One minute was enough, Tyler said, a person had to work hard for it, but a minute of perfection was worth the effort. A moment was the most you could ever expect from perfection.

Chuck Palahniuk - Fight Club

5

Athena: A Schema Definition Language

In a multi-model environment, unified or generic metamodels such as ER [Che76] are commonly used. These generic metamodels are defined with the purpose of representing schemas of different data models in the same format. As shown in Chapter 4, the U-Schema metamodel was mainly conceived to represent inferred schemas from NoSQL stores and relational systems. However, even when the majority of NoSQL databases are schemaless, the possibility of working with schemas is a convenient commodity, as noted in [Des20].

Athena is a domain specific language (*DSL*) built upon the U-Schema metamodel to provide a generic schema definition language with high expressive power able to specify database-independent schemas. Athena models provide a textual representation for a better expressiveness of schemas, and can be applied in applications and utilities that require developers to define schemas from scratch independent to any database or data model. This language also proves its usefulness in scenarios in which schema inference cannot be applied, as in data generation for testing purposes, or schema manipulation when there is no database whose schema can be inferred.

Since Athena is based on the U-Schema metamodel, it includes a rich set of abstractions (e.g., structural variations, references, and relationship types). Moreover, Athena offers mechanisms for reusing and composing schemas, as well as syntactic sugar to ease writing them: defining types with incomplete information and restrictions, working with a large number of

variations, or declaring features without types, among others.

The Athena language will be the base layer on which the rest of the tools developed in this thesis will be based, as will be shown in the following chapters. The aim of this chapter is to provide an in-depth analysis of the Athena language, from the gathering of requirements and its implementation to transformations involving Athena schemas and artifact generation.

This chapter has been organized as follows: First, we compose a list of requirements to be fulfilled by the language. Next, we introduce the Athena language detailing its concrete syntax and how requirements are satisfied. In the final section we analyze available transformations and artifact generation from Athena schemas.

5.1 Language Requirements

While obtaining U-Schema models from existing databases is automatically performed by an inference tool, as discussed in Chapter 4, when designing schemas, or changing existing ones, using a language to easily manipulate such schemas provides several advantages. Athena has been designed as a DSL to tackle this need by taking into account the following requirements:

Requirement 1 *The language should be **complete**, and adhere to the **principle of least astonishment**.*

The language will be *complete* if each element of the U-Schema data model can be mapped in Athena. This assures that any U-Schema model can be described by Athena. On the other hand, the *principle of least astonishment* [Blo06] proposes that the result of performing some operation should be obvious, consistent, and predictable, based upon the name of the operation and other clues. The interpretation of this principle in the context of language design is that every construction of the language must behave as obvious and predictable as possible.

Requirement 2 *The language should favor reusability, including mechanisms for importing, inheritance, and versioning.*

When declaring schemas, creators should be able to reuse elements of previously defined schemas. With this purpose any element susceptible of being reused (e.g., schema types or schema variations) should be assigned an identifier that can be uniquely referenced.

Requirement 3 *The language should treat **Structure** as a first class element.*

This means that in order to favor extensibility and composability, *structures* should be able to be created anonymously, and manipulated using specific operators, i.e., generating a new structure by compositing other structures, building schema types using structures, and so on.

Requirement 4 *The language should be able to cope with incomplete information.*

The language should support typeless features, aggregations of unknown variations, or Option types. This feature allows schemas to be created without the comprehension of all of its details and to apply processes tolerant to this lack of knowledge.

Requirement 5 *The language should allow to specify restrictions on types.*

We want to provide the language of mechanisms to define restrictions on types to gain expressivity. Restrictions can be applied to primitive types, and some examples can be regular expressions and enumerate values for Strings, ranges for Numbers, etc. The language must also provide extension mechanisms to define additional restrictions.

Requirement 6 *The language should allow the specification of* ***Union schemas****.*

In addition to specifying schema types as sets of variations (as specified in the U-Schema data model) *union schema types* should be able to be defined as a list of features flagged as *required* or *optional*. This mechanism will prove its usefulness to support systems where structural variations come from the usage of optional features.

