Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language

Olaf Hartig
Linköping University
olaf.hartig@liu.se

Jan Hidders

hidders@gmail.com

ABSTRACT

GraphQL is a highly popular new approach to build Web APIs. An important component of this approach is the GraphQL schema definition language (SDL). The original purpose of this language is to define a so-called GraphQL schema that specifies the types of objects that can be queried when accessing a specific GraphQL Web API. This paper focuses on the question: Can we repurpose this language to define schemas for graph databases that are based on the Property Graph model? This question is relevant because there does not exist a commonly adopted approach to define schemas for Property Graphs, and because the form in which GraphQL APIs represent their underlying data sources is very similar to the Property Graph model. To answer the question we propose an approach to adopt the GraphQL SDL for Property Graph schemas. We define this approach formally and show its fundamental properties.

CCS CONCEPTS

• **Information systems → Graph-based database models**; *Integrity checking*;

KEYWORDS

schema, constraints, graph database

ACM Reference Format:
Olaf Hartig and Jan Hidders. 2019. Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language. In *2nd Joint Int. Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19), June 30 2019, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3327964.3328495

1 INTRODUCTION

While graph database systems are becoming increasingly popular [2], their range of use cases broadens and new application requirements emerge. One of these requirements is the option to specify rigid forms of logical schemas that define exactly how a valid instance of a graph database has to look like and what constraints it has to satisfy. In the context of RDF-based graph databases, this development has led to the definition of the SHACL standard [14] and the Shape Expressions Language [16]. In contrast, in the context of the other prevalent form of graph databases, namely Property Graphs, such commonly agreed-upon approaches to define schemas do not yet exist. Motivated by the question of how this gap may be filled, we have looked at GraphQL [9], which is a new approach to building Web-based data access APIs that represent an underlying data source in a Property Graph-like form [12]. What makes GraphQL interesting in this context is that every GraphQL API is based on some kind of schema and for defining these schemas, the GraphQL approach introduces a developer-friendly language. Consequently, we aim to answer the following research question:

Can the GraphQL Schema Definition Language (SDL) be repurposed to also define schemas for Property Graphs?

To address this question we make the following contributions.

(1) We propose an approach to adopt the GraphQL SDL for Property Graph schemas. Section 3 describes this approach informally, and Section 5 provides a formal definition.
(2) We analyze the approach formally and show the following fundamental properties (cf. Section 6). The validation problem for the approach is in AC_0, which means that the computational complexity of verifying whether a Property Graph conforms to a schema is low. However, the approach also has a downside: it allows for schemas that are unsatisfiable (i.e., there can be no Property Graph that conforms to such an unsatisfiable schema). We show that the problem of checking the satisfiability of schemas is NP-hard and, as an upper bound, we show that the problem is in PSPACE.
(3) As a formal basis of the aforementioned contributions, and also for further work on GraphQL schemas and their underlying type system, we provide a concise formalization of the notion of schemas captured by the GraphQL SDL (Section 4).

To the best of our knowledge, this paper is the first to provide a *comprehensive* formal foundation for Property Graph schemas. Before we present our contributions, we discuss relevant existing work that is related to our work in this paper, which includes providing an overview of the main features of GraphQL schemas.

2 RELATED WORK

2.1 Schemas for Property Graphs

We first recall that, informally, a Property Graph is a directed multigraph in which every node and every edge may be assigned a label as well as a set of so-called properties, where each property consists of a name and a value [17]. There are several proposals to define the notion of a Property Graph formally (e.g., [11], [4], [3], [7]). For our work in this paper we adopt the following definition of Angles et al. [4], which asssumes three infinite countable sets: *Labels* (labels), *Props* (property names), and *Values* (property values).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GRADES-NDA'19, June 30-July 5 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-6789-9/19/06…$15.00
https://doi.org/10.1145/3327964.3328495

Definition 2.1 (Property Graph [4]). A Property Graph is a tuple $(V, E, \rho, \lambda, \sigma)$ with the following five components:

- V is a finite set of *vertices* (or *nodes*);
- E is a finite set of *edges* such that $V \cap E = \emptyset$;
- $\rho \colon E \to (V \times V)$ is a total function;
- $\lambda \colon (V \cup E) \to Labels$ is a total function;
- $\sigma \colon (V \cup E) \times Props \nrightarrow Values$ is a partial function.

When it comes to (logical) schemas for Property Graphs, there is no approach that is commonly agreed upon. However, many database management systems for Property Graphs support some proprietary form of schemas. In some systems, users can specify the schema using a data definition language such as Neo4j's Cypher [15, Chapter 5] and TigerGraph's GSQL [19]; in other systems, the schema is specified using the system-specific API (e.g., Sparksee [18, Section 3.4], JanusGraph [13, Chapter 5]). Typical features are notions of node types and edge types, as well as property types that are associated with a node type or with an edge type; additionally, the systems support different kinds of constraints. A detailed survey of the exact features supported by each system is outside the scope of this paper. In fact, a thorough discussion and comparison of these features may hardly be possible because there does not exist any publication that provides a formal definition of any of the system-specific notions of schema and schema validation. One of our contributions in this paper is to introduce such formal definitions for the GraphQL schema definition language.

To the best of our knowledge, the only work on Property Graph schemas in the research literature is by Angles [3]. The author provides a formal definition of some form of a Property Graph schema. This definition introduces a notion of node types and a notion of edge types, as well as constraints that allow users to specify (i) what properties can be used for each node type and for each edge type, and (ii) what types of edges can be used between any given pair of node types. Additionally, Angles outlines other possible kinds of constraints and, for some of them, describes how they may be incorporated into the given definition. In particular, these constraints consider mandatory properties, mandatory edges, uniqueness of properties, and cardinality constraints. All these features are also covered by the approach proposed in this paper. Like Angles, we define our approach formally. However, in contrast to Angles, we additionally show fundamental properties of our approach. Another difference to Angles' work is that our approach is grounded in a concrete language that developers can use.

2.2 GraphQL Schemas

Despite the name, GraphQL schemas are not a form of schemas in the traditional sense. That is, such a schema does not specify what the instances of a particular database may look like and what constraints have to be enforced for these database instances. Instead, a GraphQL schema represents more a form of vocabulary supported by a GraphQL Web API. More specifically, the schema of such an API specifies the types of objects for which the API has data and what kind of data it has for these objects. The GraphQL schema definition language (SDL) for defining GraphQL schemas has been officially introduced in the June 2018 Edition of the GraphQL specification [9].

In this paper we assume familiarity with the GraphQL SDL. For readers who are not familiar with this language we refer to the Appendix in which we illustrate the main features the language.

GraphQL schemas—and, in fact, GraphQL in general—have not attracted much attention in the research community so far. The only work related to our work in this paper is the work by Hartig and Pérez who study the semantics and the complexity of the GraphQL query language [12]. As one of the foundations of their work, the authors provide a formal definition of the notion of a GraphQL schema. We have used this definition as a starting point for our definitions in Section 4. However, we had to make several important extensions because Hartig and Pérez' definition covers only a limited subset of the features of the GraphQL SDL.

3 DESCRIPTION OF THE APPROACH

This section describes the proposed approach informally. That is, we outline our general idea of what it means for a Property Graph to conform to a schema defined in the GraphQL SDL and we introduce how the various features of the SDL are interpreted in this context.

3.1 Specifying Types of Nodes using Object Type Definitions

The main type of elements defined in a GraphQL schema are object types that consist of a name and a list of field definitions. If used for Property Graphs, we propose that object types define the types of nodes that a schema-conformant Property Graph may contain. That is, for every node in the graph, the label of the node must be the name of one of the object types in the schema. We call this object type the *type of the node* or we say that *the node is of this type*.

Example 3.1. Consider the following example schema. Property Graphs that conform to it can contain only two types of nodes: nodes with the label "UserSession" and nodes with the label "User".

```
1 type UserSession {
2  id: ID!           @required
3  user: User!       @required
4  startTime: Time!  @required
5  endTime: Time!
6 }
```

```
7  type User {
8    id: ID!          @required
9    login: String!   @required
10   nicknames: [String!]!
11 }
12 scalar Time
```

While we use the name of an object type as the label of each node that is of this type, the field definitions of the object type specify what properties and what outgoing edges these nodes may have. To this end, we distinguish two types of field definitions: (i) *attribute definitions* are field definitions in which the type of the possible field values is a scalar type (e.g., `Int`, `Float`, `String`), an enumeration type, or a list type that wraps a scalar type or an enumeration type, and (ii) *relationship definitions* are field definitions in which the type is an object type or a list type wrapping an object type.

Example 3.2. The object type `UserSession` (cf. Example 3.1) contains one relationship definition (the field `user`) and three attribute definitions (fields `id`, `startTime`, and `endTime`). The field definitions in the type `User` all are attribute definitions (notice that `String` and `ID` are built-in scalar types in the GraphQL DSL).

