Linking Documentation and Source Code in a Software Chrestomathy

Jean-Marie Favre
University of Grenoble, France

Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, Andrei Varanovich
Software Languages Team, University of Koblenz-Landau, Germany

***Abstract*—The software chrestomathy of the *101companies* community project demonstrates 'many' software languages and software technologies by implementing 'many' variants of a human resources management system; each implementation selects from 'many' optional features. All implementations are available through a source-code repository and they are documented on a wiki. Source code and documentation encode references to software languages, software technologies, software concepts, and product features, which, by themselves, are also documented and linked on the wiki.**

This setup implies the challenges of establishing links between source code and documentation as well as verifying that source code and documentation are in agreement. We describe an approach that addresses these challenges; it relies on a rule-based system which extracts relevant information from source-code artifacts (e.g., information about language and technology usage) and assigns metadata to the artifacts (e.g., methods for validation and fact extraction). The linked source-code repository and wiki as well as various derived information resources are available through the *101ecosystem* for the benefit of the reverse engineering community.

***Index Terms*—software chrestomathy; software language; software technology; linguistic architecture; reverse engineering; source-code repository; documentation; traceability; architecture reconstruction; 101companies**

I. Introduction

Linking the expected architecture described by documentation with the actual architecture extracted from source code is a well-known reverse engineering problem [1], [2]. The present paper is specifically concerned with these kinds of links:

Actual links. These links reside in the source code. Reverse engineering techniques may recover these links. The graph of the links and the underlying entities is referred to as the *actual* (or 'as-implemented') architecture.

Expected links. These links reside in higher-level models or documentation. The corresponding graph is referred to as the *expected* (or 'as-designed') architecture.

Links to establish. In order to ensure consistency between source code and documentation, links should be established between source-code and documentation entities. These links may be documented within the code, within the documentation, or elsewhere; they may also be discovered by appropriate analyses, e.g., based on name conventions.

The present paper is specifically concerned with the *linguistic architecture* of software products [3] with links, for example, from source-code artifacts to software languages, software technologies, and software concepts. (Traceability for product features in source code is also of interest.)

The present paper focuses on *software chrestomathies* as opposed to traditional software products. A software chrestomathy collects source-code samples that exercise 'many' software languages, technologies, and concepts. Specifically, the present paper contributes to the chrestomathy of the *101companies* community project [4].[1]

Chrestomathies imply specific forms of complexity that challenge reverse engineering techniques: heterogeneity in terms of the 'many' languages and technologies used by the collected samples and variability in terms of tasks or features implemented by the samples.

Figure 1 compares traditional software products and software chrestomathies in terms of the kinds of links, as discussed above. Corresponding associations are depicted informally. (In UML terminology, links are instances of associations.)

The approach of the present paper is loosely inspired by methodologies for *software architecture reconstruction* (SAR) including Symphony [5], *CacOphoNy* [6], and others [7], [8], [9]. However, previous research on SAR addressed traditional architecture (as opposed to linguistic architecture; see [10] for a survey) and traditional products (as opposed to software chrestomathies).

Contributions

C1. A simple approach to establishing links between source code and documentation is described. The approach relies on rules that pattern-match on names and content of files or that invoke file processors for validation, fact extraction, fragment location, and others. The approach can handle many languages and technologies while using techniques on a scale of language-specific to language-agnostic.

C2. The proposed techniques are demonstrated for the *101companies* chrestomathy. We exercised the techniques for a variety of software languages and software technologies.

C3. The refined *101companies* chrestomathy as well as the underlying techniques and derived information resources are available as the open source *101ecosystem*, in support of *Research 2.0* and, in particular, *Open Science* [11] and *Linked Data* [12], thereby enabling further reverse engineering and data mining efforts; see the paper's website for pointers.[2]

[1] http://101companies.org
[2] http://softlang.uni-koblenz.de/101meta/

(This is not a UML diagram.) This picture illustrates informally the differences between the constituents of traditional software products and those of software chrestomathies. The model will be further refined in the paper, leading progressively to an accurate UML class diagram in Figure 4 specifying precisely the problem to be solved and a UML deployment diagram in Figure 5. The different kinds of links are drawn in different colors and modes (rectilinear, dotted, oblique) to emphasize their fundamentally different nature.