Requirement 7 *The language should allow the usage of other established schema specification languages when defining schemas.*

In order to be easy to adopt by developers, the language should allow the possibility to embed fragments of other schema specification languages, such as SQL or JSON Schema [D+22]. This will also enhance the integration of Athena with existing languages. The language should be open to be integrated with other languages in the future.

To sum up, our main goal when designing the language is to allow it to provide as much syntax mechanisms as possible to favor flexibility, ease its learning curve and be friendly with developers used to work with other data models.

5.2 Implementation of the Athena Language

Athena has been implemented using the Xtext workbench [Bet16]. We have applied metamodeling to define the language: First, an abstract syntax has been defined for the language in form of an Ecore metamodel [SBPM09]. Then, the concrete syntax has been defined as a Xtext grammar. With that, the Xtext engine automatically generated the editor, parser, and model injector. The stack of technologies used and the artifacts generated automatically can be seen in Figure 5.1. A translational approach was applied to define the semantics: Athena models are transformed to U-Schema models, and then utilities can be built for Athena.

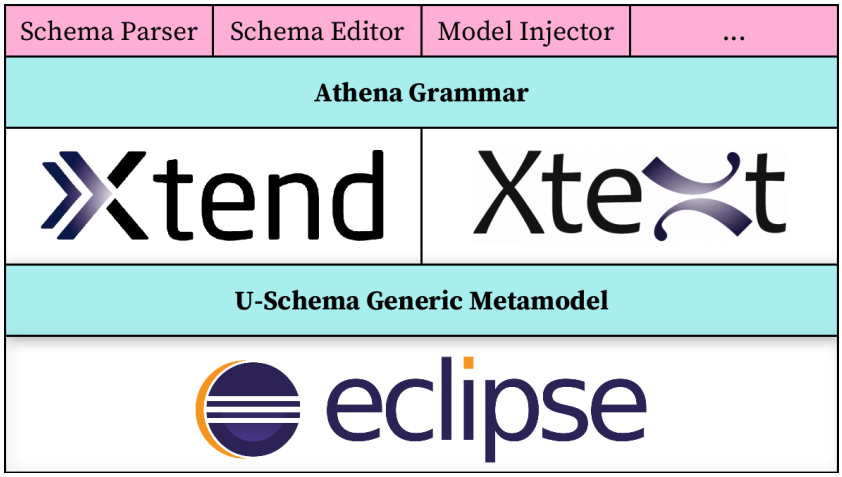

Figure 5.1: Stack of technologies used in Athena and the generated artifacts.

5.2.1 Grammar

Since the Athena metamodel is too large to be shown, we consider an excerpt of the grammar notation to understand the language. Once some notions about the notation are provided, we will show a running example that will be useful to illustrate all the syntactic mechanisms Athena provides in detail. Later, mappings between U-Schema and Athena will be shown.

Figure 5.2 shows an excerpt of the Athena grammar in EBNF format. An Athena schema (`<AthenaSchema>`) is organized in three parts: A header specifying the schema name, an optional set of import statements, and a set of type definitions. In the next section we will detail each part along with the running example. The majority of elements defined with Athena can have additional arguments compared to their equivalents in U-Schema or

can be defined in several ways in order to achieve a more writable syntax. For example, an entity type (`EntityDecl`) can be defined by using either the `<ShortEntityDecl>` rule or the `<RegularEntityDecl>` rule. Each one of these rules end up producing one or more `<StructurePrimary>`, `StructureExpr` or `StructureLiteral` rules, which hold the content of the schema type being defined. Elements of the grammar are usually defined by a keyword (e.g., `entity` in case of entity types, `variation` for variations, `List` for the list types and so on) and some optional parameters. Also a subset of the SQL language has been included in the `<SQLStructure>` rule to allow the usage of SQL statements inside a structure. The complete Athena grammar can be found in Appendix 10.4.1.

In order to better reflect the usage of the Athena language, we will introduce the *SoftwareDev* schema developed with Athena, shown in Figure 5.3. In this example, two feature sets are defined: `Popularity_features` and `Timestamp_features`, by using the `<FeatureSetDecl>` rule, and four entity types: `Developer`, `DeveloperInfo`, `Repository` and `Ticket`, by using the `<EntityDecl>` rule.

