On Dependencies in Knowledge Graph Construction

Eduard Kamburjan[1,2], Romana Pernisch[3,4], Oscar Corcho[5] and David Chaves-Fraga[6,7]

[1]*IT University of Copenhagen, Denmark*
[2]*University of Oslo, Norway*
[3]*Vrije Universiteit Amsterdam, The Netherlands*
[4]*Discovery Lab, Elsevier, The Netherlands*
[5]*Ontology Engineering Group, Universidad Politécnica de Madrid, Spain*
[6]*Department of Electronics and Computing, Universidade de Santiago de Compostela, Spain*
[7]*CiTIUS, Universidade de Santiago de Compostela, Spain*

Abstract
Knowledge graph construction (KGC) requires numerous assets, such as shapes or mappings, to interact correctly. However, maintenance and assessment of the quality of the pipeline implementing the construction is difficult, as software engineering analyses and quality measures do not address the technologies used in KGC. In this paper, we propose a syntactic, easy to compute notion of dependencies between assets, and show its capability to assess change propagation. Furthermore, we discuss potential to use it for coupling and impact estimation. We evaluate our approach using a prototypical implementation and a case study from the literature, where we find two bugs where missing dependencies indicated an error due to miscommunication during change propagation between the developers of two different assets.

Keywords
Software Analysis, Knowledge Graph Construction, Change Propagation, Impact Analysis

1. Introduction

Knowledge graph construction (KGC) is concerned with the systematic and disciplined development, operation and maintenance of software for the construction of knowledge graphs, and, as such, is a part of software engineering [1]. The field has resulted in methodologies and tools for construction and has reached a stage of maturity where it looks beyond its immediate purpose. Modern KGC requires a way to handle the complexity arising from the ever-increasing number of interacting assets employed in construction: ontologies, shapes, mappings, queries, scripts to run the processes and processing data etc. In contrast to other parts of software engineering, however, tool support is rare and methodologies to guide the developer are mostly implicit or ad hoc.

In this work, we investigate *dependencies*, a central notion of software engineering, from the KGC perspective. Numerous techniques in software analysis rely on dependencies. For example, coupling [2] defines how closely related assets are, and impact analysis to track or predict the effect of changes can be built on dependencies and coupling [3]. Implicit dependencies between assets are a recognized negative quality measure [4]. Dependencies naturally give rise to modules: A module [5] is a set of components that provides encapsulation and only allows dependencies to assets in the interface. In its most general form, a dependency is a relation between two assets A_1 and A_2 that expresses that the functionality of A_2 depends on some functionality provided by A_1[1]. For example, an *input*-dependency between asset A_1 and asset A_2 expresses that A_1 relies on the output of A_2, while an *import*-dependency expresses that A_1 relies on a function, or a data set, within A_2.

KGCW'25: 6th International Workshop on Knowledge Graph Construction, June 1st, 2025, Portorož, SLO
eduard.kamburjan@itu.dk (E. Kamburjan); r.pernisch@vu.nl (R. Pernisch); oscar.corcho@upm.es (O. Corcho); david.chaves@usc.es (D. Chaves-Fraga)
0000-0002-0996-2543 (E. Kamburjan); 0000-0001-8590-1817 (R. Pernisch); 0000-0002-9260-0753 (O. Corcho); 0000-0003-3236-2789 (D. Chaves-Fraga)

[1]This is adapted from the definition of IEEE 24765 [1], *"existence dependency: a constraint between two related entities indicating that no instance of one can exist without being related to an instance of the other."* Our dependencies are existence dependencies on functionality. To the best of our knowledge, there is no agreed-upon definition of *general* dependencies.

```
id,name,location
1,Server1,Frankfurt
2,Server2,Darmstadt
...
```

```
id,first,last,role
1,Peter,Schmitt,Admin
2,Pia,Schwarz,Admin
...
```

```
sysid,userid,dt
1,1,11-12-2024
1,2,11-12-2024
...
```

(a) Example CSV Input

```
OWL
Class: dep:Access          // A1
Class: dep:Role            // A2
Class: dep:System          // A3
Class: dep:User            // A4
ObjectProperty: dep:hasRole
  Domain: dep:User         // A5
  Range: dep:Role          // A6
```

```
OWL
ObjectProperty: dep:accessedBy
  Domain: dep:Access    // A7
  Range: dep:User       // A8

ObjectProperty: dep:accesses
  Domain: dep:Access    // A9
  Range: dep:System     // A10
```

(b) Subset of the ontology, defining OWL classes, domain and range axioms.

Figure 1: Starting point for the example KGC pipeline.

Knowledge graph construction employs numerous different tools and languages, yet no notion of dependency is available. Thus, it is not possible to automatically assess the possible impact of a change, track changes throughout a construction pipeline or reason about maintainability. Certainly, ideas building on dependencies and notions extending them, such as coupling, are used in KGC, but as long as dependencies remain implicit between different kinds of assets, they cannot be connected to quality assessment and analysis of a whole pipeline. For example, the RDF Mapping language (RML) [6] is a domain specific language to describe the transformation of heterogeneous data structures into RDF. Its mappings have explicit dependencies, as they may refer to other mappings and are grouped in files according to their references, i.e., they form modules by coupling. However, there is no dependency relation with further steps in the pipeline.

In this paper, we propose a notion of dependencies in KGC and report on our efforts to link KGC to software engineering best practices and software analysis to (1) enable tool support and increased automation for KGC and (b) explore the idiosyncrasies of KGC tools that distinguish it from other software. In particular, we hope to develop a notion of modularity to increase reusability.