3.2 Specifying Node Properties using Field Definitions

Every field definition that is an attribute definition specifies that the corresponding nodes may have a property whose name is the name of the field and whose values must be of a type that depends on the type given in that field definition. Hence, if the type in the field definition is a scalar type, the property value must be of that type; if, on the other hand, the type in the field definition is an enumeration type, the property value must be one of the values of that enumeration type. If the type in the field definition is a list type that wraps a scalar type or an enumeration type, then the property value must be an array of values of the wrapped type.

For some properties, we may want to specify that they are required for all nodes of the corresponding type. To capture this kind of constraint we use the notion of directives that the GraphQL SDL introduces as a form of annotations that can be added to field definitions (and to other elements in a schema). That is, we introduce the directive `@required` for the aforementioned purpose.

Example 3.3. Given the schema of Example 3.1, every node with the label "User" may have two or three properties. That is, two properties are mandatory and they must have the name "id" and "login", respectively. The third property is optional and, if used, its name must be "nicknames". The value of the "id" property must be some form of an identifier, the value of "login" must be a (single) string, and the value of "nicknames" must be an array of strings. Nodes with the label "UserSession" must have the properties "id" and "startTime" and, additionally, may have the property "endTime". Notice that the field definition of the field called `user` does not define a property because it is not an attribute definition (cf. Example 3.2).

As another form of constraints that involves node properties, we define a notion of key constraints. To this end, we introduce the directive `@key` that can be added to the definition of an object type. Each such `@key` directive must contain an argument names `fields` that lists the names of all properties that belong to the key.

Example 3.4. Consider again the schema of Example 3.1. To specify that the (mandatory) "id" property of all "User" nodes is a key (i.e., all nodes of type "User" must have a unique value for this property) we have to modify line 7 of the schema as follows.

```
7 type User  @key(fields:["id"])  {
```

Alternatively, the "login" property may also be used as a key, which we may indicate by extending line 7 further:

```
7 type User  @key(fields:["id"])  @key(fields:["login"])  {
```

3.3 Specifying Outgoing Edges using Field Definitions

Every field definition that is a relationship definition specifies that the corresponding nodes may have outgoing edges whose label is the name of the field and whose target node is of the type that is the object type given in that field definition. If the type in the field definition is a list type (that wraps an object type), then a node may have multiple such outgoing edges; otherwise, the nodes must not have more than one such outgoing edge. Hence, the latter case (no list type) presents a form of cardinality constraints for the relationship type that is captured by the field definition. Additionally, it is also possible to specify a form of participation constraints: If field definition contains the `@required` directive, then it is mandatory for nodes to have such an outgoing edge; otherwise, it is optional.

Example 3.5. In a Property Graph that conforms to the schema of Example 3.1, every "UserSession" node must have exactly one outgoing edge. The label of this edge must be "user" and the edge must point to a node with the label "User".

Example 3.6. The following schema illustrates other possible combinations of the aforementioned constraints on outgoing edges.

```
1 type Author {
2  favoriteBook:Book
3  relatedAuthor:[Author]
4 }
```

```
5 type Book {
6  title:String!
7  author:[Author] @required
8 }
```

Based on this schema, every "Author" node may have at most one "favoriteBook" edge to a "Book" node, but it is not mandatory for "Author" nodes to have such an edge. Additionally, every "Author" node may have an arbitrary number of "relatedAuthor" edges to "Author" nodes (including none). Hence, there may also be "Author" nodes that do not have any outgoing edge. In contrast, every "Book" node must have at least one outgoing edge, but may also have more, all of which must be labeled "author" and point to an "Author" node.

In the cases in which a node may have multiple outgoing edges of the same type (i.e., with the same label), we may want to require that each of these edges must point to a different target node. To capture this type of constraints we introduce the directive `@distinct`. Additionally, we introduce the directive `@noloops` that can be used to specify that the target node of edges must not be the same as their respective source node; i.e., they must point to a target node that is different from the source node. Apparently, such a no-loops constraint makes sense only for edges for which source and target nodes may be of the same type.

Example 3.7. We may extend line 7 of the schema in Example 3.6 by adding the `@distinct` directive as follows.

```
7  author: [Author]  @required @distinct
```

As a consequence of this modification, for every "Book" node, each of its outgoing "author" edges must point to a different "Author" node. The same type of constraint is reasonable for the "relatedAuthor" edges (line 3). Additionally, we may want to specify that a "relatedAuthor" edge must not point back to the same "Author" node, for which we my use the `@noloops` directive as follows.

```
3  relatedAuthor: [Author]  @distinct @noloops
```

Notice that the directive `@distinct` is symmetric; that is, it represents a constraint not only for the source node of an edge but also for the target node. For instance, the `@distinct` constraint on the "author" edges (cf. Example 3.7) does not only mean that for every "Book" node, all of its outgoing "author" edges must point to different "Author" nodes, but also that for an "Author" node, its incoming "author" edges must all come from different "Book" nodes.

There are further constraints with a focus on the target nodes of edges: On the one hand, we may want to require that for some type

of edges, any possible target node can have at most one incoming edge of this type. We introduce the directive `@uniqueForTarget` to represent this constraint. On the other hand, we may also want to require that for some type of edges, any possible target node must have at least one incoming edge of this type. To capture this constraint we introduce the directive `@requiredForTarget`.

Example 3.8. Consider the following extension of the schema in Example 3.6. Given this extension, we have the following additional constraints for "Book" nodes: Every "Book" node may have at most one incoming "contains" edge (which are required outgoing edges for "BookSeries" nodes), but there may be "Book" nodes that do not have any such incoming edge. Additionally, every "Book" node must have exactly one incoming "published" edge.

```
9  type BookSeries {
10  contains:[Book]
11           @required
12           @uniqueForTarget
13 }
```

```
14 type Publisher {
15  published:[Book]
16          @uniqueForTarget
17          @requiredForTarget
18 }
```

We emphasize that the types of constraints that we have introduced can be used to capture any combination of cardinality restrictions that is possible for binary relationships. The following table illustrates all these combinations for an example relationship labeled "rel" from nodes of type "A" to nodes of type "B".

If "rel" is a	then the definition of the type `A` contains:
1:1 relationship,	`rel: B @uniqueForTarget`
1:*N* relationship,	`rel: B`
N:1 relationship,	`rel: [B] @uniqueForTarget`
N:*M* relationship,	`rel: [B]`

3.4 Specifying Edges with Multiple Types of Nodes based on Interfaces or Union Types

In addition to object types, the GraphQL SDL introduces the notion of interface types and union types. While we do not use these notions as types that can be explicitly assigned to nodes in Property Graphs, we propose to use these notions to capture cases in which some type of edges may have multiple types of target nodes.

Example 3.9. According to the following example schema, every "Person" node may have an outgoing edge labeled "favoriteFood" that points either to a "Pizza" node or to a "Pasta" node.

```
1  type Person {
2    name: String!
3    favoriteFood: Food
4  }
5
6  union Food = Pizza | Pasta
7
```

```
8  type Pizza {
9    name: String!
10   toppings: [String!]!
11 }
12 type Pasta {
13   name: String!
14 }
```

Example 3.10. Consider the following example schema. It captures exactly the same restrictions on Property Graphs as are captured by the schema in the previous example.

```
1  type Person {
2    name: String!
3    favoriteFood: Food
4  }
5  interface Food {
6    name: String!
7  }
```

```
8  type Pizza implements Food {
9    name: String!
10   toppings: [String!]!
11 }
12 type Pasta implements Food {
13   name: String!
14 }
```

As the two examples illustrate, in the context of SDL-based schema definitions for Property Graphs, using interface types or union types are two different options that serve the exact same purpose. Our proposal allows for both options to give users more flexibility if they aim to use their schema also as a basis for developing a GraphQL API on top of their Property Graph.

While using union types or interface types as described above focuses on the possible target nodes of edges, our proposed approach also covers cases in which some type of edges may have multiple types of source nodes. To this end, the corresponding relationship definition simply needs to be repeated in every object type definition of all types of nodes that may have such outgoing edges.

Example 3.11. The following extension of the previous example schema allows "Person" nodes to have incoming "owner" edges from "Car" nodes as well as from "Motorcycle" nodes.

```
15 type Car {
16   brand: String!
17   owner: Person
18 }
```

```
19 type Motorcycle {
20   brand: String!
21   owner: Person
22 }
```

3.5 Specifying Edge Properties using Field Argument Definitions

An important feature of Property Graphs that we have not covered so far are edge properties. To specify what properties an edge may have we use the definition of field arguments that can be provided for every field definition. That is, every field argument defined in a relationship field definition specifies that the corresponding edge may have a property whose name is the name of the field argument and whose value must be of the type mentioned in the definition of the field argument. Hence, field argument definitions that can be used in this way must have a scalar type, an enumeration type, or a list type that wraps a scalar or an enumeration type.