5.2.2 Syntactic Mechanisms

Athena has been enriched with several syntactic mechanisms to provide flexibility for developers to build schemas in intuitive ways. Here, we will overview these mechanisms and map them with the requirements they are related to.

Schema Identifiers Several mechanisms of the language, such as schema importing and versioning, require assigning identifiers to schemas. A schema `identifier` is composed of a name and a version number separated by a colon. The version number allows the evolution of a schema to be tracked over time. It is initialized to 1, and it will be incremented as new schema versions are generated. In the running example, the schema is named *SoftwareDev* and it is in its first version. Once new schema versions are generated, they will be named *SoftwareDev:2*, *SoftwareDev:3* and so on. This mechanism will help satisfy Requirement 2.

Importing Schemas By using the previous mechanism, schemas can be uniquely identified. Schemas can then be imported using their identifier. When declaring a schema, all the types of the imported schemas can be directly used. Figure 5.4 shows a schema that imports *SoftwareDev:1* and therefore elements of this new schema can reference and inherit any elements defined in *SoftwareDev:1* by using their identifiers. This feature

⟨*AthenaSchema*⟩ ::= 'Schema' ⟨*SchemaId*⟩ ('import' ⟨*Import*⟩)* (⟨*FeatureSetDecl*⟩ | ⟨*EntityDecl*⟩ | ⟨*RelationshipDecl*⟩)*

⟨*EntityDecl*⟩ ::= ⟨*ShortEntityDecl*⟩
| ⟨*RegularEntityDecl*⟩

⟨*ShortEntityDecl*⟩ ::= 'root'? 'entity' ID ('::' ID (',' ID)*)? ⟨*StructureExpr*⟩

⟨*RegularEntityDecl*⟩ ::= 'root'? 'entity' ID ('::' ID (',' ID)*)? '{' ⟨*CommonSpec*⟩? ⟨*VariationDecl*⟩+ '}'

⟨*CommonSpec*⟩ ::= 'common' ⟨*StructureExpr*⟩

⟨*VariationDecl*⟩ ::= 'variation' INT ⟨*StructureExpr*⟩?

⟨*StructurePrimary*⟩ ::= ⟨*StructureLiteral*⟩
| ⟨*TopLevelStructureDefiningElementDeclRef*⟩
| ⟨*VariationDeclRef*⟩
| ⟨*SQLStructure*⟩
| '(' ⟨*StructureExpr*⟩ ')'

⟨*SQLStructure*⟩ ::= 'SQL' 'CREATE' 'TEMPORARY'? 'TABLE' ('IF' 'NOT' 'EXISTS')? ID? '(' ⟨*SQLDefinition*⟩ (',' ⟨*SQLDefinition*⟩)* ')' ';'

⟨*FeatureSet*⟩ ::= ⟨*Feature*⟩ (',' ⟨*Feature*⟩)*

⟨*Feature*⟩ ::= ⟨*ComposedReference*⟩
| ⟨*SimpleFeature*⟩

⟨*SimpleFeature*⟩ ::= ('+' | '?'? '!'?) ID (':' ⟨*Type*⟩)?

⟨*Type*⟩ ::= ⟨*DataType*⟩
| ⟨*SimpleReferenceTarget*⟩
| ⟨*SimpleAggregateTarget*⟩
| ⟨*InnerStructureLiteralArray*⟩
| ⟨*InnerStructureLiteral*⟩

⟨*DataType*⟩ ::= 'Null'
| ⟨*PrimitiveType*⟩
| ⟨*List*⟩
| ⟨*Set*⟩
| ⟨*Map*⟩
| ⟨*Tuple*⟩

Figure 5.2: Excerpt of the Athena language in EBNF format.