Legend

Family of *actual* associations (coded knowledge), e.g., calls, imports, and other inter/intra-artifact references.
Family of *expected* associations (documented knowledge), e.g., *subsystemImplemlentsFeature*.
Family of associations to *establish*, e.g., *classPertainsToSubsystem* and *functionImplementsFeature*.

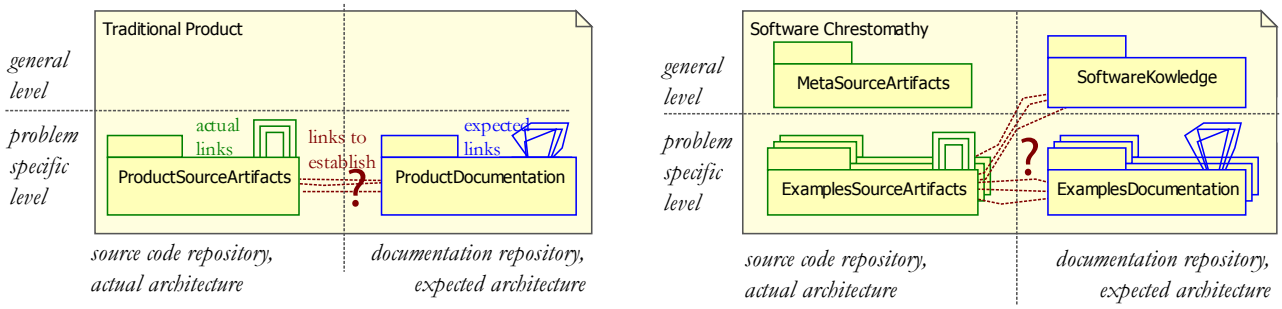

Fig. 1. Linking in the context of traditional software products vs. software chrestomathies.

Non-contributions: We delimit the scope of the paper by highlighting absent forms of validation and absent claims.

NC1. The paper lacks validation for traditional architecture of traditional software products. However, we believe that the paper's approach is not limited though to the linguistic architecture of software chrestomathies.

NC2. The paper lacks validation in terms of corpus size. While many experimental papers focus on a particular technique, and then validate it for a large software corpus, the focus of this paper is on the aforementioned complexity challenges for software chrestomathies.

NC3. The techniques of this paper are not claimed to be exhaustive or definitive. On the contrary, *C3* expresses that a longer community effort is needed to obtain more exhaustive and more definitive results.

Road-map: (The *CacOphoNy* methodology [6] is loosely adopted.) §II compiles an inventory of the *101companies* chrestomathy including initial metamodels for source-code repository and wiki as well as opportunities for links (see Figure 2 for a preview). §III describes the central use case for exploration to be supported by the enriched chrestomathy. §IV specifies more precisely what information to prepare (see Figure 4 for a preview). §V presents language support for metadata assignment. §VI describes the integration of all information in the *101ecosystem* (see Figure 5 for a preview). Related work is discussed in §VII and the paper is concluded in §VIII.

II. *101companies* CHRESTOMATHY – INVENTORY

A. Programming vs. software chrestomathies

Wikipedia defines the notion of chrestomathy as follows:[3]

> *Chrestomathy ([...]; from the Greek words khrestos, useful, and mathein, to know) is a collection of choice literary passages, used especially as an aid in learning a foreign language.*
>
> *In philology or in the study of literature, it is a type of reader or anthology which presents a sequence of example texts, selected to demonstrate the development of language or literary style.*
>
> *In computer programming, a program chrestomathy is a collection of similar programs written in various programming languages, for the purpose of demonstrating differences in syntax, semantics and idioms for each language.*

A *programming chrestomathy* (or program chrestomathy according to Wikipedia) presents solutions to programming tasks (e.g., tasks concerning classic algorithms and data structures) and thereby, it illustrates programming idioms, commonalities and differences of programming languages. The solutions are relatively small programs or excerpts. *RosettaCode*[4] is a good example of a programming chrestomathy; it provides examples in more than 450 programming languages.