This work is structured as follows. We first illustrate the need for tool support based on explicit dependencies (Section 2), give a first, purely syntactic notion of dependencies (Section 3) and a preliminary evaluation on two KGC projects (Section 4). We discuss its effects, possible applications and limitations (Section 5), as well as related work (Section 6) before we conclude (Section 7).

2. Overview

Motivating Example We illustrate the need for tool support in maintaining KGC pipelines, and applications that build on them, using the following example, pictured in Figure 2. Some system logs and user information are available in CSV files and must be transformed into a KG to be accessed by two different programs. One python program, in the following denoted P_1, is part of the user management and uses a query to show all users and how often they access any system (Figure 4c). Another program, P_2, uses two different queries to find all system accesses made by admins before a certain date and shows all admins (Figures 4a and 4b). The data is transformed as follows. Four RML mappings (in YARRRML [7] syntax, extracted from and based on an ontology, Figure 3a) transform three CSV files (Figure 1a) into RDF, which is subsequently validated using two SHACL shapes (Figure 3b) and accessed by the aforementioned programs and queries.

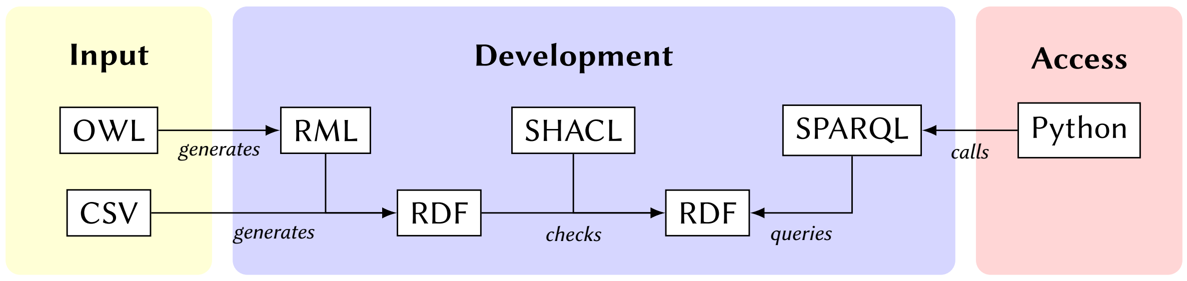

Figure 2: Overview of our example: RML mappings generate RDF from some CSV files. This RDF is subsequently first validated using SHACL, and then accessed from python code using SPARQL queries. The RML itself is also generate, namely from an OWL ontology. The edge between the two RDF nodes is only a virtual transformation.

Challenge A simple impact analysis should answer the following question: *Given a change to a CSV schema, an OWL axiom, or a mapping which other assets (i.e., mappings, shapes, queries, programs) need to be reconsidered?*

It is not easy keeping track of dependencies in the above scenario. For example, a change to a single axiom in the ontology requires considering changes to any mapping that use terms affected by this axiom. Similarly, a change to the mapping requires reconsidering all shapes and queries operating on triples generated by this mapping. However, the shape or query does not refer to the mapping it depends on. The dependency between mapping and axiom is also implicit — even if the mapping is automatically derived from the ontology (e.g., using a tool like OWL2YARRRML[2]): while the dependency is explicit in the generator tool, it is implicit in the mapping itself. Not having a way to discover and define these dependencies creates a complex and manual challenge, compromising the evolution of KG development assets, especially if the assets are managed by different development teams.

3. Dependencies in Knowledge Graph Construction

Intuitively, we require dependencies that express that an asset may not work correctly if another asset is changed. For example, if a class is renamed in an RML mapping, then a SHACL shape operating on the data it generates, i.e., refers to the modified class, may also be modified. As discussed, this dependency from a shape to a mapping is currently implicit.

Before we introduce dependencies formally, we need to distinguish between semantic and syntactic dependencies. A *semantic* dependency is a dependency between two assets A_1, A_2, that precisely characterizes when a change of A_1 to A_1' affects the functionality of A_2. It is defined over the functionality of A_1 and A_2, or the transformations that use them. A *syntactic* dependency is defined over the syntax of A_1 and A_2, it may be an approximation of a semantic dependency[3], or it may be a useful heuristic, which indicates that a change of A_1 may in *some situations* effect the functionality of A_2.

Syntactic dependencies are generally cheap to compute and are sufficient for a first investigation of dependencies in KGC. Semantic dependencies become expensive quickly: For example, to characterize the dependency of a query to an ontology we must characterize whether a change can affect any data consistent with the part of the ontology relevant for the query – this means to employ *ontology extraction* [9, 10] – an expensive operation that has also been shown to be too conservative in its fully expressive form [11, 12].

Syntactic Dependencies. First, we distinguish between external and semantic assets. A semantic asset is operating on the knowledge graph under construction, while an external asset is either input to the construction pipeline, or refers to a semantic asset to execute it.

[2]https://github.com/oeg-upm/owl2yarrrml/

[3]For example, let us consider graph-based deadlock analysis [8]. A syntactic (or abstract) dependency graph of a program is an over-approximation over all semantic (or concrete) dependency graphs that may occur when executing the program. If the abstract dependency graph is cycle-free, then so are all concrete ones occurring during any of its executions.

