "Publisher",
79             "ofType": null
80         },
81         "isDeprecated": false,
82         "deprecationReason": null
83     }
84 ],
85 "inputFields": null,
86 "interfaces": [],
87 "enumValues": null,
88 "possibleTypes": null
89 },
90 {
91     "kind": "OBJECT",
92     "name": "Book",
93     "description": null,
94     "fields": [
95         {
96             "name": "id",
97             "description": null,
98             "args": [],
99             "type": {
100                 "kind": "SCALAR",
101                 "name": "ID",
102                 "ofType": null
103             },
104             "isDeprecated": false,
105             "deprecationReason": null
106         },
107     ]

```

```

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

```

```

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

```

```

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

```

```

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

```

Query1

```

1  {
2    group(fullPath: "e\u0000") {
3      avatarUrl
4      description
5      descriptionHtml
6      fullName
7      fullPath
8      id
9      lfsEnabled
10     name
11     path
12     requestAccessEnabled
13     visibility
14     webUrl
15     projects(
16       includeSubgroups: false,
17       after: "CXPnWOYLTu0jSbbwJqqY",
18       before: "CrGVlZBseDurRlgzEbtU",
19       first: 2522,
20       last: 3011
21     ) {
22       nodes {
23         archived

```

```

24     avatarUrl
25     containerRegistryEnabled
26     createdAt
27     description
28     descriptionHtml
29     forksCount
30     fullPath
31     httpUrlToRepo
32     id
33     importStatus
34     issuesEnabled
35     jobsEnabled
36     lastActivityAt
37     lfsEnabled
38     mergeRequestsEnabled
39     mergeRequestsFfOnlyEnabled
40     name
41     nameWithNamespace
42     onlyAllowMergeIfAllDiscussionsAreResolved
43     onlyAllowMergeIfPipelineSucceeds
44     openIssuesCount
45     path
46     printingMergeRequestLinkEnabled
47     publicJobs
48     removeSourceBranchAfterMerge
49     requestAccessEnabled
50     sharedRunnersEnabled
51     snippetsEnabled
52     sshUrlToRepo
53     starCount
54     tagList
55     visibility
56     webUrl
57     wikiEnabled
58     issues(
59         iid: "ijrhRHNqHyAjlpaJknYi",
60         iids: "oSTpjUyfHvXKPFvrnNAK",
61         state: closed,
62         labelName: "NGJwsiFoOWbPIDPCEOPS",
63         createdBefore: "2023-07-19T22:26:43",
64         createdAfter: "2023-04-15T21:00:50",
65         updatedBefore: "2023-07-10T11:48:12",
66         updatedAfter: "2023-02-24T21:34:52",
67         closedBefore: "2023-04-23T17:41:34",

```

```

68         closedAfter: "2023-01-28T01:02:27",
69         search: "ZnbnsbggmMpWkRkQfZDT",
70         sort: DUE_DATE_DESC,
71         after: "onP0zYJlSSdeVODfTFZd",
72         before: "crsihXiPwGbeXvUzAHCM",
73         first: 7038,
74         last: 5497
75     ) {
76         nodes {
77             closedAt
78             confidential
79             createdAt
80             description
81             descriptionHtml
82             discussionLocked
83             downvotes
84             dueDate
85             iid
86             reference
87             relativePosition
88             subscribed
89             timeEstimate
90             title
91             titleHtml
92             totalTimeSpent
93             updatedAt
94             upvotes
95             userNotesCount
96             webPath
97             webUrl
98             notes(
99                 after: "EdfPOYFMhnmaRwXCdIXk",
100                 before: "wMdIUdZSYCYGUHShEdSY",
101                 first: 7363,
102                 last: 8871
103             ) {
104                 edges {
105                     cursor
106                     node {
107                         body
108                         bodyHtml
109                         createdAt
110                         id
111                         resolvable

```

```

112         resolvedAt
113         system
114         updatedAt
115         createdAt
116     }
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 }

```

Query2