Example 3.12. We may modify line 3 in our initial example schema about user sessions (cf. Example 3.1) as follows.

```
3   user(certainty:Float! comment:String): User!  @required
```

Now, every "user" edge must have a “certainty” property with a floating point number as value. Additionally, such an edge may have an optional "comment" property with a string value.

As the previous example demonstrates, if the type in the field argument definition is marked as non-nullable, then the specified edge property is mandatory. If the type is a list type (wrapping a scalar type or and enumeration type), then the value of the specified edge property must be an array of values of the wrapped type.

3.6 Additional Remarks

While our proposal leverages and repurposes most of the features of the GraphQL SDL, some features cannot be meaningfully adapted

when it comes to defining schemas for Property Graphs. Consequently, we have ignored these features for our proposed approach. If a schema definition uses such a feature that is not covered by our approach, this part of the schema definition is simply ignored when checking whether a Property Graph satisfies the schema definition.

For instance, when adopting the definition of field arguments as a means to specify what properties an edge may have, we deliberately consider only the field argument definitions whose type is either a scalar type, an enumeration type, or a list type that wraps one of the former. Hence, we ignore field argument definitions in which the type of possible values is defined to be a complex input type. Such field argument definitions are not suitable to specify potential edge properties because the value of any edge property can only be a simple atomic value or a list of such values [7].

A related example are field arguments in attribute definitions. Recall that attribute definitions are the field definitions that, according to our proposal, can be used to specify what properties particular nodes may have. Since in the Property Graph model the name (or the value) of a node property cannot have additional arguments associated with it, field arguments in attribute definitions cannot be meaningfully used for our proposal. Hence, an attribute definition in an SDL-based schema definition for Property Graphs should not contain field arguments (and if it does, we ignore these arguments).

A last example of SDL features that are not meaningful for Property Graph schemas are the root operation types called `Query`, `Mutation`, and `Subscription`. These types specify the objects that have to be used as entry points in requests to a GraphQL API. Since the purpose of our proposal is to use the SDL to define schemas for Property Graphs (and not to define schemas for GraphQL APIs over Property Graphs), root types are not needed in our context. Notice, however, that by omitting root operation types, the SDL-based Property Graph schemas created based on our proposal are not complete GraphQL schemas (as used for GraphQL APIs) because at least the `query` type is mandatory in such GraphQL API schemas.

Nonetheless, even if it is not the primary purpose of the schemas defined based on our proposed approach, it seems like a natural next step to also use them as a basis for developing GraphQL APIs to access Property Graphs. To this end, such schemas may be extended into actual GraphQL schemas as required for creating GraphQL APIs. From a technical perspective, the only thing that needs to be added in this case is the `query` type, and perhaps also the `mutation` type for providing write access. However, for practical purposes, further elements will have to be added when extending an SDL-based Property Graph schema into a GraphQL API schema.

In particular, it may be useful for GraphQL APIs over Property Graphs to support queries with which the directed edges in the graph can also be traversed in their opposite directions. Such a bidirectional traversal is not possible with a schema defined based on our approach. The reason for this limitation is that our SDL-based Property Graph schemas specify potential edges in the object types for the nodes for which the edges are outgoing. Hence, the object types in the schema that specify the potential target nodes do not contain any mention of the incoming edges. We emphasize that specifying every type of edges only once is sufficient for the purpose of defining a Property Graph schema, but it is not sufficient for supporting bidirectional traversal in GraphQL queries. We also emphasize that in query languages that are explicitly designed for Property Graphs (such as Gremlin and Cypher) it is a native feature that edges can be traversed both ways. In contrast, to enable bidirectional traversal in GraphQL queries, the schema of the GraphQL API has to explicitly mention potential edges twice: once from the perspective of the source nodes and once For this purpose, an extended GraphQL schema has to explicitly mention potential edges also from the perspective of the target nodes. Although we believe that it is not difficult to address this limitation when extending an SDL-based Property Graph schema into a GraphQL API schema, we are planning to complement our proposed approach with guidelines on how to develop such an extension in a principled manner.

4 FORMALIZATION OF GRAPHQL SCHEMAS

To define our approach formally we first need to formalize the notion of GraphQL schemas. To this end, we adopt Hartig and Pérez' formalization approach [12] and extend it by also capturing non-null types, the semantics of wrapping types, and directives (which have been ignored by the original authors). This section presents this extended definition. As done by the original authors, for each concept that the definition captures, we refer to corresponding section of the GraphQL specification that introduces the concept.

4.1 GraphQL Type System

We consider the following pairwise disjoint, countably infinite sets: $\mathtt{Types}$ (type names, §3.4 [9]), $\mathtt{Fields}$ (field names, §3.6 [9]), $\mathtt{Arguments}$ (argument names, §3.6.1 [9]), and $\mathtt{Directives}$ (directive names, §3.13 [9]). Moreover, there exists a set $\mathtt{Scalars}$ (scalar type names, §3.5 [9])[1] that is a subset of $\mathtt{Types}$, and there are five built-in scalar types: `Int` (§3.5.1 [9]), `Float` (§3.5.2 [9]), `String` (§3.5.3 [9]), `Boolean` (§3.5.4 [9]), and `ID` (§3.5.5 [9]). We also consider a set $\mathtt{Vals}$ (scalar values) and a function $values : \mathtt{Scalars} \rightarrow 2^{\mathtt{Vals}}$ that assigns a set of values to every scalar type. We assume that $\mathtt{Types} \cup \mathtt{Fields} \cup \mathtt{Arguments} \cup \mathtt{Directives} \subset values(\mathtt{String})$.

In addition to the named types in $\mathtt{Types}$, the GraphQL SDL introduces two kinds of so-called *wrapping types* that are created based on the types in $\mathtt{Types}$ (§3.4.1 [9]). One kind of wrapping types are *non-null types* (§3.12 [9]); given a type $\mathtt{t}$, we write $\mathtt{t}^!$ to denote the non-null type that wraps $\mathtt{t}$. The other kind of wrapping types are *list types* (§3.11 [9]); we write $[\mathtt{t}]$ to denote the list type constructed from a type $\mathtt{t}$, where $\mathtt{t}$ may be in $\mathtt{Types}$ or it may be a non-null type $\mathtt{nt}^!$ that wraps a named type $\mathtt{nt} \in \mathtt{Types}$. A list type $[\mathtt{t}]$ may also be wrapped as a non-null type $[\mathtt{t}]^!$ (§3.12.1 [9]). Hence, by combining these definitions, the following four types can be formed by wrapping a named type: $\mathtt{t}^!$, $[\mathtt{t}]$, $[\mathtt{t}^!]$, and $[\mathtt{t}^!]^!$. Hereafter, for any subset $X \subseteq \mathtt{Types}$ we let $\mathtt{W}_X$ denote the set of all the types that can formed by wrapping the types in X.

To refer to the underlying named types of wrapping types we introduce the function *basetype* which we define recursively as follows. If $\mathtt{t} \in \mathtt{Types}$, then $basetype(\mathtt{t}) = \mathtt{t}$; if $\mathtt{t}$ is $\mathtt{tt}^!$ or $[\mathtt{tt}]$ such that $\mathtt{tt} \in \mathtt{Types}$, then $basetype(\mathtt{t}) = \mathtt{tt}$; if $\mathtt{t}$ is $[\mathtt{tt}]$ such that $\mathtt{tt}$ is a non-null type, then $basetype(\mathtt{t}) = basetype(\mathtt{tt})$; finally, if $\mathtt{t}$ is $\mathtt{tt}^!$ such that $\mathtt{tt}$ is a list type, then $basetype(\mathtt{t}) = basetype(\mathtt{tt})$.

The semantics of wrapped scalar types is defined by generalizing the function *values* to $values_{\mathtt{W}}$ for types in $\mathtt{Scalars} \cup \mathtt{W}_{\mathtt{Scalars}}$. To

[1] For the sake of simplicity, we assume that `Scalars` includes the *enum types* that are treated separately in the GraphQL specification (cf. §3.9 [9]).

this end, we assume the existence of a special value null that is not in $\mathtt{Vals}$. Then, for all types $\mathtt{t} \in \mathtt{Scalars} \cup \mathtt{W}_{\mathtt{Scalars}}$, the function $values_{\mathtt{W}}$ is defined recursively as follows:

(1) if $\mathtt{t} \in \mathtt{Scalars}$, then $values_{\mathtt{W}}(\mathtt{t}) = values(\mathtt{t}) \cup \{\text{null}\}$;
(2) if $\mathtt{t}$ is $\mathtt{tt}^!$, then $values_{\mathtt{W}}(\mathtt{t}) = values_{\mathtt{W}}(\mathtt{tt}) \setminus \{\text{null}\}$;
(3) if $\mathtt{t}$ is $[\mathtt{tt}]$, then $values_{\mathtt{W}}(\mathtt{t}) = \mathcal{L}(values_{\mathtt{W}}(\mathtt{tt})) \cup \{\text{null}\}$ where $\mathcal{L}(X)$ is the set of all finite lists with elements from the set X.

