A *Linked Data* approach to surfacing a software chrestomathy

Kevin Klein, Ralf Lämmel, Martin Leinberger,
Thomas Schmorleiz, and Andrei Varanovich

Software Languages Team
University of Koblenz-Landau, Germany

Abstract. A software chrestomathy collects 'little software systems' and their documentation; those systems implement certain requirements (tasks, features, etc.). Many different languages, technologies, and concepts are exercised in this manner. A key challenge in this context is to expose the various involved resources in a way that is the most useful for the stakeholders (such as software analysts and consumers of software knowledge). In particular, the resources should be conveniently explorable; navigation should be discoverable and feasible for all relationships (links) between resources; the relevant formats and the underlying ontology should be accessible and documented; both programmatic and interactive access should be generally supported. In the 101companies project (also: 101project or just 101), we have enriched a software chrestomathy according to *Linked Data* principles to address the stated challenge. We describe the design and the implementation of the corresponding evolution of 101. We evaluate the resulting expressiveness and convenience by addressing diverse software (language) engineering scenarios based on programmatic access to the resources of the chrestomathy.

1 Introduction

The 101companies project[1] (also: 101project or just 101) [6] is dedicated to the collection of software knowledge, more specifically knowledge about software concepts, software languages, and software technologies. To this end, 101 adopts the notion of a software chrestomathy such that it collects many 'little software systems'—the so-called *contributions*—that implement some of a given set of mainly optional features in many different ways while also adding highly structured documentation to each contribution, while also relying on general documentation of software concepts, software languages, and software technologies.

As one can imagine, such a chrestomathy is a highly heterogeneous collection of resources with much relationships and much dependencies on an ontological dimension. Since 101 is aimed at representing and conveying *knowledge*, every effort must be made to serve well the knowledge consumer. In particular, the

[1] `http://101companies.org/`

resources should be conveniently explorable; navigation should be discoverable and feasible for all relationships (links) between resources; the relevant formats and the underlying ontology should be accessible and documented; both programmatic and interactive access should be generally supported. This paper describes the evolution of 101 to address said challenge with the help of *Linked Data* principles.

The basic *Linked Data* principes are quickly recalled [4,9]:

1. Use URIs as names for 'things'.
2. Use HTTP URIs so that people can look up names.
3. Provide useful information in the HTTP response.
4. Use standards for response formats and query languages (RDF, SPARQL).
5. Include links to other URIs, so that one can discover more things.

When applied to 101, these principles imply that all wiki pages, all source code units, all derived resources including metadata, and all ontological entities (software concepts, languages, technologies) must be referable through HTTP URIs with responses that reveal content, metadata, and semantic links to other resources, also including external resources.

The additional challenge lies in the heterogeneous setup at hand. Source-code repository (101repo), wiki-based documentation (101wiki), and derivatives (as computed by so-called 101worker modules) evolve independently and with different scopes of change. Thus, we need to surface 101 in a way that confederates all resources in a useful and sufficiently efficient manner. This disqualifies, for example, a naive approach of dumping all data into a triplestore as its consistence would be hard to maintain.

We think of this work as properly publishing 101's ontology and content so that the underlying paradigm and all available knowledge is made available more appropriately for validation.

Contributions

- We describe the process of organizing a highly heterogeneous collection of resources according to *Linked Data* principles so that they can be effectively addressed, interactively explored, and programmatically processed. This is the first time that such a process has been developed and implemented in a software (language) engineering context.
- We demonstrate the resulting expressiveness and convenience for software language engineers by addressing diverse software (language) engineering problems on top of 101's software chrestomathy. This evaluation effort includes code sharing detection of system variants and complexity comparison across different languages.

Relation to our previous work [6] introduced 101 broadly; no *Linked Data* approach was available or conceived at that time. [7] linked entities in megamodels to resources on 101 and [5] dealt with the architectural recovery aspects of links between 101 source artifacts and documentation. In the latter two cases, some

Linked Data ideas surfaced in passing. The present paper presents a complete design and implementation of an *Linked Data* approach for 101 and it presents the power of the approach with S(L)E scenarios that were previously inconvenient, if not infeasible. This approach addresses the increasing heterogeneity and complexity of 101.

Road-map of the paper §2 takes an inventory of 101's resources. §3 describes requirements for a suitable *Linked Data* approach. §4 develops schemas for the previously described resources and links. §5 sketches the 101explorer service which is a web-based service for the exploration of 101, which can be used both interactively and programmatically. §6 evaluates the expressiveness and convenience of the proposal by means of some software (language) engineering scenarios. Related work is discussed in §7 and the paper is concluded in §8.

2 Inventory of 101's resources

101 is based on the following major resources: the 101repo with folders, files, and fragments, the 101wiki with pages that are linked in certain ways, derivatives computed by the 101worker, and external resources. In discussing these resources, we already begin to take a *Linked Data*-centric point of view, but defer several *Linked Data* requirements to the next section.

2.1 The 101repo

All source code for contributions and other illustrations are stored in 101's repository: the 101repo. This is a virtual repo in that it consists of many physical repos that contribute to 101. This principle of confederation is needed to enable scalability as well as collaborative or loosely coupled development.

