Test-driven knowledge graph construction

Jindřich Mynarz Kateřina Haniková Vojtěch Svátek

2022

Abstract

We propose how to construct knowledge graphs using a method based on user requirements formulated as competency questions. It suggests to formalize the competency questions as SPARQL queries wrapped as SHACL constraints allowing them to be used as automated tests. The method defines a process to guide knowledge graph construction based on feedback from tests. It aims to reduce the engineering effort required to construct a knowledge graph that meets the requirements, while assessing the quality of semantic artifacts produced in this effort. We intend the method to provide a solid engineering basis for knowledge graph construction using the lessons learnt from software development and based on open standards. We demonstrate using the method on construction of a knowledge graph about antigen covid tests.

Knowledge graphs use graph-based data models to capture knowledge commonly combined from large and diverse data sources (Hogan et al. 2021). Constructing knowledge graphs can be an intricate and open-ended task that is in many ways an art rather than a craft.[1] There is a shortage of solid engineering practices guiding through this task.

We propose how to construct knowledge graphs using a standards-based method founded upon user requirements formulated as competency questions (CQs). It suggests to formalize the CQs as SPARQL queries (Harris and Seaborne 2013) and wrap them as SHACL constraints (Holger Knublauch and Kontokostas 2017) in order to allow them to be used as automated tests. The method defines a process guiding knowledge graph construction based on feedback from tests. It aims to reduce the effort required to construct a knowledge graph that meets its requirements and passes quality assessment of semantic artifacts, such as ontologies, that are made in the process.

[1] The same argument was made about designing ontologies in Soldatova, Panov, and Džeroski (2016).

Motivation

Following a method for knowledge graph construction offers several benefits. In general, a method helps decide what to do next. In particular, it is useful in the beginning in helping to overcome the blank canvas paralysis. Moreover, a method breaks down the effort required for knowledge graph construction into sub-tasks, which can help organize and coordinate a team working on them.

The direction provided by a method anchored in user requirements can reduce the undirected upfront effort and help avoid overengineering. For example, it discourages needless effort spent on achieving a lossless transformation to the target knowledge graph that preserves all source data even though it might not be needed. It can help avoid premature abstraction and premature optimization, such as for performance or readability. Thanks to the tests checking if the user requirements are satisfied, we get an early proof of value instead of speculating about it.

Related work

When outlining this method, we can learn from ontology engineering, as it has a head start of several decades on knowledge graphs. There is a long tradition of using competency questions as requirements for ontologies (Gruninger and Fox 1994). Much of this experience can be reused, since ontologies serve as essential building blocks of knowledge graphs imbuing them with explicit semantics. Knowledge graphs in turn can be considered as ontologies populated with data. Creating tests for ontologies out of CQs is also nothing new Zemmouchi-Ghomari and Ghomari (2013). More recently, it was adopted for knowledge graphs too (Pan et al. 2017).

Broader still, we can adopt the practices established in software engineering. The hereby presented method borrows from the test-driven development cycle (Beck 2003), which is commonly characterized by the following steps:

- Add a test
- Run all tests, expecting the new test to fail
- Write the simplest code that passes the new test
- All tests should pass
- Refactor as needed

A fundamental feature of this cycle is that it intertwines development with testing, so that tests exercise the development artifacts and provide feedback informing further work. Tests provide a controlled way how to evolve the artifacts under development in response to changing requirements, such as when their scope is extended or they are refined iteratively. The field of ontology engineering is already adopting agile development approaches using tests. Ontology-