helps contributing to Requirement 2. In this example, `Project.repositories` and `ProjectDeveloper.solved_tickets` are references to `Repository` and `Ticket`, respectively, defined in *SoftwareDev:1*.

```
Schema SoftwareDev:1

FSet Popularity_features {
 num_forks:          Integer ( 0..1000 ),
 num_stars:          Integer ( 0..1000 )
}

FSet Timestamp_features {
 created_time:       Timestamp,
 last_activity_date: Timestamp
}

Root entity Developer {
 Common
 {
  + id:              Identifier,
  dev_info:          Aggr<DeveloperInfo>&,
  ! email:           String /^.+@.+\\.com$/,
  permissions:       String in
              ("R","W","X","RW","RX","WX","RWX")
 }
 Variation 1
  { is_active:       Boolean }
 Variation 2
  { suspended_acc:   Option<String, Boolean> }
}

Entity DeveloperInfo {
 ? about_me:         String
 name:               String /^[A-Z][a-z]*$/,
 team:               String
}
```

```
Root entity Repository
{
 + id:               Identifier,
 developers:         Ref<Developer>+,
 requests:
 [
  {
   branch:           String,
   status:           String in ("Open", "Closed"),
   numLabels:        Number
  }
 ],
 ? tags:             List<String>,
 title:              String,
 url:                String
} U Popularity_features
  U Timestamp_features

Root entity Ticket
 SQL CREATE TABLE Ticket (
  id            VARCHAR(255) NOT NULL,
  message       VARCHAR(255) NOT NULL,
  repository_id VARCHAR(255) NOT NULL,
  developer_id  VARCHAR(255) NOT NULL,
  FOREIGN KEY ( repository_id )
                REFERENCES Repository ( id ),
  FOREIGN KEY ( developer_id )
                REFERENCES Developer ( id ),
  PRIMARY KEY ( id )
 );
 + Popularity_features
 + Timestamp_features
```

Figure 5.3: The *SoftwareDev* schema defined using Athena.

```
Schema SoftwareProject:1

Import SoftwareDev:1

Root entity Project {
  + id:          Identifier,
  developers:    Ref<ProjectDeveloper>+,
  due_date:      Timestamp,
  name:          String,
  repositories:  Ref<SoftwareDev:1.Repository>+
}
```

```
Root entity ProjectManager::SoftwareDev:1.Developer
{
 in_active:        Boolean,
 managed_projects: Integer
}

Root entity ProjectDeveloper::SoftwareDev:1.Developer
{
 languages:        List<String>,
 solved_tickets:   Ref<Ticket>+
}
```

Figure 5.4: The *SoftwareProject* schema designed as an extension of *SoftwareDev*.

Schema Types A schema is formed by a set of `Entity types` and `Relationship types`. Both schema types are composed by a set of features. In addition, the `Feature Set` construct is provided to group features with the purpose of being reused in the definition of other types. In the running example of Figure 5.3, four entity types are defined: `Developer`, `Repository`,

`Ticket`, and `DeveloperInfo`. The former three specify root entities, while the latter specifies an aggregate type used to declare a `Developer` feature. Also, the schema includes two feature sets (`FSet`) named `Popularity_features` and `Timestamp_features`.

When a schema type has more than one variation, each one of them must be explicitly declared, as shown in the `Developer` entity type of the running example. Instead of specifying variations, a schema type can also be defined as a *union schema type* by providing a set of features labeled as required or optional (Requirement 6). In the running example, `Repository` is a *union schema type*, with five required features and `tags` as an optional feature.

Structures To satisfy Requirement 3, Athena allows the definition of *structures* in order to ease type definitions. A structure is defined by means of an *expression*. A set of structure operators has been defined to create structures as a composition of others, such as *union* ("`U`" or "`+`"), *intersection* ("`I`"), and *difference* ("`-`"). Each operand may be a schema type, a variation, a feature set (that can be previously defined or can be defined inline) or a SQL statement (Requirement 7).

For example, in Figure 5.3, the structures of `Ticket` and `Repository` are formed by adding their own structure to the structure of `Popularity_features` and `Timestamp_features`, and `Ticket` is defined by using a SQL statement as a structure.