YARRRML

```yaml
roles:
 sources:
  - access: 'users.csv'
    referenceFormulation: csv
 s: dep:$(role)
 po:
  - [a, dep:Role]
  - [dep:roleName, $(role)]

accesses:
 sources:
  - access: 'accesses.csv'
    referenceFormulation: csv
 po:
  - [a, dep:Access]
  - p: dep:at
    o: value: $(dt)
       datatype: xsd:date
  - p: dep:accesses
    o: value: dep:$(sysId), type: iri
  - p: dep:accessedBy
    o: value: dep:$(userId), type: iri
```

YARRRML

```yaml
persons:
 sources:
  - access: 'users.csv'
    referenceFormulation: csv
 s: dep:$(id)
 po:
  - [a, dep:User]
  - [dep:name, $(first) $(last)]
  - p: dep:hasRole
    o:
     - mapping: roles
       condition: ...

systems:
 sources:
  - access: 'systems.csv'
    referenceFormulation: csv
 s: dep:$(id)
 po:
  - [a, dep:System]
  - [dep:systemName, $(name)]
  - [dep:location, $(location)]
```

(a) YARRRML

```turtle
dep:UserShape
    a sh:NodeShape ;
    sh:targetClass dep:User ;
    sh:property [
        sh:path dep:hasRole ;
        sh:maxCount 1 ;
        sh:minCount 1 ;
    ] .
```

```turtle
dep:AccessShape a sh:NodeShape ;
 sh:targetClass dep:Access ;
 sh:property [
   sh:path dep:accessedBy ;
   sh:maxCount 1 ; sh:minCount 1];
 sh:property [
   sh:path dep:accesses ;
   sh:maxCount 1; sh:minCount 1] .
```

(b) SHACL

Figure 3: Processing assets for the example pipeline.

Definition 1 (External and Semantic Assets)**.** *Ontologies, data files and source code*[4] *are* external assets. *Ontologies consist of axioms, which we also consider external assets. Shapes, mappings and queries are* semantic assets.

Based on the different kinds of assets involved, we differ between external and internal dependencies. The reason for this distinction is that external dependencies are based on explicit reference or modification, not shared vocabulary.

Definition 2 (External Dependencies)**.** *A mapping M depends on a data file D, if D is input to M in the construction. A mapping M depends on an axiom X if M is generated from X. A program P depends on a semantic asset A, if A occurs within P.*

For example, an RML mapping depends on the data files that occur in its `sources-access` clause. If the mapping is realized by a python function, then that function may depend also on those files that are loaded at some other place in the program and then passed to it as a parameter. A program using a query certainly depends on that query, independent of whether the query is syntactically in the program or loaded from an external file. The dependency of mappings to data is straightforward, but

[4]Excluding those implementing mappings manually.

SPARQL
```sparql
SELECT * {
?x a dep:User;
   dep:name ?name;
   dep:hasRole [dep:roleName ?roleN].
FILTER (?roleN = "Admin")
}
```

(a) Query 1

SPARQL
```sparql
SELECT * {
?x dep:accessedBy [ dep:hasRole [  dep:roleName "Admin" ]];
   dep:at ?date;
   dep:accesses [ dep:systemName ?name ].
FILTER (YEAR(?date) < 2024)
}
```

(b) Query 2

SPARQL
```sparql
SELECT ?name ( COUNT(?x) AS ?nr ) {
    ?x dep:accessedBy [ dep:name ?name ]
} GROUP BY ?name
```

(c) Query 3

Figure 4: SPARQL Queries

the dependency of mappings on ontology axioms is more intricate, as it requires keeping track of the *generation* and the subsequent *modification* of a mapping from an axiom, as the original axiom is not necessarily referred to from the template. We deem such tracking realistic.

Internal dependencies are based on explicit references or shared vocabulary but also assume an order in the construction pipeline. Thus, we assume that in the pipeline, there is a driver that executes all assets $(A_i)_{i \in I}$ in some order, and we refer to this partial order by $\preceq$.

Definition 3 (Internal Dependencies)**.** *Let* L *be a set of URIs, which we denote as* library*. A semantic asset* A_1 *depends on another semantic asset* A_2 *if either (1)* A_1 *refers to* A_2 *explicitly, or (2a)* $A_1 \preceq A_2$*, and (2b) there is some URI from* L *that occurs in both* A_1 *and* A_2*.*

The reason why dependencies are defined relative to a library L is that we want to omit trivial dependencies due to common vocabulary such as rdf:type. The above definition uses a white-list approach by making all allowed URIs explicit, an alternative could be a black-list approach that only records the URIs that do not induce a dependency. Note that if the semantic assets in question are serialized in RDF, then dependencies can be retrieved using a SPARQL query [13].

Case (1) in definition 3 is straightforward and explicit. The core of our dependencies is case (2). It is based on the observation that there is no explicit input-output relation between the semantic assets. A whole graph is produced and then processed—in a strict sense, it is not true that the output of one mapping is the input to one shape. However, we can approximate the parts of the graph that are relevant to the functionality of a semantic asset by approximating the part of the graph that it will, in practice, affect. To do so, we approximate the signature by collecting all URIs (which are also in the library). If the signatures of the two semantic assets overlap, then they may affect the same part of the graph.

For example, the shape `dep:UserShape` (Figure 3b) depends on the mapping `persons` (Figure 3a) because they both contain the URLs `dep:User` and `dep:hasRole`. Similarly, query 2 (Figure 4b) depends on both `dep:UserShape` and `persons` because it contains the URL `dep:hasRole`. All triples with these

Python
```python
def make_role(onto : Ontology):
    roles = set(df_names['role'])
    roles.remove(np.nan)
    with onto:
        for role in roles:
            new_role = onto.Role()
            new_role.roleName = role
    return onto
```

Figure 5: Python mapping

elements are created by `persons` and validated by `dep:UserShape`, thus the results of the query depend on these assets.

The order excludes dependencies between assets that do not operate consecutively. For example, both the query in fig. 4a and the query in fig. 4b contain the URI for dep:hasRole, but there is no dependency, because they operate on the same input RDF, and not on each others results.