We coin the term *software chrestomathy* to refer to a collection of related software products (i.e., possibly small, actually running systems) using various combinations of software languages and technologies, for the purpose of demonstrating languages and technologies as well as software engineering concepts such as architecture, modeling, deployment, documentation, testing, and reverse engineering. *101companies* [4] is designed as a software chrestomathy; it is meant to provide another useful knowledge resource to the software development and engineering community.

B. Separation between documentation and source code

Programming chrestomathies may easily embed source code fragments directly into the documentation repository (such as a

[3] 25 August 2012, 1:01pm (EDT) http://en.wikipedia.org/wiki/Chrestomathy

[4] http://rosettacode.org/

On the left, an informal metamodel of *101repo*, the GitHub-based source-code repository, is presented. On the right, an informal metamodel of *101wiki*, the MediaWiki-based wiki, is presented. The association corresponding to the *links to establish* will be specified in Figure 4. Elements drawn in green and in rectilinear mode correspond to the actual architecture. Elements drawn in blue and in oblique mode correspond to the expected architecture. Elements in red and marked with '?' are links to establish.

Legend

Family of *actual* associations corresponding to links buried in source artifacts.
Individual *actual* UML-style associations corresponding to available facts.
Individual *expected* UML-style associations corresponding, for example, to links on the wiki.
Family of associations to *establish* to be made precise in the next UML diagram.

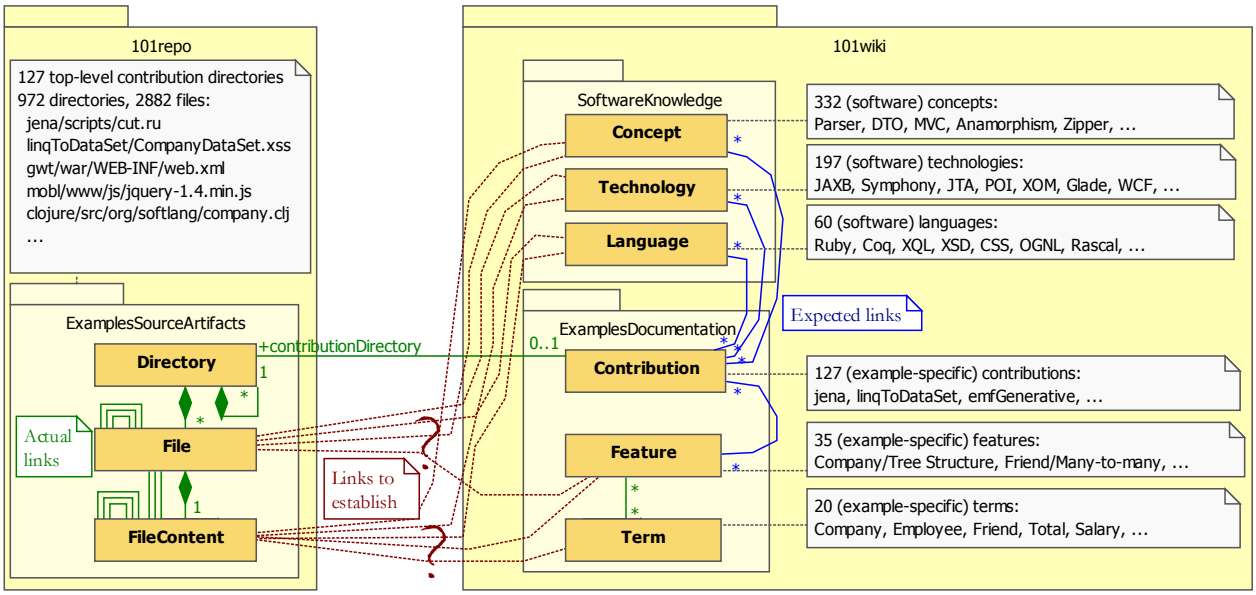

Fig. 2. Informal megamodel of *101repo* and *101wiki* with links.