Notice that, by recursion, this definition also captures the cases $values_{\mathtt{W}}([\mathtt{t}]^!)$, $values_{\mathtt{W}}([\mathtt{t}^!])$, and $values_{\mathtt{W}}([\mathtt{t}^!]^!)$.

4.2 GraphQL Schema

GraphQL schemas are defined over finite subsets of the five aforementioned sets. Hence, we assume five finite sets $\mathtt{F} \subset \mathtt{Fields}$, $\mathtt{A} \subset \mathtt{Arguments}$, $\mathtt{T} \subset \mathtt{Types}$, $\mathtt{S} \subset \mathtt{Scalars}$, and $\mathtt{D} \subset \mathtt{Directives}$, where $\mathtt{T}$ is the disjoint union of $\mathtt{O}_\mathtt{T}$ (object types, §3.6 [9]), $\mathtt{I}_\mathtt{T}$ (interface types, §3.7 [9]), $\mathtt{U}_\mathtt{T}$ (union types, §3.8 [9]) and $\mathtt{S}$. We now have everything necessary to define the notion of a GraphQL schema.

Definition 4.1 (GraphQL schema). A GraphQL schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S}, \mathtt{D})$ is composed of the following assignments:

- $type_\mathcal{S} = type^\mathsf{F}_\mathcal{S} \cup type^\mathsf{AF}_\mathcal{S} \cup type^\mathsf{AD}_\mathcal{S}$ where
  - $type^\mathsf{F}_\mathcal{S} : (\mathtt{O}_\mathtt{T} \cup \mathtt{I}_\mathtt{T}) \times \mathtt{F} \nrightarrow \mathtt{T} \cup \mathtt{W}_\mathtt{T}$ assigns a type to every field that is defined for an object type or an interface type,
  - $type^\mathsf{AF}_\mathcal{S} : \mathrm{dom}(type^\mathsf{F}_\mathcal{S}) \times \mathtt{A} \nrightarrow \mathtt{S} \cup \mathtt{W}_\mathtt{S}$ assigns a type to every argument of fields that are defined for a type, and
  - $type^\mathsf{AD}_\mathcal{S} : \mathtt{D} \times \mathtt{A} \nrightarrow \mathtt{S} \cup \mathtt{W}_\mathtt{S}$ assigns a type to every argument that is defined for a type of directives;
- $union_\mathcal{S} : \mathtt{U}_\mathtt{T} \to 2^{\mathtt{O}_\mathtt{T}}$ assigns a nonempty set of object types to every union type;
- $implementation_\mathcal{S} : \mathtt{I}_\mathtt{T} \to 2^{\mathtt{O}_\mathtt{T}}$ assigns a set of object types to every interface type;
- $directives_\mathcal{S} = directives^\mathsf{T}_\mathcal{S} \cup directives^\mathsf{F}_\mathcal{S} \cup directives^\mathsf{AF}_\mathcal{S}$ where
  - $directives^\mathsf{T}_\mathcal{S} : \mathtt{T} \to 2^{\mathtt{D} \times AV}$ assigns a set of pairs $(\mathtt{d}, argvals) \in \mathtt{D} \times AV$ to every type, where the set AV consists of all possible partial functions $argvals : \mathtt{A} \nrightarrow \bigcup_{\mathtt{st} \in \mathtt{S} \cup \mathtt{W}_\mathtt{S}} values_\mathtt{W}(\mathtt{st})$,
  - $directives^\mathsf{F}_\mathcal{S} : \mathrm{dom}(type^\mathsf{F}_\mathcal{S}) \to 2^{\mathtt{D} \times AV}$ assigns a set of pairs $(\mathtt{d}, argvals) \in \mathtt{D} \times AV$ to every field in a type, and
  - $directives^\mathsf{AF}_\mathcal{S} : \mathrm{dom}(type^\mathsf{AF}_\mathcal{S}) \to 2^{\mathtt{D} \times AV}$ assigns a set of pairs $(\mathtt{d}, argvals) \in \mathtt{D} \times AV$ to every field argument in a type.

Example 4.2. The schema in Example 3.9 is captured formally by the following schema $\mathcal{S}$ over the following sets $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S}, \mathtt{D})$:

- $\mathtt{F} = \{\mathtt{name}, \mathtt{favoriteFood}, \mathtt{toppings}\}$,
  $\mathtt{A} = \emptyset$,
  $\mathtt{T} = \mathtt{O}_\mathtt{T} \cup \mathtt{I}_\mathtt{T} \cup \mathtt{U}_\mathtt{T} \cup \mathtt{S}$ where
  - $\mathtt{O}_\mathtt{T} = \{\mathtt{Person}, \mathtt{Pizza}, \mathtt{Pasta}\}$,
  - $\mathtt{I}_\mathtt{T} = \emptyset$,
  - $\mathtt{U}_\mathtt{T} = \{\mathtt{Food}\}$,
  $\mathtt{S} = \{\mathtt{String}\}$,
  $\mathtt{D} = \emptyset$;
- $type^\mathsf{F}_\mathcal{S} = \{ (\mathtt{Person}, \mathtt{name}) \mapsto \mathtt{String}^!,$
  $(\mathtt{Person}, \mathtt{favoriteFood}) \mapsto \mathtt{Food},$
  $(\mathtt{Pizza}, \mathtt{name}) \mapsto \mathtt{String}^!,$
  $(\mathtt{Pizza}, \mathtt{topppings}) \mapsto [\mathtt{String}^!]^!,$
  $(\mathtt{Pasta}, \mathtt{name}) \mapsto \mathtt{String}^! \};$
- $type^\mathsf{AF}_\mathcal{S} = \emptyset$; $type^\mathsf{AD}_\mathcal{S} = \emptyset$;
- $union_\mathcal{S} = \{ \mathtt{Food} \mapsto \{\mathtt{Pizza}, \mathtt{Pasta}\} \}$;
- $implementation_\mathcal{S} = \emptyset$;
- $directives_\mathcal{S} = \emptyset$.

By Definition 4.1, it is captured implicitly which fields a certain type may have (by letting $type_\mathcal{S}$ be defined for the relevant combination of type and field). To make this explicit for both fields and arguments, we introduce three helper functions. Given a GraphQL schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S}, \mathtt{D})$, for every $\mathtt{t} \in \mathtt{O}_\mathtt{T} \cup \mathtt{I}_\mathtt{T}$ we define

$$fields_\mathcal{S}(\mathtt{t}) = \{\mathtt{f} \in \mathtt{F} \mid (\mathtt{t}, \mathtt{f}) \in \mathrm{dom}(type^\mathsf{F}_\mathcal{S})\},$$

and for every $\mathtt{f} \in fields_\mathcal{S}(\mathtt{t})$ we define

$$args_\mathcal{S}(\mathtt{t}, \mathtt{f}) = \{\mathtt{a} \in \mathtt{A} \mid ((\mathtt{t}, \mathtt{f}), \mathtt{a}) \in \mathrm{dom}(type^\mathsf{AF}_\mathcal{S})\}.$$

Additionally, we overload $args_\mathcal{S}$, and define for every $\mathtt{d} \in \mathtt{D}$,

$$args_\mathcal{S}(\mathtt{d}) = \{\mathtt{a} \in \mathtt{A} \mid (\mathtt{d}, \mathtt{a}) \in \mathrm{dom}(type^\mathsf{AD}_\mathcal{S})\}.$$

To avoid an overly complex formalization, our definition of a GraphQL schema does not capture the additional notion of *input types* (cf. §3.10 [9]). Moreover, since we are interested in using these schemas for Property Graphs and not for GraphQL APIs, we do not explicitly consider *root operation types* (e.g., query; §3.3 [9]). However, we capture the concept of interfaces and their implementations (cf. §3.7 [9]) with the following notion of consistency.