URIs may occur either directly, or indirectly in an asset. In the RML examples, URIs occur directly. However, consider the python function in Figure 5, that performs the equivalent of the `roles` mapping using owlready2 [14]. It contains the URIs for dep:Role (implicitly in `onto.Role`) and dep:roleName (implicitly in `new_role.roleName`). Analysis of assets written in general purpose programming languages is out of scope for this work, as we focus on assets written in specialized languages, but we conjecture that our notion of internal dependency naturally fits into dependencies in programming.

Figure 6 shows the dependency graph of our example (w.r.t. to the library of all URIs prefixed with `dep:`). It enables us to estimate change propagations: If the ontology is refactored and `dep:Role` is moved to a new URL or is renamed, then we can now explicitly see that program P_1 that contains a query that may need to be refactored the next time the KGC pipeline is run.

The dependency graph shows some information that is not visible otherwise without reading the code of the assets and that fits our intuition: A change in the systems and access log data does not require to change the HR application, as there is no dependency between `systems.csv`, `accesses.csv` and P_1. It also shows that the mapping of persons is tightly coupled (i.e., has many incoming dependencies), which fits the intuition that users are relevant for both applications.

Example 1. *Let us examine the dependencies of `access.rml` as pictured in Figure 6. The dependency on `access.csv` is an external dependency, as `access.csv` is explicitly referred to from the mapping. The dependencies on `A1`, `A7` and `A9` are external, but implicitly – the mapping is generated based on these axioms. The other dependencies are internal due to common URIs.*

- *Assets `access.rml` and `access.shacl` share dep:access, dep:accessedBy and dep:accesses.*
- *Assets `access.rml` and `q2.sparql` share dep:accessedBy and dep:accesses.*
- *Assets `access.rml` and `q3.sparql` share dep:accessedBy.*

Semantic Dependencies. The derived dependency graph also illustrates the limitation of the syntactic analysis: Only q_2 has a dependency on `system.rml`. However, q_3 relies on the OWL concept `System` as well, as it is in the range of `accessedBy`. Thus, the results of q_3 may be different if the mapping for the system is changed such that this concept is used differently. Due to the lack of reasoning when computing dependencies, this is not detected. A further case is the following: suppose the ontology providing context contains the following axiom.

OWL
```
dep:Admin EquivalentTo: dep:hasRole some dep:roleName value "Admin"
```

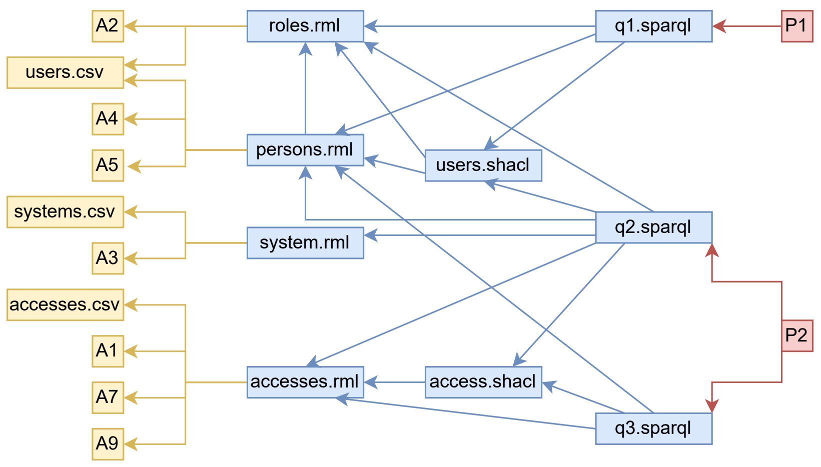

Figure 6: Dependency Graph of the Example

Now, the following query is equivalent to q_2 using a suitable entailment regime, but loses two dependencies (the one to `roles.rml` and `persons.rml`).

SPARQL
```
SELECT * {
?x dep:accessedBy [ a dep:Admin ];
   dep:at ?date;
   dep:accesses [ dep:systemName ?name ].
FILTER (YEAR(?date) < 2024)
}
```

It may be necessary to extend L to include further URIs with different degrees of reasoning – as discussed, in the worst case, L requires computing the deductive module over the signature L. We conjecture, however, that for applications without reasoning, i.e., for *data* processing, syntactic dependencies are sufficiently precise. Making *precision* concrete requires defining semantic dependencies, which we leave to future work.

4. Evaluation

Implementation A preliminary implementation is available online.[5] The tool takes a set of files that either contain CSV, RML mappings, singular SHACL shapes, SPARQL shapes or python code. The order is fixed: CSV $\preceq$ RML $\preceq$ SHACL $\preceq$ SPARQL $\preceq$ Python.

For each semantic asset A, the tool extracts the contained URIs and then removes those not in the provided library L. The result $\mathrm{sig}(A)$ is then used in the next step. A dependency between two assets A_1 and A_2 is created if $\mathrm{sig}(A_1) \cap \mathrm{sig}(A_2) \neq \emptyset$ and $A_1 \preceq A_2$.

The aim of our validation is to find out whether syntactic dependencies (1) make structures explicit that are implicitly known before, and (2) can be used to find mistakes.

Case Study 1. Our first case study is a knowledge graph construction pipeline for a teaching ontology [15] which follows the structure described above: CSV is translated into RDF using RML mappings, then validated using SHACL and accessed using SPARQL. The SHACL shapes are generated from the RML mappings and from the underlying OWL ontology using SCOOP [16].