wiki). In contrast, software chrestomathies have to use a source code repository to organize, persist, and maintain all software artifacts. Thus, an additional documentation repository (which may also be based on a wiki) is used on top. Likewise, source code and some forms of documentation are commonly separated for traditional software products; see Figure 1.

C. Product-specific versus general level

Software chrestomathies differ from traditional software products in an important way that is depicted in Figure 1 by the *general level* with the explicit materialization of *software knowledge* and *meta-source artifacts* that help with managing such knowledge. In the case of *101companies*, software knowledge is represented as wiki pages containing information about software languages, software technologies, and software concepts. Gathering and organizing such knowledge is an integral, non-trivial objective of the chrestomathy. As suggested by the dotted associations on the right of Figure 1, the problem of establishing links is no longer limited to links between source code and its documentation, but links between source code and software knowledge have to be established as well. For instance, one may want to establish that a particular file is valid according to a given language, that a particular file fragment uses a particular technology, or that a particular set of files constitutes an instance of a software concept, such as the MVC pattern. Meta-source artifacts help with establishing such links and with processing the samples of the chrestomathy otherwise. For instance, one kind of meta-source artifact may validate files to pertain to certain languages.

D. Metamodels for the 101companies chrestomathy

In Figure 2, the repositories and constituents of the *101companies* project are described by simplified and informal metamodels. We take the view that the instances of the classes on the left are materialized by actual directories and files in the *101repo* source-code repository, while the instances of the classes on the right are materialized by wiki pages on the *101wiki*. Examples of names of such instances are provided as an illustration. The total number of instances (at the time of writing) is also given for each class.

Consider the top-level directories in the middle of the figure: 'jena', 'linqToDataSet', etc. correspond to different variants of the *101companies* system, called *contributions* (or *implementations*).

Consider the metamodel for the source-code repository on the left of Figure 2. The hierarchical nature of the file system is modeled in a straightforward manner. Files are also decomposed into content (or fragments), which is important if, for example, the expected architecture makes claims at the fragment level.