Inheritance Mechanism The language also provides a type inheritance mechanism, which allows to establish a *child-parent* relationship between types (Requirement 2). Each feature defined in the parent type will be included in the child type. This mechanism can also be used to allow schema types to inherit from schema types defined in other schemas. In Figure 5.4, two inheritance relationships exist in which `ProjectManager` and `ProjectDeveloper` inherit from the `SoftwareDev:1.Developer` entity type.

Features A feature is declared by a name and a type, and the type is not mandatory (Requirement 4). A type can denote an `Aggregation`, a `Reference`, a scalar data type, a nested type that will be transformed into an aggregate type, or an array of types. In Figure 5.3, the `Repository.developer_id` feature references `Developer`, and `Developer.dev_info` aggregates `DeveloperInfo`. Also `Repository.requests` is defined as an inline aggregation. The multiplicities are expressed as "`+`" for *one to many*, "`*`" for *zero to many*, "`?`" for *zero*

to one, and "&" for one to one. When declaring features, some qualifiers can be specified to denote that the feature is a key ("+"), optional ("?") or restricted to unique values ("!").

In addition to the types described above, the language includes: (i) Primitive types that can be simple scalar types such as `String` and `Integer`, or specific types that will be system-dependent once a schema is translated to a specific database, such as `Identifier` or `Timestamp`. For example, `Repository.id` is of type `Identifier`, which in MongoDB would be translated to an *ObjectId*, while in Cassandra it would be translated to an `uuid`. (ii) Collection types that can be `Map`, `List`, `Tuple` and `Set`, and (iii) A *union* type of scalar types (`Option` types), e.g., `Developer.2.suspended_acc` in the running example can be of type `String` or `Boolean`.

When declaring features of some scalar types, *ranges* and *regular expressions* can be used to restrict the possible values (Requirement 5). In Figure 5.3, `Developer.email` and `DeveloperInfo.name` are restricted by a regular expression, in `Developer.permissions` and `Repository.requests.status` where only certain values are allowed, and integer values need to be on a certain range in `Popularity_features.num_forks` and `Popularity_features.num_stars`.

5.2.3 U-Schema Mappings to Athena Elements

As by Requirement 1, Athena must be *complete*, and this is achieved by assuring that each U-Schema element can be mapped to Athena.

U-Schema schema types (`entity` and `relationship` types) are directly mapped to Athena equivalents. `Structural Variations` can also be defined in Athena, as `Developer` was defined in Figure 5.3. Although in U-Schema variations hold sets of `features`, in Athena features common to all variations are directly declared in the schema type.

`Attributes` and `aggregates` are also mapped directly to Athena equivalents, but `Keys` and `References` are defined slightly differently. First, a key is directly created when an attribute, or attributes, are promoted by using the "`+`" modifier, as seen in the `Developer.id` field of the running example. References in Athena can be defined with a named feature and the keyword `Ref`, and composed references of more than one attribute can be defined by using the following syntax: `(name1, name2): <EntityName>`.

Finally, the U-Schema `Type` hierarchy has been mapped to Athena elements and then this hierarchy has been extended by adding restrictions and the `Option` type, as described above.

5.3 Athena Schema Transformations

Once the Athena language is defined, we implemented the Athena engine as shown in Figure 5.5. The Athena engine consists of two sets of elements: model-to-model (*m2m*) transformations and model-to-text (*m2t*) transformations. Each one of these sets has a number of implemented transformations to other models (*m2m*) or to generate code for specific databases (*m2t*).