The pipeline has 3 CSV files, 11 `rml:TriplesMap`, 19 SHACL sh:NodeShape and 8 SPARQL queries. Analyzing the pipeline takes 2s on a standard office laptop and results in 269 dependencies. For the

[5]https://github.com/Edkamb/ConstructionDependencies

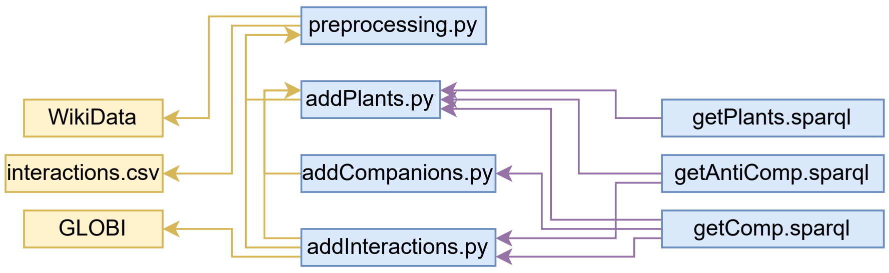

Figure 7: Dependency Graph of CoPla

library L_{teach} (cf. definition 3) we use all URIs outside standard namespaces such as `rdf`. As expected, we can recover some structures.

- The shapes generated from a mapping all depend on the mapping. As 3 of the 11 `rml:TriplesMap` are for one concept and result in 8 `sh:NodeShape`, these 11 assets form a tight cluster.
- Some concepts (in our case `schema:name`) with a very general domain are used in multiple mappings, which makes it hard to interpret the dependencies. This can be counteracted by excluding them from the library, or possibly by outputting all terms that can justify a dependency.

During manual analysis of the dependencies, we were indeed able to detect two anomalies:

- One query (`q8_no_data.rq` in the auxiliary material) has no dependencies. The reason is that it queries for triples using only one URI from the library L_{teach}, which is not generated by any mapping. The corresponding line was commented out in the RML, because no data was available for it. As the query always returns an empty set, any application using it has a vacuous feature.
- One shape similarly had no dependencies, because it used a different prefix for (`coursesonto:Lecturer` vs. a local URI from the developer). Here, it seems a change in the prefixes was not correctly propagated between two developers.

Both anomalies are exactly corresponding to dependency mismanagement in general software engineering: A change was not propagated to another party or tool, which in turn had no automatic way to detect the change, as it was implicit what assets it needs to monitor in the first place.

Case Study 2. Our second case study is the Companion Planting Decision Support system [17]. The system is a prototype which combines multiple information sources and builds a complex ontology to make use of reasoning capabilities. In this case study, the original sources were preprocessed (`preprocessing.py`) into 2 CSV files that are mapped to OWL axioms using 3 Python functions. One of these functions (`addInteractions.py`) calls the SPARQL endpoint of GLOGI to retrieve triples, which are directly integrated in the ontology. The prototype is then using the OWL API as a Java backend, from which we extracted 3 SPARQL queries for this case study. Further queries are part of the prototype but could not be transformed into SPARQL as they make use of complex reasoning capabilities, by first constructing an Abox from the user input, reasoning, explaining or fixing axioms before returning explanations rather than triples.

The dependency graph is shown in Figure 7 which we constructed manually. On the on hand, this highlights the issue from software engineering of tracking dependencies within python functions as a still unsolved problem. On the other hand, the resulting graph is not big. It shows that for small pipelines the dependency graph is small enough to be investigated visually. For larger projects with more python scripts involved a manual investigation of dependencies is not feasible, hence showcasing the need for further research.

5. Discussion

Best Practices In [18], the authors highlight the need to integrate software engineering methods and best practices into knowledge graph development, given the complexity, dependencies, and large number of assets involved in the process. Modularity has been key to software reusability but despite the declarative nature of many assets used in KG construction, their reusability remains minimal. To the best of our knowledge, the concepts of modularity and dependencies in this context have not yet been fully addressed. Exploiting them in KG construction will not only improve the quality of the generated graphs but also enhance the overall construction process. Additionally, it will reduce manual effort in developing mapping rules, SHACL shapes, and other components while preventing errors and facilitating their resolution.

Further Structural Notions Modules are a central notion for composition and reuse, which in turn are major quality measures both for software [19, 20] and data products [21]. In ontology engineering, the notion of modularity is highly volatile, and several kinds of definitions corresponding to different kinds of reuse are used in practice [22].

KGC requires a notion of modularity on the level of all its assets, not merely ontology modules, and dependencies can be used to compute *coupling* metrics [23], i.e., measures for how connected a set of assets are. A potential avenue for further research is to investigate which of the ontology module notions and underlying metrics [24] are good indicators for modules of the software and pipelines operating on them.

Based on modules, further notions can be built. For example, reuse can be enabled through variability on a module level [25], where parts of the systems are exchanged depending on the features required by the overall application.

Impact Analysis Analysis of the impact of ontology changes is not a novel topic. So far, it has been addressed from the perspective of impact on downstream applications, where the two (ontology and application) are looked at in a decoupled way. For example, in the analysis by Gross et al. [26] the application of functional enrichment analysis over the Gene Ontology is assessed. The authors define stability metrics and examine the evolution of the Gene Ontology from a macroscopic perspective. A similar approach is taken by Pernisch et al. [27], where the authors analyse the impact of evolution on the inference calculation of the same ontology and capture the impact on a large scale. Such studies do not consider individual change and its direct consequences as such but rather deal with the notion of ontology evolution.

There are few other studies that take a more detailed approach, such as Gottron and Gottron [28] or Osborne et al. [29]. In these studies, the authors provide an analysis of changes and their direct or indirect impact on the downstream application (indexing, predictions). The problem is looked in a feed-forward way, where the change is applied, the application executed and the impact analysed. The authors do not take an explicit dependency analysis approach to the problem and this remains an open research problem.