For some time now, 101 leverages GitHub as the underlying repo platform. The confederation of 101repo from the contributing repos relies on a 101-specific mechanism comparable to but more general than GitHub's module mechanism. That is, 101repo maintains a registry of contributing repos and metadata for mounting those repos in the 101repo tree. This registry is also maintained by the submission and administration services for contributions.

2.2 File fragments

Files in the 101repo are not monolithic resources; rather they are of a nested container-like structure, subject to the association of a software language with the file and assuming the availability of fact extraction and fragment location capabilities for the language. Here is a JSON-based representation of the fragment structure (as understood by 101) of a Haskell-based stack implementation:

```
[ { classifier: "data", name: "Stack" },
  { classifier: "pattern", name: "empty" },
  { classifier: "pattern", name: "isEmpty" },
  { classifier: "function", name: "push" }, ... ]
```

- Namespace: all namespaces including those listed below.
- Language: software languages such as Haskell, XML, or UML.
- Technology: software technologies such as JUnit, GitHub, or Ruby on Rails.
- Concept: software concepts such as parsing, abstraction, or visitor pattern.
- Feature: features (requirements) of the 101system such as Cut or Total.
- Contribution: implementations, models, etc. of the 101system.
- Contributor: open-source developers and wiki authors contributing to the 101project.
- Theme: themes (collections) of contributions addressing stakeholder perspectives.
- Vocabulary: vocabularies of software concepts, e.g., for Software engineering.
- Course: open online courses leveraging resources of the 101project.
- Script: scripts for individual lectures, labs, etc. in courses.
- Module: modules of the 101worker deriving resources and dumps.

Fig. 1. Namespaces managed on the 101wiki (shortlisted).

2.3 The 101wiki

The wiki comprises of wiki pages, which in turn break down into sections, which may also be addressed, in principle, in a URI-based manner. Pages refer to each other via plain links or semantic properties (see below). All pages are organized in namespaces to distinguish major content categories on the wiki; see Figure 1.

The namespace-based organization of the 101wiki and the virtual layout of the 101repo are designed to be in sync. That is, the top-level folders of the repo correspond to the namespaces on the wiki. The second-level folders of the repo correspond to the member pages on the wiki. Beyond that level, all files and folders are conceptually associated with the member page.

2.4 Semantic properties

Recently, the 101wiki has been turned into a semantic wiki, inspired by Semantic MediaWiki [10][2]. Semantic properties are used specifically for assigning 'types' to links. Consider, for example, the following links as rendered on the 101wiki page for the Prolog language:

◀ this *instanceOf* Logic programming language
◀ this *instanceOf* Namespace:Language
▶ Contribution:prologStarter *uses* this

'this' proxies for the current page (i.e., the page for Prolog). There are three links in which 'this' is involved. First, 'this' is said to be an 'instance of' the concept 'Logic programming language'. Second, 'this' is said to be an 'instance of' (as in 'element of') the namespace 'Language'. Third, a certain contribution, i.e., 'prologStarter' is said to 'use' 'this' (i.e., Prolog). The first two links (with

[2] `http://semantic-mediawiki.org/`

Predicate	Meaning	# Triples
uses	A resource uses a language or technology.	565
implements	A contribution implements a feature.	628
instanceOf	'instance of' relationship	1578
isA	'is-a' relationship on concepts	84
developedBy	A contribution is developed by a contributor.	191
reviewedBy	A contribution is reviewed by a contributor.	16
relatesTo	A resource relates to (ontological) to another resource.	80
mentions	A resource mentions another resource (weak internal link).	4717

Fig. 2. Properties (types) for typed links internal to the 101wiki.

'this' on the left) reside on the page of 'this'. The third link resides on said contribution page.

Figure 2 lists most properties used currently on the 101wiki. (The counts are supposed to go up, as more pages make comprehensive use of the new style.) Overall, the use of namespaces and properties clearly contributes to a *Linked Data* approach for 101.

2.5 External resources

The semantic approach for internal links is complemented by a similar approach for links to external resources. From a *Linked Data* point of view, links to external websites should be typed again by a property as opposed to plain (semantically weak) links. We have started to use semantically strong links as follows. A wiki page for a resource r can qualify its link to a web site w with a predicate.

Predicate	Meaning	# Triples
identifies	w is designated to the resource r at hand.	384
linksTo	w is concerned with the resource r at hand.	198

In the first case, the idea is that the r and w are ontologically (about) the same resource. In the second case, the idea is to express that w is highly relevant to r; it relates to r, but there is some ontological mismatch so that they cannot be considered the same. For instance, 101wiki's 'Abstraction' links to Wikipedia's 'Abstraction (computer science)' with predicate 'identifies'. By contrast, 101wiki's 'Abstraction mechanism' links to the same Wikipedia page with predicate 'linksTo', as there is no designated Wikipedia page (at the time of writing), but the page on 'Abstraction' discusses an ontologically close concept.

2.6 Derived resources and dumps

The 101worker analyzes primary resources and synthesizes derived resources or dumps ('derivatives'). All these analyses and computations are modularized and composed in a pipeline to deal with module dependencies (through derivatives).