```

1 {
2   namespace(fullPath: "e\u0000") {
3     description
4     descriptionHtml
5     fullName
6     fullPath
7     id
8     lfsEnabled
9     name
10    path
11    requestAccessEnabled
12    visibility
13    projects(
14      includeSubgroups: true,
15      after: "LzGUwINJWDSdeHYamsoy",
16      before: "VdRsarUsfXODNvMdrBWx",
17      first: 6482,
18      last: 3087
19    ) {
20      nodes {
21        archived
22        avatarUrl
23        containerRegistryEnabled
24        createdAt
25        description
26        descriptionHtml

```



```

27     forksCount
28     fullPath
29     httpUrlToRepo
30     id
31     importStatus
32     issuesEnabled
33     jobsEnabled
34     lastActivityAt
35     lfsEnabled
36     mergeRequestsEnabled
37     mergeRequestsFfOnlyEnabled
38     name
39     nameWithNamespace
40     onlyAllowMergeIfAllDiscussionsAreResolved
41     onlyAllowMergeIfPipelineSucceeds
42     openIssuesCount
43     path
44     printingMergeRequestLinkEnabled
45     publicJobs
46     removeSourceBranchAfterMerge
47     requestAccessEnabled
48     sharedRunnersEnabled
49     snippetsEnabled
50     sshUrlToRepo
51     starCount
52     tagList
53     visibility
54     webUrl
55     wikiEnabled
56     repository {
57         empty
58         exists
59         rootRef
60         tree(
61             path: "XNQtaUctgSGfziijhXQv",
62             ref: "zCYbJRMCKGotQJmpBcZa",
63             recursive: true
64         ) {
65             lastCommit {
66                 authorName
67                 authoredDate
68                 description
69                 id
70                 message

```

```

71         sha
72         signatureHtml
73         title
74         webUrl
75         author {
76             avatarUrl
77             name
78             username
79             webUrl
80             snippets(
81                 ids: ["e\u0000", "", "TEST", "2"],
82                 visibility: private,
83                 type: project,
84                 after: "gVweJjzT0ptsXXmrvTtA",
85                 before: "uSKONOP1LKFpiKNhBIyN",
86                 first: 3982,
87                 last: 2317
88             ) {
89                 pageInfo {
90                     endCursor
91                     hasNextPage
92                     hasPreviousPage
93                     startCursor
94                 }
95             }
96         }
97     }
98 }
99 }
100 }
101 }
102 }
103 }

```

Query 3

```

1 {
2   project(fullPath: "e\u0000") {
3     archived
4     avatarUrl
5     containerRegistryEnabled
6     createdAt

```

```

7      description
8      descriptionHtml
9      forksCount
10     fullPath
11     httpUrlToRepo
12     id
13     importStatus
14     issuesEnabled
15     jobsEnabled
16     lastActivityAt
17     lfsEnabled
18     mergeRequestsEnabled
19     mergeRequestsFfOnlyEnabled
20     name
21     nameWithNamespace
22     onlyAllowMergeIfAllDiscussionsAreResolved
23     onlyAllowMergeIfPipelineSucceeds
24     openIssuesCount
25     path
26     printingMergeRequestLinkEnabled
27     publicJobs
28     removeSourceBranchAfterMerge
29     requestAccessEnabled
30     sharedRunnersEnabled
31     snippetsEnabled
32     sshUrlToRepo
33     starCount
34     tagList
35     visibility
36     webUrl
37     wikiEnabled
38     snippets(
39         ids: ["e\u0000", "e\u0000", "e\u0000", "2"],
40         visibility: private,
41         after: "GCGqcYIymVfIjZJrghZx",
42         before: "jbZllxYHDwZVZKgtDtBQ",
43         first: 7612,
44         last: 3392
45     ) {
46         edges {
47             cursor
48             node {
49                 content
50                 createdAt

```