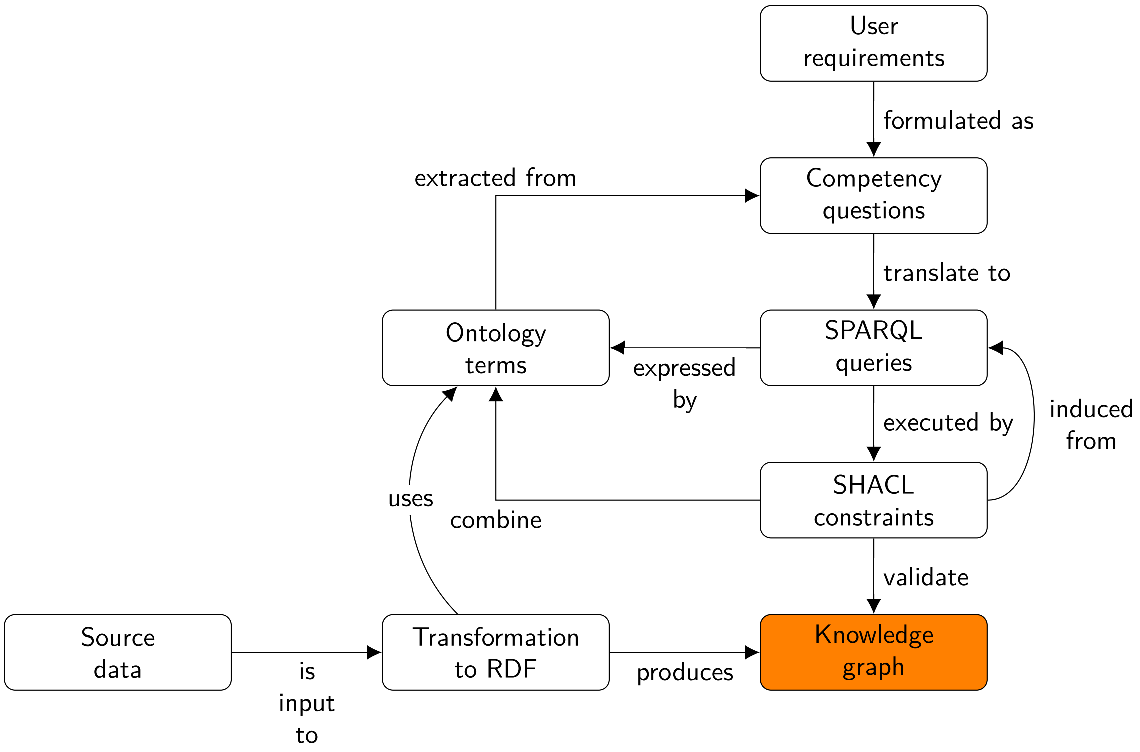

Figure 1: Knowledge graph construction method

specific methods, such as SAMOD (Peroni 2016) or Linked Open Terms (Poveda-Villalón et al. 2022), are being proposed.

The following text describes the proposed method for knowledge graph construction and demonstrates its use on a knowledge graph about antigen covid tests.

Method

We present a method for test-driven knowledge graph construction. The method proposes a sequence of steps and feedback loops between them. Each step produces or consumes one or more artifacts, such as ontologies or SPARQL queries. An overview of the relations between the key artifacts used by the method is depicted in Fig. 1. The method involves the following steps:

1. Start by using knowledge elicitation techniques (Shadbolt and Smart 2015) to gather user requirements. User requirements can be gathered from subject-matter experts or prospective users of the knowledge graph. For-

mulate the user requirements as CQs (Gruninger and Fox 1994). CQs are analytical questions that the target knowledge graph is expected to answer. In order to avoid misinterpretation, they should be reviewed by those who expressed the requirements the CQs capture.

2. Analyze the CQs to extract terms and relations. CQs can be analyzed using the linguistic notion of presupposition, which can be defined as *"a condition that must be met for a linguistic expression to have a denotation"* (Ren et al. 2014). Viewed this way, CQs presuppose the terms with which they can be expressed. The terms to extract from CQs are typically nouns or noun phrases that refer to entities from the domain of the knowledge graph under construction. They can be considered lexical surface forms of these entities. The terms can be identified manually or aided by tools, such as part-of-speech tagging. The relations to extract might be represented by verbs connecting the terms co-occurring in a CQ. Pronouns may translate to joins between the relations.