A derived resource is basically a file that associates with a file in the 101repo. Given, for example, a repo file f, there exist derived files like this (and yet others):

- f.metrics.json: summary of basic metrics of f such as LOC.
- f.extractor.json: facts extracted from f by a fact extractor.
- f.fragments.json: a list of fragment descriptors for f.
- f.matches.json: metadata units assigned to f via 101meta rules [5].
- f.commitInfo.json: contributor data inferred from the GitHub history.
- ...

Each of these files has its own format; use of JSON is popular, but other formats are also used occasionally.

Dumps are used to represent larger data artifacts or the aggregated representation of certain derived resources. Consider these examples of dumps:

- A dump of the wiki content (e.g., in JSON) for structured text processing.
- A dump of all metadata units for all files in the repo.
- Dumps (logs) of schema validation in the repository.
- Dumps (views) of concept usage in themes of contributions.
- ...

3 101's *Linked Data* requirements

Given the heterogeneity and multiplicity of 101 resources as well as the rich semantic dependencies between them, it is clear that the resources should be linked. In the past, we realized some forms of links in some ad-hoc manner, e.g., by deploying a wiki plugin to support navigation from the wiki to the repo. Eventually, the linking requirements were rich enough to trigger a more systematic approach. In this section, we list requirements.

3.1 Navigate from wiki to repo

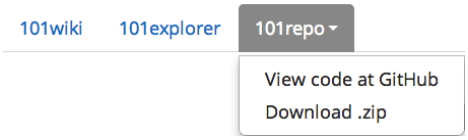
In the days of a centralized 101repo (i.e., with one physical repo), we added clickable links on contribution pages on the 101wiki to be taken to the associated folder in the physical repo. This was easy as we simply relied on a name convention shared between wiki and repo. When we switched to a distributed repo, we had to use a plugin to enable navigation from the wiki to the repo; the plugin had to interpret the registry of the confederated repo. More recently, we established a dump to map wiki pages to GitHub repo folders, which can be trivially interpreted by a wiki plugin. Such mapping is no longer restricted to contributions; it is used for all namespaces.

3.2 Navigate from repo to wiki

Navigation between repo and wiki shall be bidirectional, in fact. The wiki is under our control and thus, wiki-to-repo navigation is relatively expected and straightforward. By contrast, the repo is not completely under our control, but plug-in mechanisms could possibly be used to enable this direction. We favor yet another approach: an extra explorable view on the repo from which one can

navigate both to the physical repo and the wiki. The explorable view would also collect other links to related resources.

For illustration, when positioned on a wiki page for a contribution, the following navigation options are required:



That is, one may navigate from the wiki to the repo. However, one may also navigate to the 101explorer (see §5), which is exactly the aforementioned, explorable view on the repo, from which one can go both ways: to the repo and to the wiki.

3.3 Reference fragments on wiki pages

Contributions are documented in two ways: as regular source code and as contributions on the 101wiki. Regular documentation is a matter of best software engineering practices; we decided that it should not be affected (disrupted) by 101-specific concerns. The documentation on the 101wiki should be selective to highlight the more interesting and distinguished aspects of the source code. Further, the wiki-based documentation would fully embrace ontology of the chrestomathy; such documentation shall not be injected into the source code, as it would be hard to author and maintain this way, as it would disturb the developer view on the source code.

We address this challenge by allowing wiki content to refer to source folders, source files, and—most importantly—source-code fragments in a systematic manner. For instance, consider the following markup which appears on a wiki page for a simple Haskell-based contribution:

```
<fragment url="src/Main.hs/type/Company"/>
```

The URL is given relative to the contribution page. The absolute(d) URL points to a file and, in fact, to a fragment in the file: a type declaration for *Company*. Thus, the fragment is rendered as follows:

Explore

```haskell
-- Companies as pairs of name and employee list
type Company = (Name, [Employee])
```

Please observe the 'Explore' link which should take one to the explorable view on the 101repo, as discussed before. We refer to this style of documentation as 'inverted literate programming' as the documentation refers to the code as opposed to the code to contain the documentation. Such referencing of fragments relies on the concepts of fragment descriptor and fragment location, as we introduced them elsewhere for the sake 101's metadata framework [5]. In fact, the original proppsal was not compatible with *Linked Data*, as we assumed DSLs for fragment description, while a URI-based approach is needed now.

3.4 Associate derived resources with primary resources

Primary and derived resources (§2.6) should be properly linked. Otherwise, the derived resources are hard to discover. This would be a violation of a *Linked Data* principle. Actually, an effective association of primary and derived resources should also include links to the relevant modules. Thus:

- For each 'primary' file, link to all 'derived' files.
- For each 'derived' file, link to the 'primary' file.
- For each 'derived' file, link to the producing module.
- Likewise, for dumps.
- ...

3.5 Operate on the wiki like a graph

Conceptually, the 101wiki is a graph; pages are connected through links (and backwards, through backlinks). We need to expose the 101wiki as a graph proper with pages as nodes and semantic links as edges. Updates of the wiki affect the graph in a controlled (scoped) manner. Thus, we can maintain a triplestore for the wiki.

3.6 Operate on the repo like a tree

Conceptually, the 101wiki is a tree; this is obviously also the way how a repo browser provides access; this is how the repo can be materialized in a file system. It is important to expose the 101repo as tree (rather than a graph) because maintenance of a triplestore for the repo would be challenging in terms of the dynamicity of changes on the confederated repo and the derived resources under the control of many different 101worker modules. The tree experience implies that it must be possible to transition from a folder to all the files in the folder or to return from a file to its parent folder.