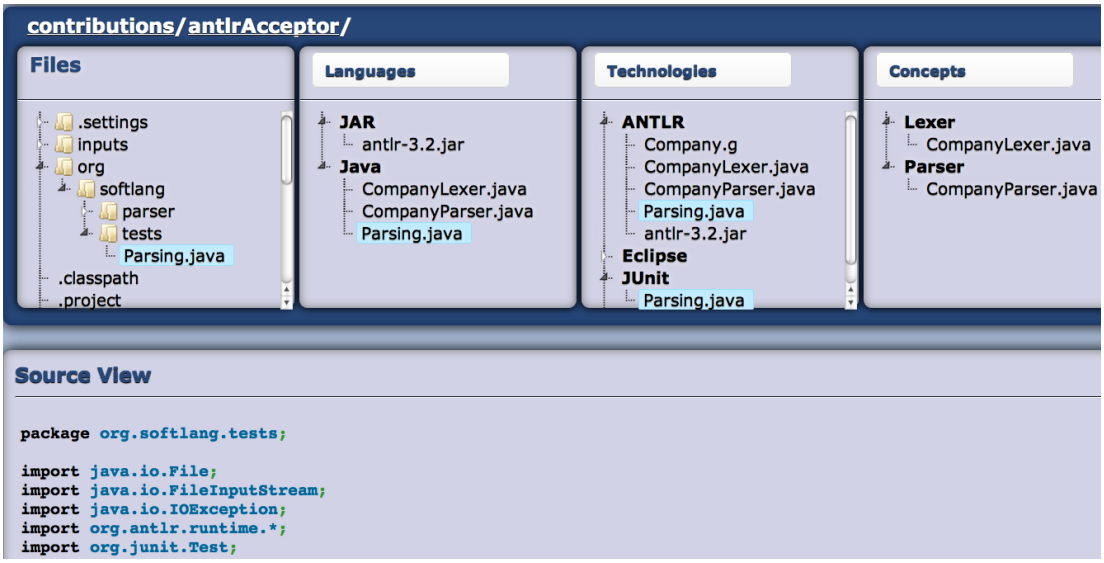

Fig. 3. Exploration of the *101companies* implementation *antlrAcceptor*.

Consider the metamodel for the wiki on the right of Figure 2 rooted in the *Contribution* class whose instances document contributions. Physically, instances correspond to wiki pages decomposed into sections that can be represented in the metamodel by attributes such as 'heading', 'description', or 'issues'. The details of this metamodel are out of the scope of the present paper, but it is important to note that contributors are required to document all software languages, software technologies, and software concepts used in their contributions as well as the implemented product features. This leads to the *expected* links on the right of Figure 2. There is no guarantee, though, that these 'claims' comply with the actual architecture.

E. Opportunities for links in the 101companies chrestomathy

The problem to be solved is therefore to compare the actual architecture buried in *101repo* with the expected architecture documented in *101wiki*. Such reconciliation begins at the roots with the classes *Directory* and *Contribution*. The corresponding links are trivially established because contributions have the same name in *101repo* and *101wiki*. At the component level, one has to dive into the file system hierarchy and possibly into file contents to establish the remaining links. Some links may be established generally on the grounds of rules that capture rules of usage or, at least, best practices for languages and technologies. For instance, there is a rule for files with a file extension '.rb' to imply that the files uses the *Ruby* language. Other links may require rules that are specific to a contribution and possibly software artifacts thereof. For instance, a rule may express that a specific class of a specific contribution associates with the software concept *Parser*.

III. The Exploration Use Case

Before we discuss what information needs to be prepared, we should consider the different stakeholders related to the enriched chrestomathy (developers, teachers, learners, etc.) and their needs. For brevity, we focus on the central use case of *exploration*, which is of interest to all stakeholders. That is, consumers of and contributors to the *101companies* chrestomathy want to navigate contributions in the hierarchical sense of the repository (i.e., directories, files, and fragments) while also observing relevant knowledge about software languages, technologies, and concepts, thereby also taking advantage of additional navigation paths.

Figure 3 shows a snapshot of the *101explorer* tool, which provides designated support for the exploration use case. The figure snapshots some state of exploration for a specific contribution *antlrAcceptor*, which was designed to demonstrate the *ANTLR* parser generator in a *Java* setup. The four panels show the files, languages, technologies, and concepts for the contribution. Each listed language, technology, and concept is also associated with the files that justify the listing. For instance, the file `Company.g` is associated with the technology *ANTLR* because this file is an input of *ANTLR* as hinted at by the '.g' extension. Likewise, some '.java' files are associated with the software concepts *Parser* and *Lexer*.

IV. Specification of the Information of Interest

We need to specify in more detail what information to extract, to assign, and otherwise to prepare in support of the exploration use case specifically; see Figure 4. One can view all such information as *metadata* to be associated with directories, files, and file fragments. Association of metadata may rely on automated, generic rules (e.g., based on matching suffixes of filenames) or it may require contribution- or file-specific rules or declarations.

A. Classification of metadata

Links to establish. The links define, for instance, which languages and technologies are used (in the sense of megamodeling or linguistic architecture [3]), and also which software concepts, features and (domain) terms are involved; see the associations 'elementOf', 'inputOf', and others on the right-hand side of Figure 4.