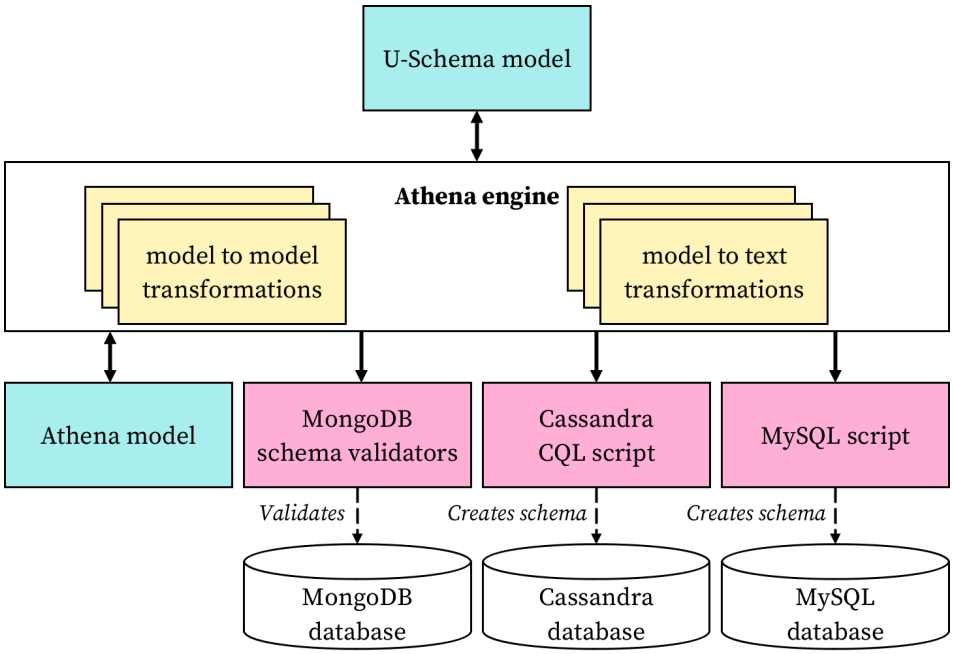

Figure 5.5: The Athena engine developed to handle *m2m* and *m2t* transformations.

In this section we will briefly discuss several available transformations involving Athena schemas and which artifacts are generated from them.

5.3.1 Model to Model Transformations

Since Athena provides several syntactic mechanisms to ease the definition of schemas, we have defined a *normalization* process that receives an Athena schema and transforms it, resulting in an equivalent independent schema more readable and easier to process, visualize and modify. This *normalization* process is always applied before any other transformation in a transparent way to ease that process.

The changes performed are listed below: (i) `import` statements and parent relationships are solved, copying imported and inherited features into schema types as needed. (ii) *Structure* operations are solved: *unions*, *intersections* and *differences* are processed and new structures replace these expressions. (iii) Inner structures and arrays of them are transformed into entity types and aggregations. Finally, (iv) references are filled with their corresponding types if no type was specified by searching for the type of the key feature of the referenced entity type.

Figure 5.6 shows the running example after applying the *normalization* process. Here, the schema version has been incremented, the `Ticket` structure previously provided as a SQL statement has been translated to a *union entity type* and features defined in feature sets have been added to the structure. `Repository` has also been modified to add the features defined in `Popularity_features` and `Timestamp_features`. Finally, the `Repository.requests` field has been replaced by an aggregation of a new entity type named `Requests`.

```
Schema SoftwareDev:2

FSet Popularity_features {
 num_forks:          Integer ( 0 .. 1000 ),
 num_stars:          Integer ( 0 .. 1000 )
}

FSet Timestamp_features {
 created_time:       Timestamp,
 last_activity_date: Timestamp
}

Root entity Developer {
 common {
  + id:              Identifier,
  dev_info:          Aggr<DeveloperInfo>&,
  ! email:           String /^.+@.+\\.com$/,
  permissions:       String in
              ("R","W","X","RW","RX","WX","RWX")
 }
 Variation 1
  { is_active:       Boolean }
 Variation 2
  { suspended_acc:   Option<String, Boolean> }
}

Entity DeveloperInfo
{
 ? about_me:         String
 name:               String /^[A-Z][a-z]*$/,
 team:               String
}
```

```
Root entity Ticket {
 + id:               String,
 developer_id:       Ref<Developer as Identifier>&,
 message:            String,
 repository_id:      Ref<Repository as Identifier>&,
 created_time:       Timestamp,
 last_activity_date: Timestamp,
 num_forks:          Integer ( 0 .. 1000 ),
 num_stars:          Integer ( 0 .. 1000 )
}

Root entity Repository {
 +id:                Identifier,
 title:              String,
 url:                String,
 requests:           Aggr<Requests.1>+,
 developers:         Ref<Developer as Identifier>+,
 ?tags:              List<String>,
 created_time:       Timestamp,
 last_activity_date: Timestamp,
 num_forks:          Integer ( 0 .. 1000 ),
 num_stars:          Integer ( 0 .. 1000 )
}

Entity Requests {
 Common {
  branch:            String,
  status:            String in ( "Open", "Closed" ),
  numLabels:         Number
 }
 Variation 1
}
```

Figure 5.6: The *SoftwareDev* schema after applying the *normalization* process.