4 101's *Linked Data* schemas

We have reached the point where we can metamodel 101's *Linked Data* approach effectively. We sketch the schemas that are used for the various *Linked Data* views and resources.

4.1 The schema of the 101repo

Figure 3 summarize the schema for an explorable view on the 101repo. There are classes for namespaces, folders, files, and fragments with the obvious composition relationships. Modules are mentioned as special namespaces members as modules are the key resources to connect primary resources to derived resources.

JSON and RDF are supported for data representation for programmatic access as well as HTML rendering for interactive end-user needs. The corresponding

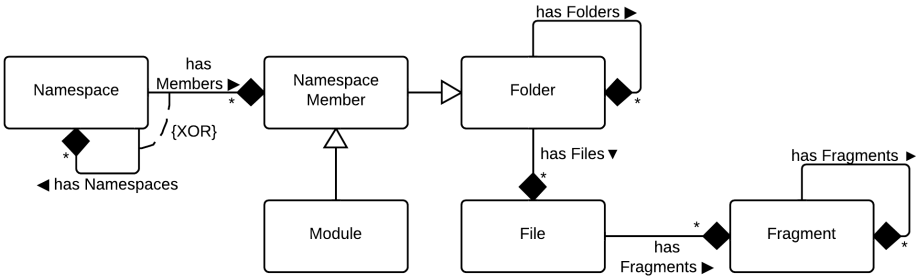


Fig. 3. UML-based metamodel of the 101repo (summary).

```
{
  "title": "Fragment schema",
  "type" : "object",
  "properties": {
    "name" : { "type": "string" },
    "namespace" : { "type": "string" },
    "headline" : { "type": "string" },
    "wiki" : { "type": "url" },
    "github" : { "type": "url" },
    "triplestore" : { "type": "url" },
    "classifier" : { "type": "string" },
    "language": { "type" : "string" },
    "content" : { "type" : "string" },
    "fragments" : {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "resource" : { "type": "resource" },
          "classifier" : { "type": "string" },
          "name" : { "type": "string" },
        }}}
  }
}
```

Fig. 4. JSON schema for fragments of the 101repo.

detailed JSON and RDF schemas are too complex to inline them into this paper. Figure 4 sketches the schema elements for file fragments. One should note the various properties for external links: the link to GitHub, the 101wiki, and the triplestore for the wiki. Further, the sub-fragments are also listed, thereby guiding continued discovery of resources.

```
{
  "derivatives" : [
    {
      "headline" : "Metrics",
      "scope" : "file",
      "suffix" : ".metrics.json",
      "language" : "JSON"
    },
    {
      "headline" : "Tokens",
      "scope" : "file",
      "suffix" : ".tokens.json",
      "language" : "JSON"
    }
  ]
}
```

Fig. 5. Description of a module for synthesizing software metrics data.

4.2 The schema of the 101worker's modules

Each 101worker module is described in a way that the description can be interpreted for drawing links between primary resources and derived files. Consider Figure 5 for illustration. The description states that the module actually constructs two derives resources for each file of the 101repo and it declares the file suffix used to create the derivative's filename from the primary file's name.

The corresponding schema for module descriptions is (partly) shown in Figure 6. Thus, derivatives may have different scopes: file, folder, dump; they may use different data models for representation: RDF and JSON.

4.3 The schema of the 101wiki's triplestore

The triplestore for the 101wiki basically needs to maintain resources for all pages (i.e., all members of all namespaces) and triples for all kinds of semantic links. Thus, there are RDFS classes for all namespaces (see Figure 1) and appropriately constrained RDF properties for all semantic properties (see Figure 2 and properties for external resources). A small part of the RDF schema is shown in Figure 7.

4.4 Miscellaneous schemas

101 relies on further schemas, which we mention here only in passing, as they are less relevant for in this paper. There is a metamodel for wiki content for each namespace. This metamodel constrains the semi-structured content on wiki pages. There is also a feature model for the features of the 101system. This model constrains valid feature configurations, as expressed through semantic links on the 101wiki.

```
{
  "title": "Module description schema",
  "type" : "object",
  "properties": {
    "derivatives" : {
      "type" : "array",
      "items": {
        "type": "object",
        "properties": {
          "headline" : { "type": "string" },
          "scope" : { "enum": ["file", "folder", "dump"] },
          "suffix" : { "type": "string" },
          "filename": { "type": "string" },
          "language" : { "enum": ["JSON", "RDF"] }
        }
      }
    }
  }
}
```

Fig. 6. JSON schema for 101worker's module descriptions.

```
<rdf:RDF xmlns:rdf="..." xmlns:rdfs="...">

  <rdfs:Class rdf:ID="http://101companies.org/schemas/wiki#Namespace"/>
  <rdfs:Class rdf:ID="http://101companies.org/schemas/wiki#Language"/>
  <rdfs:Class rdf:ID="http://101companies.org/schemas/wiki#Technology"/>
  <!-- Further classes (namespaces) omitted. -->

  <rdf:Property rdf:about="http://101companies.org/schemas/wiki#uses">
    <rdfs:range rdf:resource="http://101companies.org/schemas/wiki#Technology"/>
    <rdfs:range rdf:resource="http://101companies.org/schemas/wiki#Language"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/wiki#Contribution"/>
  </rdf:Property>

  <rdf:Property rdf:about="http://101companies.org/schemas/wiki#implements">
    <rdfs:range rdf:resource="http://101companies.org/schemas/wiki#Feature"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/wiki#Contribution"/>
  </rdf:Property>

  <!-- Further properties omitted. -->

</rdf:RDF>
```

Fig. 7. Part of the RDF schema for the triplestore.