All these studies lack a detailed look at the kind of ontology changes and their associated direct and explicit impacts. There is no notion of complexity of the change and what the complexity means for the downstream application. The closest work which assesses the type of change and its direct consequences per type of change and how to deal with that is the work by Conde-Herreros et al. [13]. The authors develop a tool for propagating changes to the ontology to RML mappings for a KG. However, there is no associated analysis with this tool as they focus on the technical challenge associated with dealing with changes rather than analysis of complexity and dependency.

6. Related Work

Knowledge Graph Construction. There are several non-technical, high-level proposals that explore the modularization of the knowledge graph construction process. These range from KG lifecycle approaches [30, 31], where KG construction is one of the tasks, to more methodological perspectives [32, 33, 34]. In general, these proposals are more focused on the tasks performed by knowledge engineers rather than on technical and low-level challenges (i.e. dependencies). In [35], the authors propose a workflow for creating mapping rules, where they structure the process around ontology classes to facilitate development. However, dependencies between mapping documents and classes remain implicit. As previously mentioned, the closest related work is presented in [13], where ontology changes are propagated over RML-defined mapping rules [6] using a fully declarative approach. However, this method requires changes to be defined as an external resource (a KG of changes, indeed) for propagation, and dependencies are not explicitly calculated.

Ontology Engineering Tools to describe workflows to construct ontologies, such as the recent Ontology Development Kit [36] and underlying ontology pipeline tools, such as ROBOT [37], focus on editing and maintaining ontologies, i.e., on composing and editing sets of OWL axioms. While they related different kinds of assets (e.g., exporting ontologies from spreadsheets), they are concerned with a restricted form of knowledge graphs and enforce the whole workflow top-down.

Djedidi and Aufaure [38] look at dependencies in ontology evolution itself. They propose an ontology change management framework or pipeline that takes the current ontology into account and the change which needs to be applied. In specific steps, the change is assessed in terms of dependencies within the ontology (and some external artifacts) itself by checking compatibility in terms of consistency of the ontology. This framework also suggests the automatic resolution of inconsistencies by adjusting how the change is to be applied to the ontology. Other works on ontology evolution and ontology changes take a less technical and detailed approach. For example, Zablith et al. [39] survey existing frameworks and techniques for the individual steps in the evolution. Two specific steps to be mentioned here are the *Validating Changes* and *Assessing Impact*. In the Validating Changes step, present in all surveyed frameworks, is used to filter our changes which would introduce incoherence or inconsistency. Even though this step does not provide automatic resolution strategies like [38], it inherently assesses potential dependencies within the ontology. In the Assessing Impact step, the focus shifts towards external artifacts that are dependent on the ontology, so directly related to the problem at hand. However, most survey framework do not concern themselves with this step and it is only present in, what the authors refer to as KOAN [40] and Protégé [41]. An example of this assessment is the checking of the ability to answer a specific query. Unfortunately, all the mentioned works above are more or almost 20 years old and do not easily translate to today's KG construction pipelines and also miss the theoretical discussion of dependencies which we could directly adapt here.

Related Fields Other fields have similar challenges when it comes to software engineering for data-driven pipelines with heterogeneous assets. Idowu et al. [42] discuss asset management in machine learning projects, where numerous software, data and other assets need to interact to produce value from a set of source data sets. In machine learning, a larger field compared to KGC, the challenging management and the need for deeper connections with software engineering practices have been recognized earlier [43].

Infrastructure-as-Code [44] describes the assets (mostly scripts and configuration files) that configure and deploy modern software systems in the cloud, a practice arising from DevOps. Due to the similarity of the assets to standard software and their critical role concerning security, reliability and resources, static analysis and other software quality measures have been investigated for it [45, 46].

OpenCAESAR [47, 48] is a development environment for ontology development using the Ontology Modelling Language (OML), which uses Gradle tasks to construct RDF from OML, and run constraints expressed in SHACL or SPARQL on the resulting data. Gradle enables to manage dependencies between these tasks, but not between the underlying assets.

7. Conclusion

In this paper we introduce and analyze the idea of KG construction dependencies. To the best of our knowledge, this is the first work that addresses explicit handling of the interrelationships between the assets involved in KG development. We have demonstrated that incorporating dependency identification will not only help create higher-quality knowledge graphs but also enable better management and utilization of the resources used for this purpose such as mapping rules, pre-processing scripts, SPARQL queries or SHACL shapes. Thanks to dependency identification, knowledge engineers can work more efficiently, leading to a broader and more effective adoption of semantic web technologies.

Future Work. To make dependencies practical, the semantics of dependencies must be fixed for each considered language used in KGC, before a stable tool that can analyse a full pipeline can be developed. In particular, assets outside the construction itself, for example competency questions [49, 50] that may change during the overall development, should be considered to enable traceability of changes back to requirements. Conceptually, the connection to change coupling [51] remains an open question.

Acknowledgments. David Chaves-Fraga is funded by the Agencia Estatal de Investigación (Spain) (PID2023-149549NB-I00), the Xunta de Galicia - Conselleria de Educación, Ciencia, Universidades e Formación (Centro de investigación de Galicia accreditation 2024-2027 ED431G-2023/04 and Reference Competitive Group accreditation 2022-2026, ED431C 2022/19) and the European Union (European Regional Development Fund - ERDF). Eduard Kamburjan is partially supported by the EU project SM4RTENANCE (101123490). We thank the organizers of the Dagstuhl Seminar 24061 *"Are Knowledge Graphs Ready for the Real World?"*, where the ideas presented here were discussed for the first time.

References

[1] ISO/IEC/IEEE 24765-2010(E), Systems and software engineering — Vocabulary, Standard, International Organization for Standardization, 2010.