In Section 5.2.3 we described a simple mapping from U-Schema to Athena. This process

is implemented in a trivial *m2m* transformation. Another transformation has been implemented to obtain U-Schema models from Athena schemas. By having available both transformations, U-Schema models may be provided with a textual notation and Athena schemas can be used as an input to U-Schema tooling.

This transformation however carries a loss of information because some Athena elements cannot be mapped to U-Schema. In some cases, as in the `Option` type, this loss of information can be mitigated by taking the most open type of the ones considered in the `Option`, but in other cases as in restrictions over `PrimitiveTypes` the loss of information is unavoidable. Figure 5.7 shows the running example transformed to U-Schema as a *union schema*.

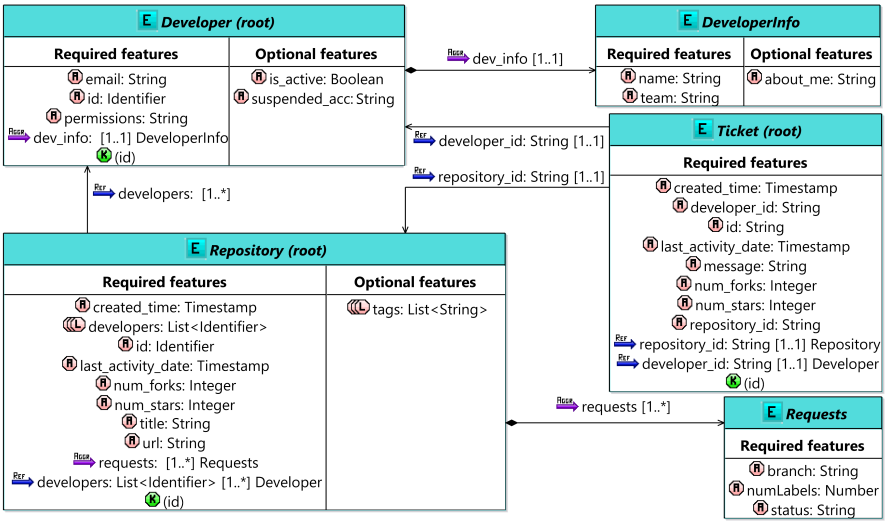

Figure 5.7: The *SoftwareDev* union schema obtained by applying the U-Schema *m2m*.

5.3.2 Model to Text Transformations

The transformations described here could have been implemented directly on the U-Schema metamodel and used after transforming an Athena schema to the corresponding U-Schema model. However, we also intend that the Athena language can be used independently of U-Schema, and therefore we provide it with the ability to generate specific database schemas as follows.

When considering databases with explicit schemas we are able to generate instructions to create the corresponding specific schema from an Athena model as scripts. First, we considered Cassandra because is the most widely used columnar NoSQL store. For this database a

CQL script is generated with instructions to create a schema. Figure 5.8 shows an excerpt of the running example. Here, `Repository` is created by using a `CREATE TABLE` statement since `Repository` is a root entity type, and `Requests` and `Developerinfo` are created by using the `CREATE TYPE` statement since they are aggregated (non-root) entity types and therefore they are mapped as *user-defined types* [CRM22].

```
CREATE TABLE IF NOT EXISTS Repository (
  id                uuid,
  created_time      timestamp,
  developers        list<uuid>,
  last_activity_date timestamp,
  num_forks         int,
  num_stars         int,
  requests          list<frozen<Requests>>,
  tags              list<text>,
  title             text,
  url               text,
  PRIMARY KEY (id)
);
```

```
CREATE TYPE IF NOT EXISTS Developerinfo
(
  about_me  text,
  name      text,
  team      text
);

CREATE TYPE IF NOT EXISTS Requests
(
  branch    text,
  numlabels varint,
  status    text
);
```

Figure 5.8: CQL code for `Repository`, `Requests`, and `DeveloperInfo`.