5 An exploration service

The triplestore of the 101wiki can be accessed in a standard manner with support, for example, for a SPARQL endpoint. The explorable view of the 101repo relies

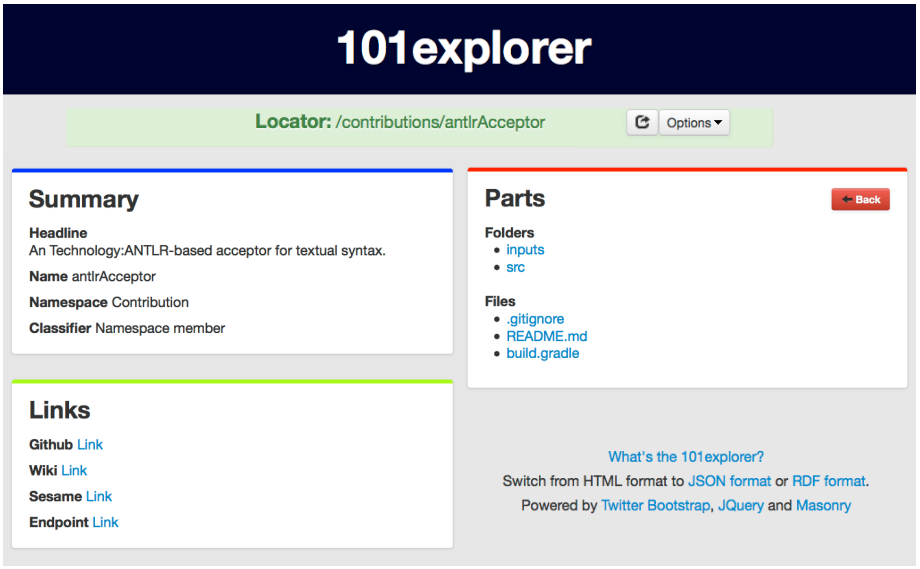


Fig. 8. Screenshot of the 101explorer's HTML view.

on a simple HTTP-based service, the 101explorer, which constructs linked data from URIs for namespace members, folders, files, and fragments on the fly.

To get a first idea, consider the snapshot in Figure 8. The 101explorer can return RDF, JSON, and HTML. The HTML format is convenient in that, this way, the 101explorer can be used immediately as an interactive tool for true exploration. For instance, one can drill into the files of a contribution and then copy and paste the URI for a fragment to be used as markup on a wiki page.

The architecture of the 101explorer is summarized in Figure 9. In essence, the service takes advantages of the 101worker infrastructure which already has materialized various data on the server side. In particular, the 101repo is materialized with all primary resources, module descriptions, etc.. Further, all derivatives are materialized, e.g., the dump needed to connect URIs to GitHub.

6 Scenario-based evaluation

A new way of working with 101's chresomapthy is enabled. There is no need to download any files and run any 101 functionality locally. Programmatic access leverages the fact that all data is schema-aware and that the actual access protocols support the tree view on the repo and the graph view on the wiki. We pick three software (language) engineering scenarios for illustration: a simple form of clone detection, naive metrics-based comparison of contributions, and a simple concept analysis.

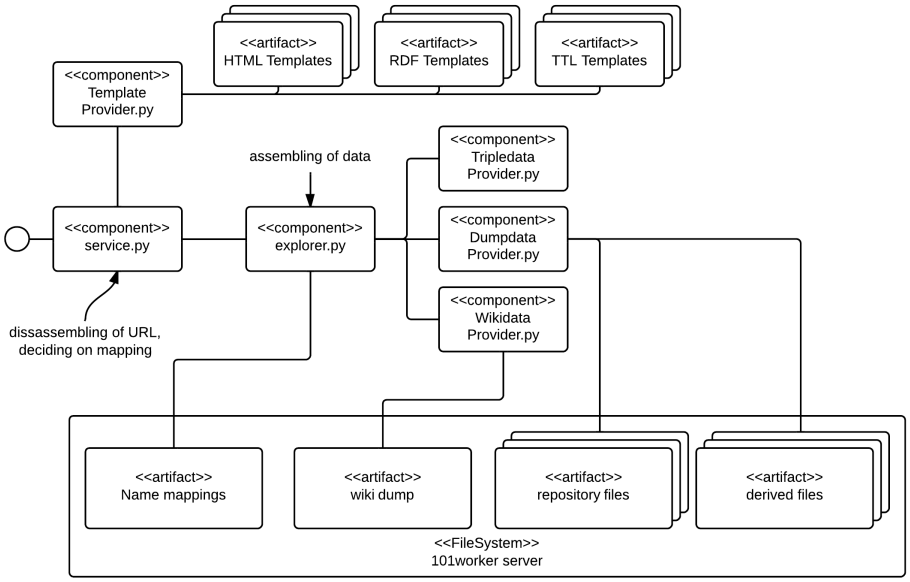


Fig. 9. Architecture of the 101explorer.