[2] A. J. Offutt, M. J. Harrold, P. Kolte, A software metric system for module coupling, J. Syst. Softw. 20 (1993) 295–308.

[3] M. Gethers, B. Dit, H. H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: ICSE, IEEE Computer Society, 2012, pp. 430–440.

[4] M. Cataldo, A. Mockus, J. A. Roberts, J. D. Herbsleb, Software dependencies, work dependencies, and their impact on failures, IEEE Trans. Software Eng. 35 (2009) 864–878.

[5] N. Wirth, The module: A system structuring facility in high-level programming languages, in: Language Design and Programming Methodology, volume 79 of *Lecture Notes in Computer Science*, Springer, 1979, pp. 1–24.

[6] A. Iglesias-Molina, D. V. Assche, J. Arenas-Guerrero, B. D. Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga, A. Dimou, The RML ontology: A community-driven modular redesign after a decade of experience in mapping heterogeneous data to RDF, in: ISWC, volume 14266 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 152–175.

[7] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative Rules for Linked Data Generation at your Fingertips!, in: Proceedings of the 15$^{\text{th}}$ ESWC: Posters and Demos, 2018.

[8] A. Flores-Montoya, E. Albert, S. Genaim, May-happen-in-parallel based deadlock analysis for concurrent objects, in: FMOODS/FORTE, volume 7892 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 273–288.

[9] U. Sattler, T. Schneider, M. Zakharyaschev, Which kind of module should I extract?, in: Description Logics, volume 477 of *CEUR*, 2009.

[10] B. Konev, C. Lutz, D. Walther, F. Wolter, Model-theoretic inseparability and modularity of description logic ontologies, Artificial Intelligence 203 (2013) 66–103.

[11] J. Chen, M. Ludwig, Y. Ma, D. Walther, Zooming in on ontologies: Minimal modules and best excerpts, in: ISWC, volume 10587 of *LNCS*, 2017.

[12] P. Koopmann, J. Chen, Deductive module extraction for expressive description logics, in: IJCAI, ijcai.org, 2020.

[13] D. Conde-Herreros, M. Poveda-Villalón, R. Pernisch, L. Stork, O. Corcho, D. Chaves-Fraga, Propagating Ontology Changes to Declarative Mappings in Construction of Knowledge Graphs, in: Fifth International Workshop on Knowledge Graph Construction @ ESWC, 2024.

[14] J. Lamy, Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies, Artif. Intell. Medicine 80 (2017) 11–28. URL: https://doi.org/10.1016/j.artmed.2017.07.002. doi:10.1016/J.ARTMED.2017.07.002.

[15] E. Ilkou, H. Abu-Rasheed, D. Chaves-Fraga, E. Engelbrecht, E. Jiménez-Ruiz, J. E. Labra-Gayo, Teaching knowledge graph for knowledge graphs education, Semantic Web (Under Review) (2025).

[16] X. Duan, D. Chaves-Fraga, O. Derom, A. Dimou, Scoop all the constraints' flavours for your knowledge graph, in: The Semantic Web: 21st International Conference, ESWC 2024, Springer, 2024, p. 217–234. doi:10.1007/978-3-031-60635-9_13.

[17] G. Zamprogno, M. Adamik, R. Roothaert, A. Naghdipour, L. Stork, P. Koopmann, R. Pernisch, B. Kruit, J. Chen, I. Tiddi, S. Schlobach, Supporting companion planting with the copla ontology, in: KG4S@ESWC, volume 3753 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 29–41.

[18] D. Chaves-Fraga, O. Corcho, A. Dimou, M.-E. Vidal, A. Iglesias-Molina, D. Van Assche, Are knowledge graphs ready for the real world? challenges and perspective (dagstuhl seminar 24061), Dagstuhl Reports 14 (2024) 1–70.

[19] W. B. Frakes, C. Terry, Software reuse: Metrics and models, ACM Comput. Surv. 28 (1996) 415–435.

[20] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, A. D. Lucia, An empirical study on the developers' perception of software coupling, in: ICSE, IEEE Computer Society, 2013, pp. 692–701.

[21] D. Arribas-Bel, M. Green, F. Rowe, A. Singleton, Open data products-a framework for creating valuable analysis ready data, J. Geogr. Syst. 23 (2021) 497–514.

[22] S. Borgo, Goals of modularity: A voice from the foundational viewpoint, in: WoMO, volume 230 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2011, pp. 1–6.

[23] S. Oh, H. Y. Yeom, J. Ahn, Cohesion and coupling metrics for ontology modules, Inf. Technol. Manag. 12 (2011) 81–96.

[24] Z. C. Khan, C. M. Keet, Dependencies between modularity metrics towards improved modules, in: EKAW, volume 10024 of *Lecture Notes in Computer Science*, 2016, pp. 400–415.

[25] F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt, L. Paolini, Variability modules, J. Syst. Softw. 195 (2023) 111510.

[26] A. Gross, M. Hartung, K. Prüfer, J. Kelso, E. Rahm, Impact of ontology evolution on functional analyses, Bioinformatics 28 (2012) 2671–2677.

[27] R. Pernisch, D. Dell'Aglio, A. Bernstein, Beware of the hierarchy — An analysis of ontology evolution and the materialisation impact for biomedical ontologies, Journal of Web Semantics 70 (2021) 100658. URL: https://www.sciencedirect.com/science/article/pii/S1570826821000330. doi:10.1016/j.websem.2021.100658.

[28] T. Gottron, C. Gottron, Perplexity of Index Models over Evolving Linked Data, in: ESWC, volume 8465, Springer, 2014, pp. 161–175.