```

51     description
52     descriptionHtml
53     fileName
54     id
55     rawUrl
56     title
57     updatedAt
58     webUrl
59     discussions(
60         after: "XwwfgUTrYgPYJwgMBmb",
61         before: "oUeZWQMPwSveCHQrBEts",
62         first: 8210,
63         last: 974
64     ) {
65         edges {
66             cursor
67             node {
68                 createdAt
69                 id
70                 replyId
71                 notes(
72                     after: "GfzdyMEZcMMPkByUduYi",
73                     before: "MMqCqCBXDTYiNgFgERvS",
74                     first: 95,
75                     last: 5742
76                 ) {
77                     nodes {
78                         body
79                         bodyHtml
80                         createdAt
81                         id
82                         resolvable
83                         resolvedAt
84                         system
85                         updatedAt
86                         position {
87                             filePath
88                             height
89                             newLine
90                             newPath
91                             oldLine
92                             oldPath
93                             width
94                             x

```

```

95         y
96         positionType
97     }
98 }
99 }
100 }
101 }
102 }
103 }
104 }
105 }
106 }
107 }

```

Query 4

```

1  {
2      project(fullPath: "groupx_3/projectx_32_6") {
3          archived
4          avatarUrl
5          containerRegistryEnabled
6          createdAt
7          description
8          descriptionHtml
9          forksCount
10         fullPath
11         httpUrlToRepo
12         id
13         importStatus
14         issuesEnabled
15         jobsEnabled
16         lastActivityAt
17         lfsEnabled
18         mergeRequestsEnabled
19         mergeRequestsFfOnlyEnabled
20         name
21         nameWithNamespace
22         onlyAllowMergeIfAllDiscussionsAreResolved
23         onlyAllowMergeIfPipelineSucceeds
24         openIssuesCount
25         path
26         printingMergeRequestLinkEnabled

```

```

27 publicJobs
28 removeSourceBranchAfterMerge
29 requestAccessEnabled
30 sharedRunnersEnabled
31 snippetsEnabled
32 sshUrlToRepo
33 starCount
34 tagList
35 visibility
36 webUrl
37 wikiEnabled
38 issues(
39     iid: "ksXuZMlxdVxz1AaqAjrf",
40     iids: "NNAPRcqvnZwDucszkDnh",
41     state: locked,
42     labelName: "UfeLauqqxLVxuylCFelM",
43     createdBefore: "2023-06-30T19:05:54",
44     createdAfter: "2023-07-10T04:47:34",
45     updatedBefore: "2023-05-14T23:47:22",
46     updatedAfter: "2022-09-13T21:09:10",
47     closedBefore: "2023-04-28T02:51:59",
48     closedAfter: "2022-09-21T20:23:50",
49     search: "vAlPWbLSJmlVURpSjmwp",
50     sort: created_asc,
51     after: "ZuiCQgLYmENZJKKScvzv",
52     before: "CKMgjxLkrigiXrEPsCP0",
53     first: 1092,
54     last: 7980
55 ) {
56     nodes {
57         closedAt
58         confidential
59         createdAt
60         description
61         descriptionHtml
62         discussionLocked
63         downvotes
64         dueDate
65         iid
66         reference
67         relativePosition
68         subscribed
69         timeEstimate
70         title

```

```

71     titleHtml
72     totalTimeSpent
73     updatedAt
74     upvotes
75     userNotesCount
76     webPath
77     webUrl
78     discussions(
79         after: "dDa0gakGAttuToCxHVCh",
80         before: "GcPWhODVHIJXhRgyVMIo",
81         first: 326,
82         last: 1256
83     ) {
84         edges {
85             cursor
86             node {
87                 createdAt
88                 id
89                 replyId
90                 notes(
91                     after: "OKLaEyaCRXZzPbczOzzL",
92                     before: "aALrKAXqGqTKmbIaQBIW",
93                     first: 6876,
94                     last: 1571
95                 ) {
96                     edges {
97                         cursor
98                         node {
99                             body
100                             bodyHtml
101                             createdAt
102                             id
103                             resolvable
104                             resolvedAt
105                             system
106                             updatedAt
107                             author {
108                                 avatarUrl
109                                 name
110                                 username
111                                 webUrl
112                                 snippets(
113                                     ids: [
114                                         "gid://gitlab/PersonalSnippet/20",

```