3. Formalize the terms and relations in a minimum viable ontology using RDF Schema (Brickley and Guha 2014). While terms typically map to classes (i.e. instances of `rdfs:Class`), relations map either directly to properties (i.e. instances of `rdf:Property`) or indirectly to n-ary relations represented by classes. Document the ontology with definitions sourced from and validated by subject-matter experts to create a shared understanding. The ontology should make minimal ontological commitment, so avoid its upfront axiomatization, such as OWL restrictions. The resulting ontology shall serve as an explicit and machine-readable conceptualization of the knowledge graph's domain. Formalizing the ontology may in turn reveal ambiguity in the CQs. Ambiguous CQs cannot be formalized reliably since misinterpretations may occur. In such cases, sources of ambiguity should be revised with subject-matter experts aiming to reformulate the CQs in an unambiguous way.

4. Translate the CQs to SPARQL queries expressed by the ontology. SPARQL queries make the CQs executable. Here, we expect a manual translation of the CQs to SPARQL queries. Automated translation was attempted by others, such as in Espinoza-Arias, Garijo, and Corcho (2022). Start with hypothetical, syntactically valid SPARQL queries expressed by the ontology. A CQ may translate to a SPARQL graph pattern composed of triple patterns joining the terms co-occurring in the question via specific relations. The way CQs should be translated to SPARQL queries largely depends on their expected answers. Ren et al. (2014) suggests that we might be *"more interested in checking if a CQ can be meaningfully answered, instead of directly answering a CQ."* When using such existential quantification, a SPARQL query formalizing a CQ can be expected to return non-empty results. What it means is that the query can answer a CQ using a given knowledge graph. It does not verify if the answer is correct. Conversely, some CQs may

describe universal invariants of their domain. Queries encoding these invariants are akin to property-based testing (Fink and Bishop 1997). In case stricter guarantees are needed, the expected correct answers can be included in the queries and validation constraints implementing the CQs, corresponding to the usual example-based testing.

5. Wrap the queries as SPARQL-based SHACL constraints to automate their execution. SHACL defines a target[2] of each constraint. A target defines the scope of a CQ. Scope of the data graph validated by a given constraint may either encompass the entire knowledge graph or cover its subset. For instance, existentially quantified queries checking the complete data graph can target it by using `sh:targetNode [] .`. It is also possible to rewrite a part of a SPARQL query translating a CQ as target selection in SHACL. Alternatively, the prerequisite part of the query can be represented as a SPARQL-based target (Holger Knublauch, Allemang, and Steyskal 2017). An entity-centric partitioning can be implemented by defining a SPARQL-based target that covers a concise bounded description (Stickler 2005). Some data sources for the knowledge graph may natively partition data to independent subsets, such as API responses or messages from a queue, that can be effectively validated by specific subsets of SHACL constraints. SHACL also defines the criteria of correctness of the SPARQL queries. An `sh:ask` query is expected to return the boolean `true`, while any results produced by an `sh:select` query are treated as violations. Specific expected results can be either hard-coded into the queries or specified via `sh:hasValue` in case a single value is expected. When including the expected results in SPARQL-based constraints, we may run into the limitations of SHACL due to pre-binding of variables.[3] For example, the `VALUES` clause, which could represent results of `SELECT` queries, is not allowed.

6. Develop a transformation of the source data to the target knowledge graph that is expected to meet the constraints. This can be implemented in many ways, largely dependent on the format of the source data, so we will not cover it here. For an overview of these approaches, see e.g., Fensel et al. (2020).