6.1 Code-sharing management

A software chrestomathy contains 'little systems' (contributions) that are similar by design, and hence, one can expect to detect clones. We are working on an approach to use clone detection in a way to actually manage the similarity of contributions to help with understanding and evolution. The following scenario is adopted from this ongoing work.

We can basically traverse the tree-like structure of the *Linked Data* exposed for the 101repo and map file content to file URIs. In this manner, we can determine groups of perfect clones. See Figure 10 for some groups: an XML schema appears in several contributions; some Haskell-based contributions use the same data model; some Java-based contributions implement some operations in the same manner. This approach can be refined to work with less than perfect clones and it can also be used in a more advanced manner at the fragment level.

Figure 11 contains the (slightly simplified) code for the clone detector. All data access boils down to 'loadPage' with URIs extracted in previous steps of the walk over the 101repo starting from the root of the contributions: `http://101companies.org/resources/contributions`.

6.2 Metrics-based comparison of contributions

It is not uncommon to encounter the expectation that a software chrestomathy such as 101 should support some sort of comparison of programming languages

```
 [
  dom/Company.xsd",
  jdom/Company.xsd",
  sax/Company.xsd",
  scalaXML/Company.xsd",
  xom/Company.xsd",
  xquery/Company.xsd",
  xslt/Company.xsd"
 ],
 [
  haskellSyb/src/Company/Data.hs",
  monoidal/src/Company/Data.hs",
  nonmonadic/src/Company/Data.hs",
  writerMonad/src/Company/Data.hs"
 ],
 [
  jaxbChoice/src/test/java/org/softlang/tests/Operations.java",
  jaxbExtension/src/test/java/org/softlang/tests/Operations.java",
  jaxbSubstitution/src/test/java/org/softlang/tests/Operations.java"
 ]
```

Fig. 10. Some groups of perfect clones found in the 101repo.

on the grounds of the contributions that exercise those languages. That is, one may just determine metrics and interpret differences as being a consequence of language choice. Clearly, various threads to validity and feasibility need to be addressed before this expectation can be scientifically satisfied. Nevertheless, we show how the available programming model can be used for the technical aspect of this endeavor.

Figure 12 shows a specific feature set of the 101system and a set of contributions that implement exactly those features. The LOC metric is shown for these contributions. It happens that these are all Java-based contributions and they do not differ much in their LOC metric. (We plan to publish more on this matter.)

Figure 13 contains the (slightly simplified) code for the metrics-based comparison of contributions. This functionality involves tree walk over the repo, access to derived resources (i.e., a JSON file with metrics), and access to the wiki's triplestore (to retrieve triples for contributions to implement certain features). Thus, we face a highly heterogeneous scenario, which however is enabled well by the comprehensive links at avail.

6.3 Concept analysis

The last scenario demonstrates the graph querying capability that is supported by *Linked Data* for the 101wiki. The idea is to collect concepts as they are referenced on documentation pages for contributions and to associate them with programming paradigms for the sake of a simple concept analysis.

Consider Figure 15. A concept is listed in the box of either paradigm, if there is a contribution such that it uses a language such that it is an instance of said

```
# Collect files and content recursively
def extractFilesFromFolder(folder):
    files = []
    data = loadPage(folder['resource'])
    for file in data.get('files', []):
        fileData = loadPage(file['resource'])
        fileData = fileData.get('content')
        files.append({
            'uri' : file['resource'],
            'data': fileData
            })
    for folder in data.get('folders', []):
        files += extractFilesFromFolder(folder)
    return files

# Iterate over all contributions
filesList = []
root = 'http://101companies.org/resources/contributions'
contributions = loadPage(root)
for member in contributions['members']:
    filesList += extractFilesFromFolder(member)

# Hash map content to file URIs
contents = {}
for file in filesList:
    content = file['data']['content']
    if not content in contents:
        contents[content] = []
    contents[content].append(file['uri'])
```

Fig. 11. Python code for perfect clone detection.

paradigm. Then, we count the number of occurrences of the concept ('#Occs') and assess whether it is associated with the paradigm uniquely. For instance, 'Algebraic data type' is used a number of times with a functional programming contribution, but never in an OO programming contribution. In this way, we work towards a concept analysis to associate concepts with paradigms.

Figure 16 shows the code that computes the occurrences of concepts for a given paradigm. We use the Gremlin query language[3] for queries on the graph of the 101wiki.