4.3 Consistency of GraphQL Schemas

Informally, a GraphQL schema is *consistent* if (i) every directive in the schema uses exactly the arguments as defined for its type and (ii) every object type that implements an interface type contains at least all the fields that the interface type contains. To define schema consistency formally, we need to introduce the notion of a *subtype relation* $\sqsubseteq_\mathcal{S}$ given a schema $\mathcal{S}$, which is defined as the smallest relation over $\mathtt{T} \cup \mathtt{W}_\mathtt{T}$ that satisfies the following rules:

$$(1)\ \mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{t} \qquad (2)\ \frac{\mathtt{t} \in implementation_\mathcal{S}(\mathtt{s})}{\mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{s}} \qquad (3)\ \frac{\mathtt{t} \in union_\mathcal{S}(\mathtt{s})}{\mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{s}}$$

$$(4)\ \frac{\mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{s}}{[\mathtt{t}] \sqsubseteq_\mathcal{S} [\mathtt{s}]} \qquad (5)\ \frac{\mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{s}}{\mathtt{t} \sqsubseteq_\mathcal{S} [\mathtt{s}]} \qquad (6)\ \frac{\mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{s}}{\mathtt{t}^! \sqsubseteq_\mathcal{S} \mathtt{s}} \qquad (7)\ \frac{\mathtt{t} \sqsubseteq_\mathcal{S} \mathtt{s}}{\mathtt{t}^! \sqsubseteq_\mathcal{S} \mathtt{s}^!}$$

Now we are ready to define schema consistency:

Definition 4.3 (Interface Consistency). A GraphQL schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S}, \mathtt{D})$ is *interface consistent* if for each interface type $\mathtt{it} \in \mathtt{I}_\mathtt{T}$, every (implementing) object type $\mathtt{ot} \in implementation_\mathcal{S}(\mathtt{it})$, and every field $\mathtt{f} \in fields_\mathcal{S}(\mathtt{it})$, it holds that

(1) $\mathtt{f} \in fields_\mathcal{S}(\mathtt{ot})$ and $type_\mathcal{S}(\mathtt{f}, \mathtt{ot}) \sqsubseteq_\mathcal{S} type_\mathcal{S}(\mathtt{f}, \mathtt{it})$,
(2) for every $\mathtt{a} \in args_\mathcal{S}(\mathtt{f}, \mathtt{it})$, we have that $\mathtt{a} \in args_\mathcal{S}(\mathtt{f}, \mathtt{ot})$ and $type^\mathsf{AF}_\mathcal{S}(\mathtt{a}, (\mathtt{f}, \mathtt{it})) = type^\mathsf{AF}_\mathcal{S}(\mathtt{a}, (\mathtt{f}, \mathtt{ot}))$, and
(3) for every $\mathtt{a} \in args_\mathcal{S}(\mathtt{f}, \mathtt{ot})$ for which $\mathtt{a} \notin args_\mathcal{S}(\mathtt{f}, \mathtt{it})$, it holds that $type^\mathsf{AF}_\mathcal{S}(\mathtt{a}, (\mathtt{f}, \mathtt{ot}))$ is not a non-null type, i.e., it is not of the form $\mathtt{t}^!$.

Definition 4.4 (Directives Consistency). A GraphQL schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S}, \mathtt{D})$ is called *directives consistent* if for every pair $(\mathtt{d}, argvals) \in \mathtt{D} \times AV$ that is in at least one of the sets assigned by $directives_\mathcal{S}$ it holds that

(1) for every $(\mathsf{d}, \mathsf{a}) \in \mathrm{dom}(type^{\mathsf{AD}}_{\mathcal{S}})$ for which $type^{\mathsf{AD}}_{\mathcal{S}}(\mathsf{d}, \mathsf{a})$ is a non-null type, we have that $\mathsf{a} \in \mathrm{dom}(argvals)$, and
(2) $argvals(\mathsf{a}) \in values_{\mathsf{W}}(type^{\mathsf{AD}}_{\mathcal{S}}(\mathsf{d}, \mathsf{a}))$ for all $\mathsf{a} \in \mathrm{dom}(argvals)$.

Definition 4.5 (Consistency). A schema $\mathcal{S}$ over $(\mathsf{F}, \mathsf{A}, \mathsf{T}, \mathsf{S}, \mathsf{D})$ is *consistent* if $\mathcal{S}$ is interface consistent and directives consistent.

We assume that all GraphQL schemas in this paper are consistent. Moreover, unless stated otherwise, we assume that D contains the directives `@distinct`, `@noLoops`, `@required`, `@requiredForTarget`, `@uniqueForTarget` and `@key`, and that their types are defined by $type^{\mathsf{AD}}_{\mathcal{S}}$ such that they have no arguments, except for `@key` for which we have that $type^{\mathsf{AD}}_{\mathcal{S}}(\texttt{@key}, \mathsf{fields}) = [\texttt{String}^!]^!$.

5 DEFINITION OF THE APPROACH

In this section we will define what it means for a property graph to satisfy a schema. This is split into three stages. First, we introduce *weak satisfaction* that captures that elements of the property graph that are assigned to certain types in the schema, satisfy the requirements of those types. Second, we define *directives satisfaction* that captures that all directives as satisfied. Finally, we present *strong satisfaction* that combines the previous two and adds that all elements in the property graph must be assigned to at least one declaration in the schema.

For our approach we assume that GraphQL type names and field names can be used as labels in Property Graphs; that is, we assume that $\texttt{Types} \subseteq Labels$ and $\texttt{Fields} \subseteq Labels$. Similarly, field names and argument names of the GraphQL SDL can be used as property names in Property Graphs, and GraphQL scalar values can be used as property values; i.e., $\texttt{Fields} \subseteq Props$, $\texttt{Arguments} \subseteq Props$, and $\texttt{Vals} \subseteq Values$ (note that the sets *Labels*, *Props*, and *Values* have been introduced in Section 2.1).

Definition 5.1 (Weak schema satisfaction). We say that a Property Graph $G = (V, E, \rho, \lambda, \sigma)$ *weakly satisfies* a GraphQL schema $\mathcal{S}$ over $(\mathsf{F}, \mathsf{A}, \mathsf{T}, \mathsf{S}, \mathsf{D})$ if the following holds:

WS1 (Node properties must be of the required type): For all $(v, \mathsf{f}) \in \mathrm{dom}(\sigma)$ such that $v \in V$, $\mathsf{f} \in fields_{\mathcal{S}}(\lambda(v))$, and $\mathsf{t} = type^{\mathsf{F}}_{\mathcal{S}}(\lambda(v), \mathsf{f}) \in \mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$, it holds that $\sigma(v, \mathsf{f}) \in values_{\mathsf{W}}(\mathsf{t})$.

WS2 (Edge properties must be of the required type): For all $(e, \mathsf{a}) \in \mathrm{dom}(\sigma)$ s.t. $e \in E$, with $(v_1, v_2) = \rho(e)$, $f = (\lambda(v_1), \lambda(e))$, and $\mathsf{a} \in args_{\mathcal{S}}(f)$, it holds that $\sigma(e, \mathsf{a}) \in values_{\mathsf{W}}(type^{\mathsf{AF}}_{\mathcal{S}}(f, \mathsf{a}))$.

WS3 (Target nodes must be of the required type): For every $e \in E$ with $\rho(e) = (v_1, v_2)$ and $f = (\lambda(v_1), \lambda(e)) \in \mathrm{dom}(type^{\mathsf{F}}_{\mathcal{S}})$, it holds that $\lambda(v_2) \sqsubseteq_{\mathcal{S}} basetype(type^{\mathsf{F}}_{\mathcal{S}}(f))$.

WS4 (Non-list fields contain at most one edge): For all edges $e_1, e_2 \in E$ with $\rho(e_1) = (v_1, v_2)$, $\rho(e_2) = (v_1, v_3)$, $\lambda(e_1) = \lambda(e_2) = \mathsf{f}$, and $type^{\mathsf{F}}_{\mathcal{S}}(\lambda(v_1), \mathsf{f})$ is not a list type or a list type wrapped in non-null type, it holds that $e_1 = e_2$.

Definition 5.2 (Directives satisfaction). We say that a Property Graph $G = (V, E, \rho, \lambda, \sigma)$ *satisfies the directives* a GraphQL schema $\mathcal{S}$ over $(\mathsf{F}, \mathsf{A}, \mathsf{T}, \mathsf{S}, \mathsf{D})$ if the following holds:

DS1 (Edges identified by nodes and label): If $(\texttt{@distinct}, \emptyset) \in directives_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$, then for all edges $e_1, e_2 \in E$ with $\rho(e_1) = (v_1, v_2)$ and $\rho(e_2) = (v_1, v_2)$ such that $\lambda(e_1) \sqsubseteq_{\mathcal{S}} \mathsf{t}$ and $\lambda(e_1) = \lambda(e_2) = \mathsf{f}$, it holds that $e_1 = e_2$.

DS2 (No loops): If $(\texttt{@noLoops}, \emptyset) \in directives_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$, then there is no edge $e \in E$ with $\rho(e) = (v, v)$ such that $\lambda(v) \sqsubseteq_{\mathcal{S}} \mathsf{t}$ and $\lambda(e) = \mathsf{f}$.

DS3 (Target has at most one incoming edge): If $(\texttt{@uniqueForTarget}, \emptyset) \in directives_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$, then for all $e_1, e_2 \in E$ with $\rho(e_1) = (v_1, v_3)$ and $\rho(e_2) = (v_2, v_3)$ such that $\lambda(v_1) \sqsubseteq_{\mathcal{S}} \mathsf{t}$, $\lambda(v_2) \sqsubseteq_{\mathcal{S}} type_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$, and $\lambda(e_1) = \lambda(e_2) = \mathsf{f}$, it holds that $e_1 = e_2$.