Attributes to assign. The attributes help with processing source-code artifacts; see in the middle of Figure 4. For

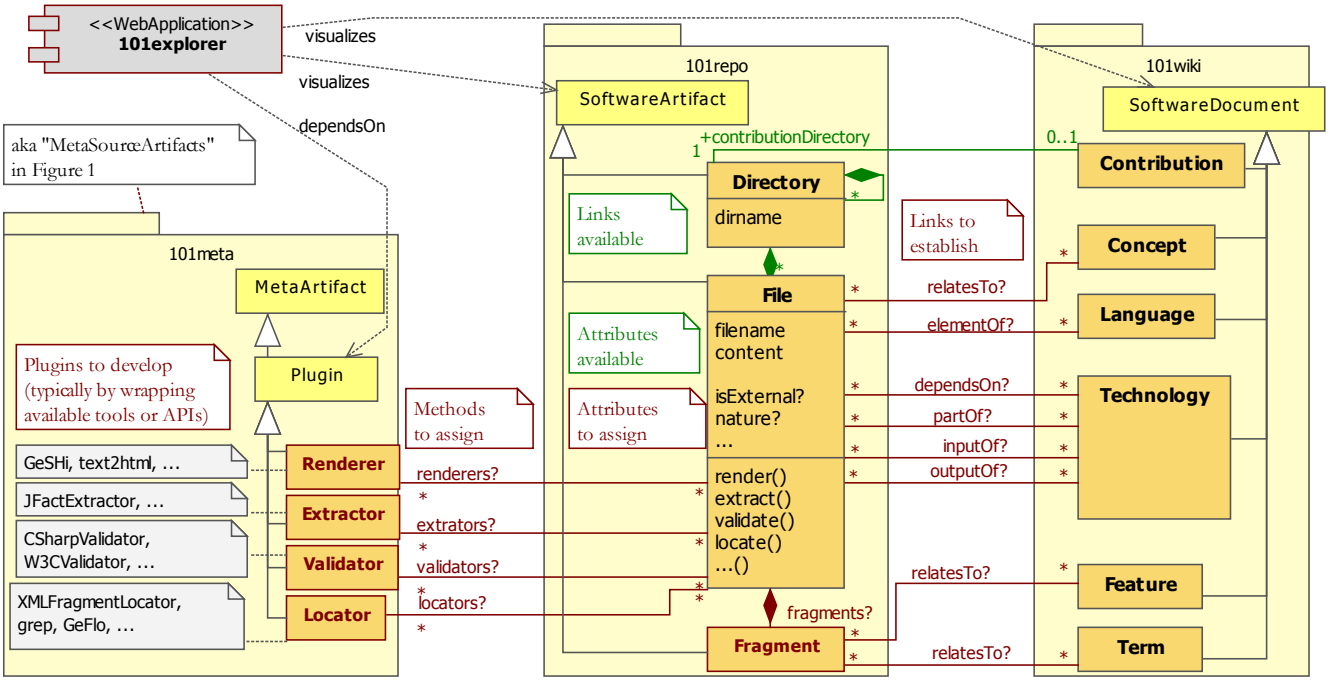

Fig. 4. Information of interest. (Some attributes and associations were omitted for brevity and clarity.)

instance, the 'nature' of each file (e.g., binary, source file, or archive) is used by the *101explorer* to determine whether the content of a file should be displayed or not.

Methods to assign. Files are associated with methods, e.g., for 'rendering' (i.e., producing output that can be presented to users), 'validation' (i.e., establishing that a file is valid according to some language), 'fact extraction', or 'fragment location'. These methods are plugged into the framework as shown on the left-hand side of Figure 4.

B. Additional requirements

In the specific context of the *101companies* project, we add the following requirements for linking documentation and source code in a software chrestomathy:

Generality. The approach must work for most, if not all, software languages and software technologies.