```

115         "gid://gitlab/PersonalSnippet/23",
116         "e\u0000",
117         "e\u0000"
118     ],
119     visibility: public,
120     type: project,
121     after: "VkZPAFgXGsLOdfHFUDRv",
122     before: "nWqYnCGxOSAGVQZHyc10",
123     first: 8620,
124     last: 9077
125 ) {
126     edges {
127         cursor
128         node {
129             content
130             createdAt
131             description
132             descriptionHtml
133             fileName
134             id
135             rawUrl
136             title
137             updatedAt
138             webUrl
139             visibilityLevel
140         }
141     }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }

```


Literaturverzeichnis

- [1] *Digitale Transformation*, <https://www.netzwerk-stiftungen-bildung.de/wissenscenter/glossar/digitale-transformation>, zuletzt besucht: 03.08.2023.
- [2] *Weltweiter IP-Traffic verdreifacht sich durch IoT und Video-Nutzung bis 2021*, <https://www.zdnet.de/88300485/weltweiter-ip-traffic-verdreifacht-sich-durch-iot-und-video-nutzung-bis-2021/>, zuletzt besucht: 03.0.2023.
- [3] *GraphQL vs REST APIs*, <https://hygraph.com/blog/graphql-vs-rest-apis>, zuletzt besucht: 03.08.2023.
- [4] *Was ist der Unterschied zwischen GraphQL und REST?* <https://aws.amazon.com/de/compare/the-difference-between-graphql-and-rest/>, zuletzt besucht: 18.06.2023.
- [5] P. A. J. Offutt, *Introduction to Software Testing*. 2008, ISBN: 978-0-521-88038-1.
- [6] D. S. Stefan Karlsson Adnan Causevic, "Automatic Property-based Testing of GraphQL APIs," *International Conference on Automation of Software Test*, 2021.
- [7] *Evo Master*, <https://github.com/EMResearch/EvoMaster>, zuletzt besucht: 15.06.2023.
- [8] S. Karlsson, A. Causevic und D. Sundmark, *QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs*, 2019. arXiv: 1912.09686 [cs.SE].
- [9] *Rest Test Gen*, <https://github.com/SeUniVr/RestTestGen/>, zuletzt besucht: 15.06.2023.
- [10] *GraphQL-API*, <https://docs.gitlab.com/ee/api/graphql/>, zuletzt besucht: 03.08.2023.
- [11] R. Diestel, *Graphentheorie*. 2000, ISBN: 3-540-67656-2.
- [12] *The Graph in GraphQL*, <https://dev.to/bogdanned/the-graph-in-graphql-1199>, zuletzt besucht: 04.08.2023.
- [13] H. Knebl, *Algorithmen und Datenstrukturen: Grundlagen und probabilistische Methoden für den Entwurf und die Analyse*. 2019, ISBN: 978-3-658-26512-0.
- [14] *Was ist eine API?* <https://www.redhat.com/de/topics/api/what-are-application-programming-interfaces>, zuletzt besucht: 03.08.2023.
- [15] *GraphQL-Specification*, <https://spec.graphql.org/June2018/>, zuletzt besucht: 16.06.2023.
- [16] *GraphQL is the better REST*, <https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>, zuletzt besucht: 15.06.2023.

- [17] *State of GraphQL 2022 survey*, <https://blog.graphqleditor.com/state-of-graphql-2022>, zuletzt besucht: 15.06.2023.
- [18] *Code using GraphQL*, <https://graphql.org/code/>, zuletzt besucht: 05.08.2023.
- [19] *Apollo Server*, <https://www.apollographql.com/docs/apollo-server/>, zuletzt besucht: 05.08.2023.