7. Validate the knowledge graph under development with the SHACL constraints. If the validation fails, resolve the reported violations by fixing the artifacts created in the previous steps. Fix the most primary artifact causing the violations. For instance, data transformation shall not work around an insufficiently expressive ontology. Similarly, imposing a more expressive data model on the knowledge graph might reveal errors in its source data. In this way, the finer structure of its ontology makes the previously hidden data quality issues visible. Some feedback may even indicate ill-formed CQs in need of revision.

[2]https://www.w3.org/TR/shacl/#targets
[3]https://www.w3.org/TR/shacl/#pre-binding

8. Refactor the artifacts. The SHACL specification of the knowledge graph is a key artifact to refactor. It can serve as a formal specification of the anticipated use of the ontology. Moreover, SHACL can transcend a single ontology and specify how to combine terms reused from multiple ontologies. SHACL core constraints can be induced from SPARQL-based constraints to represent the frequently queried patterns in data. For instance, you can infer universally quantified axioms represented as SHACL constraints from a sample of existentially quantified axioms represented as SPARQL queries. In particular, SHACL can represent the expected relations in data. For example, more complex or distant relations might require the expressivity of SHACL to be represented, using property paths or SPARQL graph patterns. Even though SHACL does not support conditional constraints directly, they can be rewritten according to the inference rule of material implication $P \rightarrow Q \Leftrightarrow \neg P \vee Q$.

Non-functional requirements

Note that the above-described method tests if a knowledge graph meets the given user requirements. It does not evaluate how well these requirements are satisfied. Absence of errors does not imply that the knowledge graph is sound. Therefore, the functional user requirements should be complemented by non-functional requirements describing the desired qualities of the solution. These requirements can be checked and refactored by using alternative sources of feedback, such as:

Code review Elicit expert insight from ontology and data engineers.

User feedback The results produced by SPARQL queries formalizing the CQs can be judged to be incorrect by users. User feedback can identify incorrect interpretation of CQs or incorrect assumptions. It might prompt revisiting any of the previously described steps or lead to adding more CQs.

Usability Usability may manifest as developer experience of writing SPARQL queries for the knowledge graph. In particular, complexity and verbosity of SPARQL queries may be considered. In this way, usability of the queries indirectly reflects the usability of the ontology. For example, verbosity may be caused by duplicate data that can be abstracted to the ontology, while complexity can be reduced by introducing ontological shortcuts. Verbose SPARQL queries may also suggest a need for introducing abstraction or additional axioms into the ontology. In fact, some CQs can be answered directly using the knowledge formalized in the ontology, since SHACL expects the validated data graph[4] to include the ontologies it populates.

Performance Performance can be evaluated indirectly using the execution time of SHACL validation. This time is proportional to the execution times of the SPARQL queries the constraints include.

[4]https://www.w3.org/TR/shacl/#data-graph

Using the above-mentioned feedback is a balancing act of multi-objective optimization. For example, performance feedback may be incompatible with the feedback about verbosity. CQs may pose conflicting requirements. Addressing the feedback thus requires making informed trade-offs.

Limitations

As can be expected, the proposed method has several limitations. Here we recount some of them and suggest how to remedy them. The method may cause the developed artifacts to overfit the user requirements in scope, which would hinder the reuse of the artifacts. This shortcoming can be remedied by including more CQs or focusing on reusability during refactoring. User requirements for the knowledge graph under construction can be more complex than what SPARQL can express. One way around it is formalizing the requirements in a more expressive programming language that extends SHACL, such as Holgen Knublauch and Maria (2017). Ultimately, in order to ameliorate the limitations of this method, it is best combined with other methods, such as those for ontology design.

Case study: Antigen covid test knowledge graph

We used the described method to create a knowledge graph about antigen tests for SARS-CoV-2. Its source data was originally gathered from regulatory bodies for medical devices to perform statistical analyses of diagnostic performance of the antigen tests (Kliegr et al. 2022). The source data includes evaluations of diagnostic performance of antigen covid tests, such as diagnostic sensitivity and specificity, that comes from several regulatory bodies from various countries and covers evaluations both from the tests' manufacturers and independent research institutes. The data was integrated into a XML file for the purposes of the afore-mentioned analyses. We set to create a knowledge graph out of this data to open it to a wider reuse.