Scalability The approach must support trading off accuracy of fact extraction against development effort for fact extractors. (That is, simple textual or lexical fact extractors should provide enough basic information, while more language-specific and possibly costly syntactical or semantical extractors provide optional information, e.g., for more advanced links and more rigorous checks.)

Declarativeness. The approach should rely on declarative rules, as opposed to any low-level encoding.

Scoping. Rules must be controllable in terms of the specific directory-, file-, or fragment-level scope.

Evolvability. Manual and automated addition and removal of rules should be straightforward and traceable.

Assistance. The analysis of existing metadata should be used in generating recommendations for metadata revisions.

Reuse. Existing language-/technology-aware analysis tools, libraries or web services should be reused.

Openness. All code and data is made available following precepts of Research 2.0, and specifically *Open Science* [11], and *Linked Data* [12].

Simplicity and incrementality. The *101companies* project always favors simplicity and incrementality over sophistication and completeness, thereby catering for community involvement and continuity of the project.

V. Rule-based metadata assignment

Our approach relies on a rule-based system to assign metadata to software artifacts so that links are established and further processing of the artifacts is instructed. The rule-based system is effectively supported by the domain-specific language *101meta*[5]. In the current implementation of *101meta*, rules are represented in JSON, but we use a simple concrete syntax in this paper for clarity's sake. The language is evolving, as new features are requested, but the assumed main elements are presented here, first using a grammar, then through a series of examples.

A. The 101meta language

Rules can be collected in sets. Each rule consists of a condition (on the left-hand side), an optional scope (to deal with fragment-level metadata), and a set of metadata units to be assigned (on the right-hand side). Thus:

RuleSet ::= *Rule*∗ .
Rule ::= *Condition* → [*AssignmentScope*] *Assignment*∗ .
AssignmentScope ::= **fragment** *FragmentDesignator* .

Operationally, rules are to be checked on files (and directories). Whenever a file meets the condition, then the corresponding metadata units are assigned to it. In fact, when

[5]http://101companies.org/index.php/Language:101meta

an fragment designator is specified, then the metadata units are effectively assigned to the specified fragment instead.

Conditions can be formed from Boolean connectors and basic constraint forms:

```
Condition ::= Constraint | ¬ Constraint | Constraint ∧ Constraint | ... .
Constraint ::=
    filename (String | RegExp)
  | basename (String | RegExp)
  | dirname (String | RegExp)
  | suffix String
  | content (String | RegExp)
  | predicate Command .
```

The **filename** constraint allows matching on a given filename either via a simple string or a regular expression. The constraints **basename** and **dirname** correspond respectively to a constraint on the filename without any directory part or on the directory name. The **suffix** constraint is essentially a shorthand for a pattern to constrain only the suffix (typically, the extension) of a filename. The **content** constraint is used to express conditions on the content of files by regular expression matching. Finally, the **predicate** constraint makes it possible to perform arbitrary computations for conditions by applying an executable command to the file under investigation, subject to an interpretation of the exit code. (We think of all such commands as being plugged into the framework.)

In the *101companies* project, so far, the following forms of metadata assignments have been found useful, but the approach is obviously extensible in that new forms of metadata could be added easily:

```
Assignment ::=
    LinkAssignment
  | AttributeAssignment
  | MethodAssignment .
LinkAssignment ::=
    elementOf LanguageName
  | dependsOn TechnologyName
  | partOf TechnologyName
  | inputOf TechnologyName
  | outputOf TechnologyName
  | concept ConceptName // association relatesTo
  | feature FeatureName // association relatesTo
  | term TermName . // association relatesTo
AttributeAssignment ::=
    external // attribute isExternal
  | nature String . // attribute nature
MethodAssignment ::=
    renderer Command
  | extractor Command
  | validator Command
  | locator Command .
```

These forms directly correspond to the specification of §IV and Figure 4 specifically.