Another system we considered is MySQL, since Athena schemas can also be translated to a relational model. For this database *SQL* scripts are generated. In Figure 5.9 an excerpt of the running example is shown. Here, after *normalization* some adjustments need to be made, such as collections being translated to additional tables.

```
CREATE TABLE Repository
(
  id                VARCHAR(255) NOT NULL,
  title             VARCHAR(255) NOT NULL,
  url               VARCHAR(255) NOT NULL,
  created_time      TIMESTAMP NOT NULL,
  last_activity_date TIMESTAMP NOT NULL,
  num_forks         INTEGER
      CHECK (num_forks BETWEEN 0 AND 1000)
      NOT NULL,
  num_stars         INTEGER
      CHECK (num_stars BETWEEN 0 AND 1000)
      NOT NULL,
  PRIMARY KEY (id)

);
```

```
CREATE TABLE DeveloperInfo (
  name              VARCHAR(255)
      CHECK (name LIKE '/^[A-Z][a-z]*$/') NOT NULL,
  team              VARCHAR(255) NOT NULL,
  about_me          VARCHAR(255),
  id                INTEGER NOT NULL,
  PRIMARY KEY (id) );

CREATE TABLE Requests (
  requests          VARCHAR(255) NOT NULL,
  branch            VARCHAR(255),
  status            ENUM ('Open', 'Closed'),
  numLabels         NUMERIC,
  id                INTEGER NOT NULL,
  FOREIGN KEY (requests) REFERENCES Repository (id),
  PRIMARY KEY (id));
```

Figure 5.9: SQL code for `Repository`, `Requests`, and `DeveloperInfo`.

Finally, we considered MongoDB as a schemaless database, because MongoDB is the most popular document NoSQL store. Given its schemaless nature, no schema can be gener-

ated for it. Instead, a JSON Schema validation* script can be provided. This validation allows to provide validation on MongoDB by checking if each object to be stored on the database conforms to a provided JSON Schema. In Figure 5.10, a JSON Schema validator for `Repository` is shown. For each `Repository` field an entry is created indicating its type, if it is required or not, and additional validation rules in case it is an array or object. For example, `Repository.requests` is an array of embedded objects that are also validated.

```
SoftwareDev.createCollection("Repository", { validator: {
$jsonSchema: {
 bsonType: "object",
 required: ["_id","title","url","requests","developers",
            "created_time","last_activity_date","num_forks","num_stars"],
 properties: {
  _id:    { bsonType: "objectId", description: "_id must be objectId and IS required." },
  title: { bsonType: "string",   description: "title must be string and IS required." },
  url:    { bsonType: "string",   description: "url must be string and IS required." },
  requests: {
   bsonType: "array", items: {bsonType: "object"}, description: "requests must be array and IS required.",
   properties: {
    branch:    { bsonType: "string", description: "branch must be string." },
    status:    { enum: [ "Open", "Closed" ], description: "status can only be: [Open,Closed]." },
    numLabels: { bsonType: "number", description: "numLabels must be number." }
   }
  },
  developers:   { bsonType: "array", items: { bsonType: "objectId" },
                  description: "developers must be array and IS required." },
  tags:         { bsonType: "array", items: { bsonType: "string" }, description: "tags must be array." },
  created_time: { bsonType: "timestamp", description: "created_time must be timestamp and IS required." },
  last_activity_date: { bsonType: "timestamp",
                        description: "last_activity_date must be timestamp and IS required."},
  num_forks:{bsonType:"int",minimum: 0,maximum: 1000,description:"num_forks must be int and IS required."},
  num_stars:{bsonType:"int",minimum: 0,maximum: 1000,description:"num_stars must be int and IS required."}
}}}});
```

Figure 5.10: Example of a JSON Schema validator generated for `Repository`.

*https://www.mongodb.com/docs/manual/core/schema-validation/.