7 Related work

Exposing heterogeneous software artifacts using *Linked Data* is an emerging research challenge. Instances of this idea are the following. In the context of

[3] https://github.com/thinkaurelius/titan/wiki/Gremlin-Query-Language

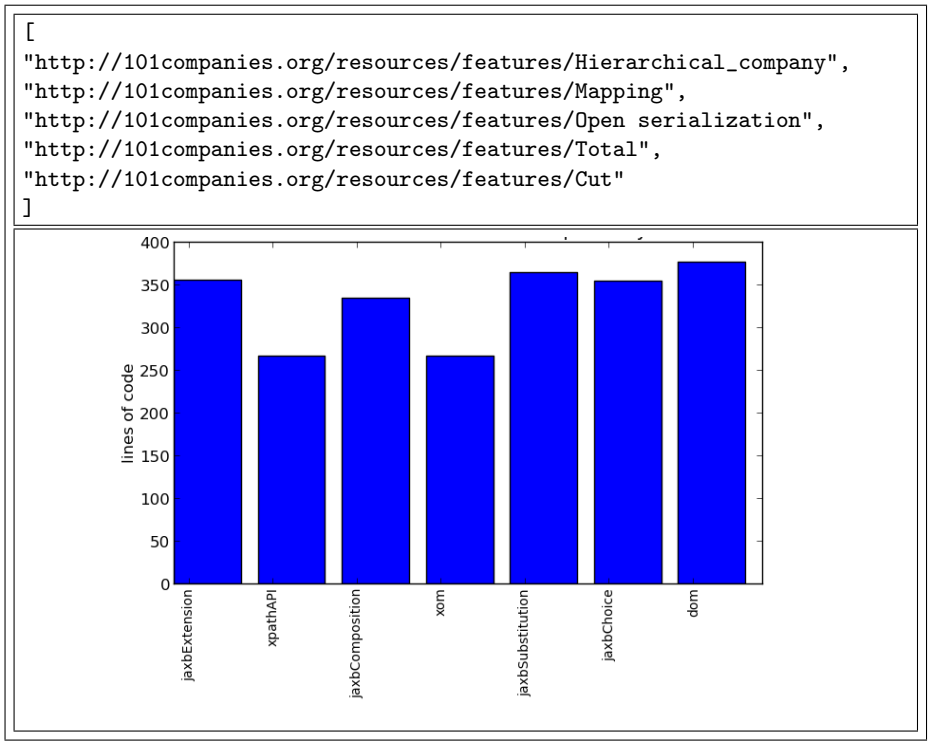


Fig. 12. A feature set with the corresponding contributions and LOC.

eGovernment, ICT systems need to support sharing of heterogeneous information and knowledge, referred to as Semantic Interoperability and defined as "highly reusable metadata (e.g., XML schemata, generic data models) and reference data (e.g., code lists, taxonomies, dictionaries, vocabularies)" [12]. This approach is implemented in a metadata vocabulary called Asset Description Metadata Schema (ADMS). In [3], RDF metadata is used to document source packages, their releases and links to other packaging artifacts, using ADMS.SW. Such an approach enables the FLOSS (Free Libre and Open Source Software) community to cross-link resources from various Linux distributions and thus, to correlate similar efforts among them.

From the application architecture perspective, our solution combines both runtime link traversal and direct querying mechanisms to address various scenarios of accessing linked data. The key challenge lies in the context of "Trust, Quality and Relevance" [9], and specifically concerned with the underlying vocabulary, which should be assessed and improved with the help of the broad SLE community. We believe, that such vocabulary, and the underlying data surfaced on top of Linked Data principles, will give a further rise of the advanced MDE languages and tools, namely *ontological* [11], *linguistic* metamodeling [1] and

```
configs = {} # Associate feature configurations with contributions
metrics = {} # Associate contributions with LOC metric

# Iterate over all contributions
contribs = loadPage('http://101companies.org/resources/contributions')
for contrib in contribs['members']:
    data = loadPage(contrib['resource'])

    # Collect features for contribution
    features = retrieveFeatures(data['triplestore'])
    key = tuple(features)
    if not key in configs:
        configs[key] = {'features': features, 'contribs': []}

    # Map feature configuration to contribution
    configs[key]['contribs'].append(contrib['name'])

    # Aggregate LOC for all files of the contribution
    files = collectFiles(contrib['resource'])
    loc = 0
    for file in files:
        mdata = retrieveMetrics(file['resource'])
        if not mdata == {}:
          if not 'relevance' in mdata or mdata['relevance'] == 'system':
                loc += int(mdata['loc'])
    metrics[contribution['name']] = loc
```

Fig. 13. Python code for metrics-based comparison. (See Figure 14 for the remainder.)

megamodeling [7], where the choice of a foundational ontology if one of the key design challenges.

To further integrate the data exposed by 101 into a global *Linked Data* cloud, the important question of publishing ontologies arises. [13] proposes an approach, which allows repositories to publish their assets' (ontologies') metadata. We consider formalizing and publishing this aspect as an important, next step. There are several efforts to propose a related standard, e.g., the so-called Ontology Metadata Vocabulary (OMV) [8], to support the creation, maintenance and distribution of such metadata. [2] discusses an initiative to develop and deploy a new federated interoperability infrastructure for metadata called the Open Ontology Repository.

8 Conclusion

We have presented a detailed, non-trivial case study on enabling an existing software engineering and programming context for *Linked Data*. That is, we enabled the 101project so that its software chrestomathy with all associated data (source code and wiki content, internal and external, primary and derived)

```
# Retrieve features for a contribution
def retrieveFeatures(url):
    triples = loadPage(url)
    features = []
    for triple in triples: # Filter triples
        predicate = triple[1]
        object = triple[2]
        if predicate == 'http://101companies.org/property/implements':
            features.append(object.replace(
                'http:/101companies.org/resources/features/', ''))
    return features

# Retrieve metrics for a file
def retrieveMetrics(uri):
    file = loadPage(uri)
    derivatives = file['derivatives']
    for derivative in derivatives: # Find associated metrics file
       if derivative['name'].endswith('metrics.json'):
            return loadPage(derivative['resource'])
    return {}

# Collect files and metrics in a folder recursively
def collectFiles(uri):
    folder = loadPage(uri)
    files = folder['files']:
    for subfolder in folder['folders']:
        files += collectFiles(subfolder['resource'])
    return files
```

Fig. 14. Figure 13 cont'd.