We started by capturing user requirements formulated as CQs, such as *"What is the sensitivity of given tests according to their manufacturers?"*. We analysed the questions and extracted terms and relations, such as *"sensitivity"*, *"test"*, or *"has manufacturer"* in the question given as an example.

```
:DiagnosticSensitivity a rdfs:Class ;
  rdfs:label "Diagnostic sensitivity"@en .

:AntigenCovidTest a rdfs:Class ;
  rdfs:label "Antigen covid test"@en .
```

```
:hasManufacturer a rdf:Property ;
  rdfs:label "Has manufacturer"@en .

:hasTest a rdf:Property ;
  rdfs:label "Has test"@en .

:hasAuthor a rdf:Property ;
  rdfs:label "Has author"@en .

:hasValue a rdf:Property ;
  rdfs:label "Has value"@en .
```

We followed the convention of prefixing the pre-bound SPARQL variables with `$`:

```
ASK {
  $test a :AntigenCovidTest ;
    :hasManufacturer ?manufacturer .

  [] a :DiagnosticSensitivity ;
    :hasTest $test ;
    :hasAuthor ?manufacturer ;
    :hasValue ?sensitivity .
}
```

Ultimately, we translated this CQ into the following SPARQL ASK query:

```
ASK {
  $test a act:AntigenCovidTest ;
    schema:manufacturer ?manufacturer .

  [] a ncit:C41394 ;
    dcterms:subject $test ;
    dcterms:creator ?manufacturer ;
    rdf:value ?sensitivity .
}
```

We transformed the input XML data to RDF/XML via an XSL transformation followed by SPARQL Update operations for post-processing. We automated the data processing and test execution by a script based on Jena command-line tools.[5] This is a small knowledge graph of around 10 thousand RDF triples, so we validated it as a whole.

All artifacts we developed in this effort, such as CQs, are available as open source.[6]

[5]https://jena.apache.org/documentation/tools
[6]https://github.com/jindrichmynarz/antigen-covid-tests-knowledge-graph

Conclusions

References

Beck, Kent. 2003. *Test-Driven Development: By Example.* Addison-Wesley.

Brickley, Dan, and Ramanathan V. Guha, eds. 2014. "RDF Schema 1.1." W3C Recommendation. https://www.w3.org/TR/rdf-schema.

Espinoza-Arias, Paola, Daniel Garijo, and Oscar Corcho. 2022. "Extending Ontology Engineering Practices to Facilitate Application Development." In *Knowledge Engineering and Knowledge Management*, edited by Oscar Corcho, Laura Hollink, Oliver Kutz, Nicolas Troquard, and Fajar J. Ekaputra, 19–35. Cham: Springer.

Fensel, Dieter, Umutcan Şimşek, Kevin Angele, Elwin Huaman, Elias Kärle, Oleksandra Panasiuk, Ioan Toma, Jürgen Umbrich, and Alexander Wahler. 2020. "How to Build a Knowledge Graph." In *Knowledge Graphs: Methodology, Tools and Selected Use Cases*, 11–68. Cham: Springer. https://doi.org/10.1007/978-3-030-37439-6_2.

Fink, George, and Matt Bishop. 1997. "Property-Based Testing: A New Approach to Testing for Assurance." *SIGSOFT Software Engineering Notes* 22 (4): 74--80. https://doi.org/10.1145/263244.263267.

Gruninger, Michael, and Mark S Fox. 1994. "The Design and Evaluation of Ontologies for Enterprise Engineering." In *Workshop on Implemented Ontologies, European Conference on Artificial Intelligence (ECAI).*

Harris, Steve, and Andy Seaborne, eds. 2013. "SPARQL 1.1 Query Language." W3C Recommendation. http://www.w3.org/TR/sparql11-query.