DS4 (Target has at least one incoming edge): If $(\texttt{@requiredForTarget}, \emptyset) \in directives_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$, then for all $v_2 \in V$ such that $\lambda(v_2) \sqsubseteq_{\mathcal{S}} type_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$, there is at least one edge $e \in E$ with $\rho(e) = (v_1, v_2)$ such that $\lambda(v_1) \sqsubseteq_{\mathcal{S}} \mathsf{t}$ and $\lambda(e) = \mathsf{f}$.

DS5 (Property is required): If $(\texttt{@required}, \emptyset) \in directives_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$ and $type_{\mathcal{S}}(\mathsf{t}, \mathsf{f}) \in \mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$, then for every $v \in V$ such that $\lambda(v) \sqsubseteq_{\mathcal{S}} \mathsf{t}$, it holds that 1) $(v, \mathsf{f}) \in \mathrm{dom}(\sigma)$ and 2) $\sigma(v, \mathsf{f})$ is a nonempty list if $type_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$ is a list type.

DS6 (Edge is required): If $(\texttt{@required}, \emptyset) \in directives_{\mathcal{S}}(\mathsf{t}, \mathsf{f})$ and $type_{\mathcal{S}}(\mathsf{t}, \mathsf{f}) \notin \mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$, then for every $v_1 \in V$ with $\lambda(v_1) \sqsubseteq_{\mathcal{S}} \mathsf{t}$, there is at least one edge $e \in E$ with $\rho(e) = (v_1, v_2)$ such that $\lambda(e) = \mathsf{f}$.

DS7 (Keys): If $(\texttt{@key}, \{\texttt{fields}:[\mathsf{f}_1, \ldots, \mathsf{f}_n]\}) \in directives_{\mathcal{S}}(\mathsf{t})$, then for every two nodes $v_1, v_2 \in V$ such that $v_1 = v_2$ if
- $\lambda(v_1) \sqsubseteq_{\mathcal{S}} \mathsf{t}$ and $\lambda(v_2) \sqsubseteq_{\mathcal{S}} \mathsf{t}$, and
- for all $i \in \{1, \ldots, n\}$ such that $type_{\mathcal{S}}(\mathsf{t}, \mathsf{f}_i) \in \mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$, holds
  (i) $(v_1, \mathsf{f}_i) \notin \mathrm{dom}(\sigma)$ and $(v_2, \mathsf{f}_i) \notin \mathrm{dom}(\sigma)$, or
  (ii) $\{(v_1, \mathsf{f}_i), (v_2, \mathsf{f}_i)\} \subseteq \mathrm{dom}(\sigma)$ and $\sigma(v_1, \mathsf{f}_i) = \sigma(v_2, \mathsf{f}_i)$.

Definition 5.3 (Strong schema satisfaction). We say that a Property Graph $G = (V, E, \rho, \lambda, \sigma)$ *strongly satisfies* a GraphQL schema $\mathcal{S}$ over $(\mathsf{F}, \mathsf{A}, \mathsf{T}, \mathsf{S}, \mathsf{D})$ if it weakly satisfies it, satisfies its directives and the following holds:

SS1 (All nodes are justified): For all $v \in V$, it holds that $\lambda(v) \in \mathsf{O}_{\mathsf{T}}$.

SS2 (All node properties are justified): For all $(v, \mathsf{f}) \in \mathrm{dom}(\sigma)$ with $v \in V$, it holds $\mathsf{f} \in fields_{\mathcal{S}}(\lambda(v))$ and $type^{\mathsf{F}}_{\mathcal{S}}(\lambda(v), \mathsf{f}) \in \mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$.

SS3 (All edge properties are justified): For all $(e, \mathsf{a}) \in \mathrm{dom}(\sigma)$ with $\rho(e) = (v_1, v_2)$, it holds that $\mathsf{a} \in args_{\mathcal{S}}((\lambda(v_1), \lambda(e))$.

SS4 (All edges are justified): For all $e \in E$ with $\rho(e) = (v_1, v_2)$, it holds that $\lambda(e) \in fields_{\mathcal{S}}(\lambda(v_1))$ and $type^{\mathsf{F}}_{\mathcal{S}}(\lambda(v_1), \lambda(e)) \notin \mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$.

6 FORMAL ANALYSIS OF THE APPROACH

In this section we analyze the computational complexity of the presented approach. We focus on two computational problems: *schema validation* and *schema satisfiability*. The first problem concerns deciding for a given schema and Property Graph whether the graph strongly satisfies the schema. The second problem concerns deciding for a given schema and some element of that schema such as an object type, a field, or an argument, if there is a Property Graph that strongly satisfies the schema and populates that element.

6.1 Validation

Formally, the *schema validation problem* is defined as follows:

THE SCHEMA VALIDATION PROBLEM	
Input:	The sets $(\mathsf{F}, \mathsf{A}, \mathsf{T}, \mathsf{S}, \mathsf{D})$, a GraphQL schema $\mathcal{S}$ over those sets, and a Property Graph G.
Question:	Does G strongly satisfy $\mathcal{S}$?

It can be shown that this problem has a low computational complexity and is highly parallelizable:

Theorem 1. *Under the assumption that* `Scalars` *is a fixed finite set and the problem of deciding if* $v \in values(\mathtt{t})$ *for a scalar value* $v \in \mathtt{Vals}$ *and a scalar type* $\mathtt{t} \in \mathtt{Scalars}$ *is in* AC_0*, the computational complexity of the schema validation problem is in* AC_0.

If we interpret the Property Graph as a database instance and the schema as a boolean query that is computed over the instance, then the previous result can be interpreted as a *combined complexity* result. Although a theoretically pleasing result, it does not immediately suggest a practical algorithm. However, from the observations in the proof it follows that a straightforward implementation of the first-order logical formulas leads already to a tractable algorithm with time complexity $O(n^3)$ and space complexity $O(\log(n))$. Moreover, if we look at *data complexity* and fix the schema, it can be verified that none of the rules has more than two nested quantifiers that quantify over elements of the Property Graph; and so under this perspective the time complexity of that algorithm is in $O(n^2)$.

6.2 Satisfiability

An important soundness property of a schema is that every part of the schema can be populated, i.e., for every type and field definition there is at least one Property Graph that contains nodes, properties or edges that instantiate that definition. This problem has been studied in different settings such as the Entity-Relationship Model [5], Object-oriented database schemas [8, 10] and UML Class diagrams [6]. Unfortunately, it can have a high computational complexity, even for schema formalisms with a relatively low expressive power. This problem may also not be trivial in our formalism as the following examples illustrate.

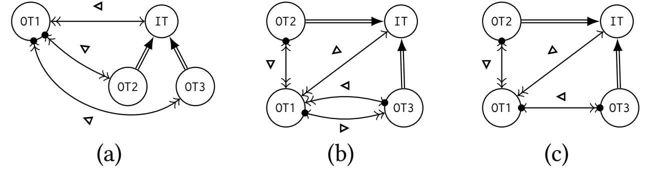
Example 6.1. Consider the following schema definition.

```
1  type OT1 {
2  }
3
4  interface IT {
5    hasOT1: OT1  @uniqueForTarget
6  }
7
```

```
8  type OT2 implements IT {
9   hasOT1: [OT1]  @requiredForTarget
10 }
11
12 type OT3 implements IT {
13  hasOT1: [OT1]  @requiredForTarget
14 }
```

This schema contains a conflict for type `OT1`. Assume a Property Graph that has a node, say v_0, with label `OT1`. Because of the `@requiredForTarget` in the definition of `OT2`, v_0 must have an incoming edge with label `hasOT1` from a node, say v_1, with label `OT2`. Similarly, v_0 must also have an incoming edge with label `hasOT1` from a node, say v_2, with label `OT3`. Since `OT2` and `OT3` both implement the interface type `IT`, it holds that v_1 and v_2 are also of type `IT`. The `@uniqueForTarget` in the definition of `IT` implies that any node of type `OT1` can have at most one incoming edge from a node of type `IT`. It then follows that $v_1 = v_2$, but this leads to a contradiction since v_1 and v_2 were assumed to be labeled with `OT2` and `OT3`, respectively, and a node can have only one label.

The previous schema can also be represented graphically, as is shown in diagram (a) in the following figure.



(a) (b) (c)

Here the double or single heads of the edges indicate if the relation is one-to-many, many-to-many, one-to-one, or many-to-one. The triangle illustrates on which side the relation is defined, pointing from source to target. The black dot illustrates on which sides the relation is required. The double edges indicate the *implements* and *is-part-of-union* relationship. We omit the field name associated with the relation, since this is in all cases the same.

In diagrams (b) and (c) we see other examples of conflicts: in both diagrams the type `OT2` is not satisfiable. In diagram (b) the problem is that any satisfying Property Graph with an `OT2` node must have an infinite alternating chain of `OT1` and `OT3` nodes where none of these nodes are allowed to be identical. The problem in diagram (c) is that any node in `OT2` must be identical to an `OT3` node to which it must be connected to via an `OT1` node.

The main problem that we focus on here is the one illustrated in the previous examples, namely that of object type satisfiability.