Functional programming

Concept	#Occs	Unique
GUI	12	false
Class	10	false
101implementation	8	false
Server	8	false
Zipper	8	false
Library	8	false
Functional programming	8	false
Float	8	false
String	8	false
Pure function	8	false
Algebraic data type	7	true

OO programming

Concept	#Occs	Unique
GUI	16	false
Class	16	false
MVC	14	false
POJO	14	true
Metamodel	13	true
101implementation	12	false
Client	12	false
Model	11	true
Server	10	false
Annotation	8	false
OO programming	8	true

Fig. 15. Concepts associated with the functional and the OO paradigms.

```
static final String resources = 'http://101companies.org/resources/'
static final String properties = 'http://101companies.org/property/'

public findConcepts(paradigm) {

   def concept = getResource(resources + 'namespaces/Concept')
   def concepts = graph.v(paradigm).
      inE(properties + 'instanceOf').outV. // Languages
      inE(properties + 'uses').outV. // Contributions
      outE(properties + 'mentions').inV. // Mentions
      toList().findAll{ // Concept mentions only
         it.outE(properties + 'instanceOf').inV.
         filter{it == concept}.toList().size() > 0
      }
   return concepts
}
```

Fig. 16. Groovy code for concept analysis for programming paradigms.

is fully linked and explorable in resource-oriented *Linked Data* fashion up to the point that one can programmatically process 101, as we have demonstrated with some diverse scenarios of software analysis in software engineering. In this manner, 101 has become more open, more connected, more standardized, more explorable, and more usable.

We suggest the following directions for future work. The schema situation of the obtained architecture is rather complex and potentially confusing. We would like to use ideas of schema mapping to reduce the redundancy and to enforce consistence across all the different schemas and type systems. Further, we would also like to work on the consistency between declared metadata (based on the properties discussed) versus inferable metadata (based on existing infrastructure for metadata inference). The idea is to push metadata inference far enough that it can be trusted up to the point that much less metadata must be explicitly declared. In this context, we should also make inferred metadata more discoverable. For instance, one would like to see concepts being associated with fragments when working in the 101explorer. Previously, we had ad-hoc tools to this end, but we would like to standardize such capabilities, indeed. Another open challenge is modeling of the consistence of the confederated data experience of 101 in the view of the heterogeneity and dynamicity of all involved resources. So far, we have applied pragmatic reasoning to arrive at a workable implementation, but we are not yet able to model these aspects and to perform any (semi-) formal reasoning.

References

1. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE 20(5), 36–41 (2003)

2. Baclawski, K., Schneider, T.: The open ontology repository initiative: Requirements and research challenges. In: Proceedings of Workshop on Collaborative Construction, Management and Linking of Structured Knowledge at the ISWC (2009)
3. Berger, O., Bac, C.: Authoritative linked data descriptions of debian source packages using adms.sw. In: Open Source Software: Quality Verification, pp. 168–181. IFIP Advances in Information and Communication Technology, Springer Berlin Heidelberg (2013)
4. Bizer, C., Cyganiak, R., Heath, T.: How to publish Linked Data on the web (2007), online tutorial `http://wifo5-03.informatik.uni-mannheim.de/bizer/pub/LinkedDataTutorial/`
5. Favre, J.M., Lammel, R., Leinberger, M., Schmorleiz, T., Varanovich, A.: Linking documentation and source code in a software chrestomathy. In: Proc. of WCRE 2012. pp. 335–344. IEEE (2012)
6. Favre, J.M., Lämmel, R., Schmorleiz, T., Varanovich, A.: *101companies*: a community project on software technologies and software languages. In: Proc. of TOOLS 2012. LNCS, vol. 7304, pp. 59–74. Springer (2012)
7. Favre, J.M., Lämmel, R., Varanovich, A.: Modeling the Linguistic Architecture of Software Products. In: Proc. of MODELS 2012. LNCS, vol. 7590, pp. 151–167. Springer (2012)
8. Hartmann, J., Palma, R., Sure, Y., Suárez-Figueroa, M.C., Haase, P., Gómez-Pérez, A., Studer, R.: Ontology metadata vocabulary and applications. In: On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops. pp. 906–915. Springer (2005)
9. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool (2011), 1st edition
10. Krötzsch, M., Vrandecic, D.: Semantic Wikipedia. In: Social Semantic Web, pp. 393–421. X.media.press, Springer (2009)
11. Laarman, A., Kurtev, I.: Ontological metamodeling with explicit instantiation. In: SLE. pp. 174–183 (2009)
12. Peristeras, V.: Open government metadata (Sep 2011), `http://joinup.ec.europa.eu/elibrary/document/towards-open-government-metadata`
13. Shukair, G., Loutas, N., Peristeras, V.: Integrating linked metadata repositories into the web of data. In: COLD (2012)