Hogan, Aidan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, et al. 2021. "Knowledge Graphs." *ACM Computing Survey* 54 (4). https://doi.org/10.1145/3447772.

Kliegr, Tomáš, Jiří Jarkovský, Helena Jiřincová, Jaroslav Kuchař, Tomáš Karel, and Ruth Tachezy. 2022. "Role of Population and Test Characteristics in Antigen-Based SARS-CoV-2 Diagnosis, Czechia, August to November 2021." *Euro Surveillance* 27. https://doi.org/10.2807/1560-7917.ES.2022.27.33.2200070.

Knublauch, Holgen, and Pano Maria, eds. 2017. "SHACL JavaScript Extensions." W3C Working Group Note. https://www.w3.org/TR/shacl-js.

Knublauch, Holger, Dean Allemang, and Simon Steyskal, eds. 2017. "SHACL Advanced Features." W3C Working Group Note. https://www.w3.org/TR/shacl-af.

Knublauch, Holger, and Dimitris Kontokostas, eds. 2017. "Shapes Constraint Language (SHACL)." W3C Recommendation. https://www.w3.org/TR/shacl.

Pan, Jeff Z., Nico Matentzoglu, Caroline Jay, Markel Vigo, and Yuting Zhao. 2017. "Understanding Author Intentions: Test Driven Knowledge Graph Construction." In *Reasoning Web: Logical Foundation of Knowledge Graph Construction and Query Answering: 12th International Summer School 2016,* *Aberdeen, UK, September 5–9, 2016, Tutorial Lectures*, edited by Jeff Z. Pan, Diego Calvanese, Thomas Eiter, Ian Horrocks, Michael Kifer, Fangzhen Lin, and Yuting Zhao, 1–26. Cham: Springer. https://doi.org/10.1007/978-3-319-49493-7_1.

Peroni, Silvio. 2016. "A Simplified Agile Methodology for Ontology Development." In *OWL: Experiences and Directions – Reasoner Evaluation*, 55–69. Cham: Springer.

Poveda-Villalón, María, Alba Fernández-Izquierdo, Mariano Fernández-López, and Raúl García-Castro. 2022. "LOT: An Industrial Oriented Ontology Engineering Framework." *Engineering Applications of Artificial Intelligence* 111. https://doi.org/https://doi.org/10.1016/j.engappai.2022.104755.

Ren, Yuan, Artemis Parvizi, Chris Mellish, Jeff Z. Pan, Kees van Deemter, and Robert Stevens. 2014. "Towards Competency Question-Driven Ontology Authoring." In *The Semantic Web: Trends and Challenges*, edited by Valentina Presutti, Claudia d'Amato, Fabien Gandon, Mathieu d'Aquin, Steffen Staab, and Anna Tordai, 752–67. Cham: Springer. https://link.springer.com/content/pdf/10.1007/978-3-319-07443-6_50.pdf.

Shadbolt, Nigel R., and Paul R. Smart. 2015. "Knowledge Elicitation." In *Evaluation of Human Work*, edited by John R. Wilson and Sarah Sharples, 4th ed., 163–200. Boca Raton: CRC Press.

Soldatova, Larisa, Panče Panov, and Sašo Džeroski. 2016. "Ontology Engineering: From an Art to a Craft." In *Ontology Engineering*, edited by Valentina Tamma, Mauro Dragoni, Rafael Gonçalves, and Agnieszka Ławrynowicz, 174–81. Cham: Springer.

Stickler, Patrick. 2005. "CBD: Concise Bounded Description." {W3C} Member Submission. W3C. https://www.w3.org/Submission/CBD.

Zemmouchi-Ghomari, Leila, and Abdessamed Réda Ghomari. 2013. "Translating Natural Language Competency Questions into SPARQL Queries: A Case Study." In *The First International Conference on Building and Exploring Web Based Environments*, 81–86.