> THE OBJECT-TYPE SATISFIABILITY PROBLEM
> **Input:** The sets $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S}, \mathtt{D})$, a GraphQL schema $\mathcal{S}$ over those sets, and an object type `ot` in $\mathcal{S}$.
> **Question:** Is there a Property Graph $(V, E, \rho, \lambda, \sigma)$ that strongly satisfies $\mathcal{S}$ and contains at least one node $v \in V$ such that $\lambda(v) = \mathtt{ot}$?

Unfortunately it can be shown that this problem is NP-hard.

Theorem 2. *The object-type satisfiability problem is NP-hard.*

The following theorem gives an upper bound for the complexity.

Theorem 3. *The object-type satisfiability problem is in PSPACE.*

We conclude with briefly discussing the satisfiability of other schema components. The satisfiability of interface and union types is directly linked to the satisfiability of their implementing object types and union components. The satisfiability problem for properties is trivial because of the consistency requirements for schemas. Finally, the satisfiability of edge definitions is reducible to the problem of type satisfiability: add the `@required` to the field definition and check if the type of the field definition is satisfiable.

7 CONCLUDING REMARKS

We have presented an approach that uses GraphQL schemas as Property Graph schemas. To this end, we have given GraphQL schemas a formal semantics and investigated the computational complexity of schema validation and schema satisfiability. In future work we aim to extend the approach for GraphQL APIs and characterize the complexity of schema satisfiability more precisely.

Acknowledgements. Hartig's work has been funded by the EU's Horizon 2020 research and innovation programme (grant no. 786993).

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Paul Andlinger. 2015. Graph DBMS increased their popularity by 500% within the last 2 years. Online at http://db-engines.com/en/blog_post/43. (March 2015).

[3] Renzo Angles. 2018. The Property Graph Database Model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management (CEUR Workshop Proceedings)*, Dan Olteanu and Barbara Poblete (Eds.). Cali, Colombia. http://ceur-ws.org/Vol-2100/paper26.pdf

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[5] A. Artale, D. Calvanese, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev. 2007. Reasoning over Extended ER Models. In *Conceptual Modeling - ER 2007*, Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim (Eds.). Vol. 4801. Springer Berlin Heidelberg, Berlin, Heidelberg, 277–292. https://doi.org/10.1007/978-3-540-75563-0_20

[6] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. 2005. Reasoning on UML Class Diagrams. *Artif. Intell.* 168, 1-2 (Oct. 2005), 70–118. https://doi.org/10.1016/j.artint.2005.05.003

[7] Angela Bonifati, G.H.L. Fletcher, Hannes Voigt, and N. Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers.

[8] Diego Calvanese and Maurizio Lenzerini. 1994. Making Object-oriented Schemas More Expressive. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '94).* ACM, New York, NY, USA, 243–254. https://doi.org/10.1145/182591.182620 event-place: Minneapolis, Minnesota, USA.

[9] Facebook, Inc. 2018. GraphQL – June 2018 Edition. Online at https://facebook.github.io/graphql/June2018/. (June 2018).

[10] Anna Formica. 2003. Satisfiability of object-oriented database constraints with set and bag attributes. *Information Systems* 28, 3 (May 2003), 213–224. https://doi.org/10.1016/S0306-4379(02)00010-8

[11] Olaf Hartig. 2014. Reconciliation of RDF* and Property Graphs. *CoRR* arXiv/1409.3288 (2014).

[12] Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In *Proceedings of The Web Conference 2018 (27th International World Wide Web Conference).*

[13] JanusGraph Authors. 2018. JanusGraph Documentation v.0.3.1. Online at https://docs.janusgraph.org/0.3.1/index.html. (2018).

[14] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). W3C Recommendation, Online at https://www.w3.org/TR/shacl/. (July 2017).

[15] Neo4j, Inc. 2019. The Neo4j Cypher Manual v3.5. Online at https://neo4j.com/docs/cypher-manual/3.5/. (2019).

[16] Eric Prud'hommeaux, Iovka Boneva, Jose Emilio Labra Gayo, and Gregg Kellogg. 2018. Shape Expressions Language 2.1. Draft Community Group Report, Online at http://shex.io/shex-semantics/. (Nov. 2018).

[17] Ian Robinson, Jim Webber, and Emil Eifrém. 2015. *Graph Databases* (2nd ed.). O'Reilly Media.

[18] Sparsity Technologies. 2015. Sparksee Starting Guide. Online at http://www.sparsity-technologies.com/StartingGuide/Index.html. (2015).

[19] TigerGraph, Inc. 2019. GSQL Language Reference, Part 1 – Data Definition & Loading. Online at https://docs.tigergraph.com/dev/gsql-ref/ddl-and-loading. (2019).

[20] Stephan Tobies. 2001. Complexity Results and Practical Algorithms for Logics in Knowledge Representation. *CoRR* cs.LO/0106031 (2001). http://arxiv.org/abs/cs.LO/0106031

APPENDIX

A OVERVIEW OF THE GRAPHQL SDL

To illustrate the main features of the GraphQL schema definition language (SDL) we refer to Figure 1. The language allows users to define so-called object types that have a name and a set of field definitions. Each field definition consists of a name and a value type. For instance, the example schema in Figure 1 contains an object type named `Starship` that has three field definitions for fields named `id`, `name`, and `length` (cf. lines 1–5). Conceptually, each field definition captures a data field that every object of the given type has and that requests to the corresponding GraphQL API can access by referring to the field name.

The value type of a field definition specifies what type of value the API will return when the field is requested. For instance, the `name` field of every `Starship` object will have a string as value. Additionally, the GraphQL specification assumes a special value called `null` that may also be returned instead of an actual value. A field can be defined to be "non-nullable" by adding an exclamation mark to the field definition, which means that for this field, `null` will not be returned. For instance, the API of the example schema will not return `null` for the `id` field of `Starship` objects.

Instead of values of a scalar type, fields may also be defined to return values of an enumeration type, of an object type, or of an interface or a union type. The latter are interesting as they allow for some limited form of type hierarchies that can consist of two levels, namely: unions over object types and, orthogonal to that, interfaces that are implemented by object types (i.e., union types and interface types cannot form hierarchies, but an object type may participate in multiple union types and it may implement multiple interface types). In the example schema we have a union type called `SearchResult` (lines 33–44), and we have an interface called `Character` (lines 9–13) which is implemented by the object types `Human` and `Droid`. Moreover, by wrapping the type of a field definition in square brackets (e.g., line 12), it can be indicated that for the corresponding field, the API may return not only a single value (of the type specified in the field definition) but a list of such values. For instance, for the field `friends` in the example, we can expect to retrieve a list of objects that are either `Human` or `Droid`.

Finally, field definitions may also contain definitions of so-called arguments (e.g., line 4) that then can be used in the requests. For instance, the value of the `length` field of a `Starship` object can be requested in meters or in feet (with meters being the default).

B PROOFS

Theorem 1 *Under the assumption that* `Scalars` *is a fixed finite set and the problem of deciding if* $v \in values(\mathtt{t})$ *for a scalar value* $v \in$ `Vals` *and a scalar type* $\mathtt{t} \in$ `Scalars` *is in* AC_0*, the computational complexity of the schema validation problem is in* AC_0*.*

Proof. (Sketch) The proof is based on the following two observations: (1) the input can be encoded in a first-order structure such that there is a family of AC_0 circuits with a fixed maximal depth that computes this encoding and (2) all rules for checking weak satisfaction, directive satisfaction and strong satisfaction can for that encoding be represented as boolean queries in the relational calculus, for which the evaluation complexity (i.e., computing their result on a give relational database) is known to be in AC_0 (See **Theorem** 17.1.2 in [1]). We elaborate on these two observations:

Encoding in first-order structure: The finite sets $(\mathsf{F}, \mathsf{A}, \mathsf{T}, \mathsf{S}, \mathsf{D})$ and be mapped to unary predicates $\mathsf{F}(f)$, $\mathsf{A}(a)$, $\mathsf{T}(t)$, $\mathsf{S}(s)$ and $\mathsf{D}(d)$. The components of the schema can be mapped as follows:

- $type^{\mathsf{F}}_{\mathcal{S}}$, $type^{\mathsf{AF}}_{\mathcal{S}}$ and $type^{\mathsf{AD}}_{\mathcal{S}}$ as
  – $\mathtt{typeF}(t, f, t')$, $\mathtt{typeAF}(t, f, a, t')$ and $\mathtt{typeAD}(d, a, t')$,
- $union_{\mathcal{S}}$ as $\mathtt{union}(t, t')$.
- $implementation_{\mathcal{S}}$ as $\mathtt{impl}(t, t')$.
- $directives^{\mathsf{T}}_{\mathcal{S}}$, $directives^{\mathsf{F}}_{\mathcal{S}}$ and $directives^{\mathsf{AF}}_{\mathcal{S}}$ as
  – $\mathtt{dirAF}(t, f, a, d, a', v)$, $\mathtt{dirT}(t, d, a, v)$ and $\mathtt{dirF}(t, f, d, a, v)$.