[29] F. Osborne, E. Motta, Pragmatic Ontology Evolution: Reconciling User Requirements and Application Performance, in: ISWC, volume 11136 of *LNCS*, Springer, 2018, pp. 495–512. Tex.ids= osborne_pragmatic_2018.

[30] A. Cimmino, R. García-Castro, Helio: a framework for implementing the life cycle of knowledge graphs, Semantic Web 15 (2024) 223–249.

[31] U. Simsek, K. Angele, E. Kärle, J. Opdenplatz, D. Sommer, J. Umbrich, D. Fensel, Knowledge graph lifecycle: Building and maintaining knowledge graphs., in: KGCW@ ESWC, 2021.

[32] G. Tamašauskaitė, P. Groth, Defining a knowledge graph development process through a systematic review, ACM Transactions on Software Engineering and Methodology 32 (2023) 1–40.

[33] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, A. Wahler,

D. Fensel, et al., How to build a knowledge graph, Knowledge Graphs: Methodology, Tools and Selected Use Cases (2020) 11–68.

[34] R. Pernisch, M. Poveda-Villalón, D. Conde-Herreros, D. Chaves-Fraga, L. Stork, When ontologies met knowledge graphs: Tale of a methodology, in: European Semantic Web Conference, Springer, 2024, pp. 286–290.

[35] D. Chaves-Fraga, O. Corcho, F. Yedro, R. Moreno, J. Olías, A. De La Azuela, Systematic construction of knowledge graphs for research-performing organizations, Information 13 (2022) 562.

[36] N. Matentzoglu, D. Goutte-Gattat, S. Z. K. Tan, J. P. Balhoff, S. Carbon, A. R. Caron, W. D. Duncan, J. E. Flack, M. Haendel, N. L. Harris, W. R. Hogan, C. T. Hoyt, R. C. Jackson, H. Kim, H. Kir, M. Larralde, J. A. McMurry, J. A. Overton, B. Peters, C. Pilgrim, R. Stefancsik, S. M. Robb, S. Toro, N. A. Vasilevsky, R. Walls, C. J. Mungall, D. Osumi-Sutherland, Ontology development kit: a toolkit for building, maintaining and standardizing biomedical ontologies, Database J. Biol. Databases Curation (2022).

[37] R. C. Jackson, J. P. Balhoff, E. Douglass, N. L. Harris, C. J. Mungall, J. A. Overton, ROBOT: A tool for automating ontology workflows, BMC Bioinform. 20 (2019) 407:1–407:10.

[38] R. Djedidi, M.-A. Aufaure, Ontology Change Management, in: A. Paschke, H. Weigand, W. Behrendt, K. Tochtermann, T. Pellegrini (Eds.), 5th International Conference on Semantic Systems, Graz, Austria, September 2-4, 2009. Proceedings, Verlag der Technischen Universität Graz, 2009, pp. 611–621. URL: http://www.i-semantics.at/2009/papers/ontology_change_management.pdf.

[39] F. Zablith, G. Antoniou, M. d'Aquin, G. Flouris, H. Kondylakis, E. Motta, D. Plexousakis, M. Sabou, Ontology evolution: a process-centric survey, The Knowledge Engineering Review 30 (2015) 45–75. doi:10.1017/S0269888913000349, publisher: Cambridge University Press.

[40] L. Stojanovic, Methods and tools for ontology evolution, PhD Thesis, Karlsruhe Institute of Technology, Germany, 2004.

[41] N. F. Noy, A. Chugh, W. Liu, M. A. Musen, A Framework for Ontology Evolution in Collaborative Environments, in: I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, L. M. Aroyo (Eds.), The Semantic Web - ISWC 2006, Springer, Berlin, Heidelberg, 2006, pp. 544–558. doi:10.1007/11926078_39.

[42] S. Idowu, D. Strüber, T. Berger, Asset management in machine learning: State-of-research and state-of-practice, ACM Comput. Surv. 55 (2023) 144:1–144:35.

[43] S. Amershi, A. Begel, C. Bird, R. DeLine, H. C. Gall, E. Kamar, N. Nagappan, B. Nushi, T. Zimmermann, Software engineering for machine learning: a case study, in: ICSE (SEIP), IEEE / ACM, 2019, pp. 291–300.

[44] K. Morris, Infrastructure as Code: Managing Servers in the Cloud, 1st ed., O'Reilly Media, Inc., 2016.

[45] A. Rahman, R. Mahdavi-Hezaveh, L. A. Williams, A systematic mapping study of infrastructure as code research, Inf. Softw. Technol. 108 (2019) 65–77.

[46] M. Chiari, M. D. Pascalis, M. Pradella, Static analysis of infrastructure as code: a survey, in: ICSA Companion, IEEE, 2022, pp. 218–225.

[47] M. Elaasar, N. Rouquette, D. Wagner, B. J. Oakes, A. Hamou-Lhadj, M. Hamdaqa, opencaesar: Balancing agility and rigor in model-based systems engineering, in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE Press, 2023, p. 221–230. doi:10.1109/MODELS-C59198.2023.00051.

[48] D. A. Wagner, M. Chodas, M. Elaasar, J. S. Jenkins, N. Rouquette, Ontological Metamodeling and Analysis Using openCAESAR, Springer International Publishing, Cham, 2023, pp. 925–954. doi:10.1007/978-3-030-93582-5_78.

[49] C. M. Keet, Z. C. Khan, On the roles of competency questions in ontology engineering, in: EKAW, volume 15370 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 123–132.

[50] G. K. da Silva Quirino, J. S. Salamon, M. P. Barcellos, Use of competency questions in ontology engineering: A survey, in: ER, volume 14320 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 45–64.

[51] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software

defects, in: WCRE, IEEE Computer Society, 2009, pp. 135–144.