- [20] *Express GraphQL*, <https://github.com/graphql/express-graphql>, zuletzt besucht: 05.08.2023.
- [21] *HyGraph*, <https://hygraph.com/>, zuletzt besucht: 05.08.2023.
- [22] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*. 2014, ISBN: 978-1-4665-6069-7.
- [23] *Teststufen: WhiteBox und BlackBox-Testing*, <https://hmc2.net/page12/page9/>, zuletzt besucht: 05.08.2023.
- [24] *Gray Box Testing*, <https://www.geeksforgeeks.org/gray-box-testing-software-testing/>, zuletzt besucht: 28.07.2023.
- [25] *Serene - Clojure.Spec from GraphQL Schema*, <https://github.com/paren-com/serene>, zuletzt besucht: 15.06.2023.
- [26] *Clojure Spec - Data structure definiton*, <https://clojure.org/guides/spec>, zuletzt besucht: 15.06.2023.
- [27] *Mali - Data Driven Specification Library for Clojure*, <https://github.com/metosin/malli>, zuletzt besucht: 15.06.2023.
- [28] A. Belhadi, M. Zhang und A. Arcuri, *White-Box and Black-Box Fuzzing for GraphQL APIs*, 2022. arXiv: 2209.05833 [cs.SE].
- [29] *NetworkX - Network Analysis in Python*, <https://networkx.org>, zuletzt besucht: 06.07.2023.
- [30] *NetworkX - Network Analysis in Python*, <https://github.com/networkx/networkx>, zuletzt besucht: 06.07.2023.
- [31] *Faker - Faker is a Python package that generates fake data for you.* <https://github.com/joke2k/faker>, zuletzt besucht: 06.07.2023.
- [32] *Experimente Directory*, <https://github.com/gernhard1337/GraphQL-Testautomatisierung/tree/main/experiment/toy-experiment>, zuletzt besucht: 20.7.2023.
- [33] *Issue 1*, <https://gitlab.com/gitlab-org/gitlab/-/issues/208672>, zuletzt besucht: 30.07.2023.
- [34] *Issue 2*, <https://gitlab.com/gitlab-org/gitlab/-/issues/208125>, zuletzt besucht: 30.07.2023.
- [35] *Issue 3*, <https://gitlab.com/gitlab-org/gitlab/-/issues/208122>, zuletzt besucht: 30.07.2023.
- [36] *Issue 4*, <https://gitlab.com/gitlab-org/gitlab/-/issues/208121>, zuletzt besucht: 30.07.2023.

- [37] *Gitlab API Population Script*, https://github.com/gernhard1337/graphql-primepath-tester/blob/master/scripts/gitlab_population_skript.py, zuletzt besucht: 30.07.2023.
- [38] *Gitlab Paths from PrimePath Generation*, <https://github.com/gernhard1337/graphql-primepath-tester/blob/master/paths.txt>, zuletzt besucht: 30.07.2023.
- [39] *Explaining GraphQL Connections*, <https://www.apollographql.com/blog/graphql/explaining-graphql-connections/>, zuletzt besucht: 18.06.2023.
- [40] *The Query and Mutation types*, <https://graphql.org/learn/schema/the-query-and-mutation-types>, zuletzt besucht: 04.08.2023.
- [41] *Faker*, <https://faker.readthedocs.io/en/master/>, zuletzt besucht: 06.07.2023.
- [42] *Pytest: Helps you write better programs*, <https://docs.pytest.org/en/7.4.x/>, zuletzt besucht: 06.07.2023.
- [43] *Max Query complexity*, <https://docs.gitlab.com/ee/api/graphql/#max-query-complexity>, zuletzt besucht: 28.07.2023.
- [44] H. Noltemeier, *Graphentheoretische Konzepte und Algorithmen*. 2012, ISBN: 978-3-8348-1849-2.
- [45] *HTTP*, <https://developer.mozilla.org/en-US/docs/Web/HTTP>, zuletzt besucht: 09.07.2023.

Onlinere Ressourcen wurden am 6. August 2023 auf ihre Verfügbarkeit hin überprüft.