The components of a Property Graph $(V, E, \rho, \lambda, \sigma)$ can be mapped as follows:

- V as $\mathsf{V}(n)$, E as $\mathsf{E}(e)$ and ρ as $\mathtt{edge}(e, n_1, n_2)$.
- λ as $\mathtt{label}(x, l)$ and σ as $\mathtt{val}(x, p, v)$.

It is then not hard to see that under this mapping, for each predicate, there is a family of AC_0 circuits with a fixed maximal depth that maps a binary representation of the input to a binary representation of these predicates.

Mapping to first-order logic. It is not hard to see that most rules have a form that is expressible in first-order logic given the presented mapping to predicates. Some rules need some special consideration since they refer to predicates that are not in the first-order structure: $values_{\mathsf{W}}$ and $\sqsubseteq_{\mathcal{S}}$. We consider both of them:

- For $values_{\mathsf{W}}$: The number of types in $\mathsf{S} \cup \mathsf{W}_{\mathsf{S}}$ has a fixed finite size since $\mathsf{S} \subseteq \mathtt{Scalars}$ and `Scalars` is assumed to contain a fixed number of types, and W_{S} contains at most 6 times as many types as S because only 8 patterns of nesting are allowed. So we can assume that there is a family of AC_0 circuits with a fixed maximal depth for computing this predicate from the original input.
- For $\sqsubseteq_{\mathcal{S}}$: The implementation and union hierarchies are only one level deep, and the wrapped types are also at most 3 levels deep. So for the resulting predicate we can assume that there is a family of AC_0 circuits with some fixed maximal depth for computing this predicate.

Given the previous two observations we can construct a family of AC_0 circuits with a fixed maximal depth that compute the encoding of the original input into a first-order structure that also contains predicates for $values_{\mathsf{W}}$ and $\sqsubseteq_{\mathcal{S}}$. This can be combined with observation that we also have such a family for evaluation the formulas that define strong satisfiability, by simply concatenating the circuits for the encoding and the formula evaluation, which results in a family of AC_0 circuits that decide whether the given property graph satisfies the given GraphQL schema. □

Theorem 2 *The object-type satisfiability problem is NP-hard.*

Proof. (Sketch) The proof proceeds by showing that the SAT problem, satisfiability of propositional formulas in conjunctive normal form, can be reduced that the object-type satisfiability problem. We do this by constructing a schema with a type `ot` such that this type is satisfiable iff the propositional formula φ is satisfiable. This construction picks a field name f and proceeds as follows:

(1) We introduce an object type `ot`.

```
1  type Starship {
2    id: ID!
3    name: String
4    length(unit:LenUnit = METER): Float
5  }
6
7  enum LenUnit { METER FEET }
8
9  interface Character {
10   id: ID!
11   name: String
12   friends: [Character]
13 }
```

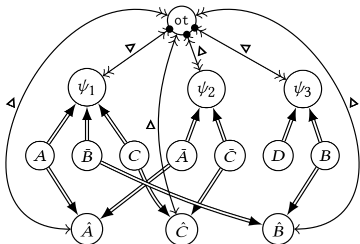
```
13 type Human implements Character {
14   id: ID!
15   name: String
16   friends: [Character]
17   starships: [Starship]
18 }
19
20 type Droid implements Character {
21   id: ID!
22   name: String
23   friends: [Character]
24   primaryFunction: String!
25 }
```

```
26 type Query {
27   hero(episode: Episode): Character
28   search(text: String): [SearchResult]
29 }
30
31 enum Episode { NEWHOPE EMPIRE JEDI }
32
33 union SearchResult = Human | Droid |
34                      Starship
35 schema {
36   query: Query
37 }
38
```

Figure 1: Example GraphQL schema written in the GraphQL schema definition language (example adapted from [12]).

(2) If $\varphi = \psi_1 \land \ldots \land \psi_n$, we introduce an interface type $\mathtt{i}_i$ for each $1 \leq i \leq n$.
(3) For each type $\mathtt{t}_i$ a field f : [ot] is defined with directive `@requiredForTarget`.
(4) If $\psi_i = \alpha_{i,1} \lor \ldots \lor \alpha_{i,m_i}$ with each $\alpha_{i,j}$ possibly negated propositional variables, then we create for each $\alpha_{i,j}$ an object type $\mathtt{ot}_{i,j}$ and declare it as implementing $\mathtt{i}_i$.
(5) For any two atoms $\alpha_{i,j}$ and $\alpha_{i',j'}$ such that $\alpha_{i,j} = \neg\alpha_{i',j'}$, we create an interface type $\mathtt{i}_{i,j,i',j'}$, and (1) declare the types $\mathtt{ot}_{i,j}$ and $\mathtt{ot}_{i',j'}$ as implementations of this interface type and (2) define with each a field f : [ot] with directive `@uniqueForTarget`.

For example, the formula $(A \lor \neg B \lor C) \land (\neg A \lor \neg C) \land (D \lor B)$ is translated to the following schema (where the types are shown labeled with the part of the formula they represent, and $\hat{A}$ represents the type that encodes that atoms A and $\neg A$ conflict):



It is not hard to see that (1) the resulting schema has a size that is polynomial in the size of φ and (2) there is a property graph with a node with label ot that strongly satisfies this schema iff φ is satisfiable, since the nodes associated with the node in type ot define a propositional truth assignment that satisfies φ and vice versa. □

Theorem 3 *The object-type satisfiability problem is in PSPACE.*

Proof. We first consider schemas without the `@key`, `@noLoops` and scalar-valued fields and arguments, which translate to properties of nodes and edges. Such schemas can be simulated the description logic called $\mathcal{ALCQI}$. This description logic has the standard $\mathcal{ALC}$ constructs, such as $\top$ (top), $\bot$ (bottom), A (a concept name), $\neg C$ (negation of a concept), $C \sqcap D$ (intersection of two concepts), $C \sqcup D$ (union of two concepts), $\exists R.C$ (existential qualification) and $\forall R.C$ (universal qualification). Next to that, $\mathcal{ALCQI}$ contains qualified number restrictions such as $\geq_n R.C$ and $\leq_n R.C$, and allows the usage of an inverse role R^- in expressions where a role is expected.

This description logic allows us to simulate the constructs in our schemas:

- If t is a union type over $\mathtt{t}_1, \ldots, \mathtt{t}_n$, or an interface type that is implemented by these types, this can be expressed as $\mathtt{ut} \equiv \mathtt{t}_1 \cup \ldots \cup \mathtt{t}_n$.
- If t has a non-scalar field with name f and with base type tt, this can be expressed as $(\exists f^-.\mathtt{t}) \sqsubseteq \mathtt{tt}$. If the type of the field is not a list type, and the edge therefore required, this can be stated as $\mathtt{t} \sqsubseteq (\leq_1 f.\mathtt{tt})$
- If the field is marked as `@required`, this can be stated as $\mathtt{t} \sqsubseteq (\exists f.\mathtt{tt})$.
- If the field is marked as `@requiredForTarget`, this can be stated as $\mathtt{tt} \sqsubseteq (\exists f^-.\mathtt{t})$.
- If the field is marked as `@uniqueForTarget`, this can be stated as $\mathtt{tt} \sqsubseteq (\leq_1 f^-.\mathtt{t})$.
- That fact that nodes belong to exactly one objects type can be expressed as (1) $\mathtt{ot1} \sqcap \mathtt{ot2} \equiv \bot$ for each distinct pair ot1 and ot2 of object types, and (2) $\top \equiv \mathtt{ot}_1 \sqcup \ldots \sqcup \mathtt{ot}_n$ where $\mathtt{ot}_1, \ldots, \mathtt{ot}_n$ is the list of all object types.

Note that we ignore `@distinct` directives, but that is because in this description logic all edges identified by their begin and end nodes. Note, however, that in this stage this does not matter because we do not consider properties, so we can merge or multiply edges with the same nodes without affecting the satisfaction of the schema. It can then be concluded that an object type in a schema is satisfiable iff its corresponding concept is satisfiable in the translation to $\mathcal{ALCQI}$.

Moreover, if we add back the constraints for scalar-valued fields and arguments, this will also not change the satisfiability since we can always assign their values such that they are of the right type. The same holds for the `@key` directives: assuming that all scalar types have an infinite set of values we can always pick the values of the involved properties such that the key constraints hold. Finally also the `@noLoops` does not affect satisfiability since we can always remove loops as follows: make a distinct copy of the property graph, remove the loop and replace it with identical back and forth edges between the loop node and its copy.

It follows that we can used the described translation to decide if a certain object type is satisfiable. Since the size of the translated schema is polynomial in the size of the original schema, it follows from the PSPACE upper bound for the problem of concept satisfiability in $\mathcal{ALCQI}$ [20, Theorem 4.29] that the object-type satisfiability problem also has a PSPACE upper bound. □