B. Language links

Here are a few, very simple examples:

```
suffix ".jar" → nature "archive" .
suffix ".xq" → nature "source" .
suffix ".xq" → elementOf "XQuery" .
```

The rules states that '.jar' files are archives, while files with suffix '.xq' are source files pertaining to the *XQuery* language. Such information can be used by tools such as the *101explorer* or metrics tool as only source files should be rendered in the browser and count in metric calculations.

There is an important conceptual difference between attribute assignment (first two rules) and link assignment (third rule): in the first case 'archive' and 'source' are just scalar values, but 'XQuery' is actually representing a reference to a conceptual entity. Since the class *Language* is the target of the 'elementOf' association (see Figure 4), the string 'XQuery' must constitute a valid language name in the ontology of the *101companies* project. In particular, language names can be completed in URLs on *101wiki*.[6] In this manner, tools like *101explorer* can interpret metadata units for the purpose of navigation. We mention in passing that metadata assignments can be represented as RDF triples, subject to an appropriate megamodeling ontology [3].

C. Technology links

Let us consider also links to technologies as opposed to languages. The parser generator *ANTLR* is used here for illustration. When *ANTLR* is used with *Java*, the technology is packaged as a '.jar' archive. Hence, let us associate, for example, the (version-specific) file 'antlr-3.2.jar' with the technology *ANTLR*.

```
basename "antlr−3.2.jar" partOf "ANTLR" .
```

The **basename** constraint implies that we do not care about the directory of the matched file here. We use **partOf** here in the sense that a software artifact, such as a '.jar' archive, can be considered part of a technology, which is a conceptual (abstract) entity [3]. We may also perform regular expression matching on the **basename** to cover all possible versions of the '.jar' file:

```
basename "#^antlr−.∗\.jar$#" → partOf "ANTLR" .
```

Let us also consider indicators of technology usage. The **suffix** '.g' is an indicator of *ANTLR* usage because this extension is used for grammar files.

```
suffix ".g" → inputOf "ANTLR" .
```

Link assignment expresses here that the matched files serves as input for the parser generator *ANTLR*. The use of *ANTLR* may also be inferred on the grounds of generated files. When *ANTLR* is used in a common manner, then generated code for parser and lexer are to be found in files with specific names as follows:

```
basename "#^.∗(Parser|Lexer)\.java$#" → outputOf "ANTLR" .
```

Actually, the use of 'Parser' or 'Lexer' in filenames does not generally imply usage of *ANTLR*. Thus, we need to further constrain the rule in a way that the content of the files can be checked to support the assumption about *ANTLR* usage. Specifically, looking at files actually generated by *ANTLR*, a simple signature stands out in the first line:

```
// $ANTLR 3.2 Sep 23, 2009 12:02:23 Company.g .
```

[6] *XQuery* maps to http://101companies.org/index.php/Language:XQuery.

This is indeed enough here to help with decision making. We would like to 'grep' the file to search both for the '$ANTLR' string and the distinguished extension '.g' in the same line.

```
basename "#^.*(Parser|Lexer)\\.java$#"
∧ content "#//␣\$ANTLR.*\\.g#"
  → outputOf "ANTLR" .
```

Another kind of evidence for *ANTLR* usage concerns imports of its runtime API 'org.antlr.runtime'.

```
suffix ".java"
∧ content "#^[␣\t]*import[␣\t]*org.antlr.runtime\.#"
  → dependsOn "ANTLR" .
```

Clearly, such import matching could be useful for many other technologies, in fact, APIs. Thus, we may also factor such matching into a more general purpose predicate:

```
#!/bin/sh
# usage: javaImportPredicate.sh <package> <javaFile>
grep -q "^[␣\t]*import[␣\t]*$1\." $2
```

Thus, we revise the rule to invoke the predicate instead of using a content constraint:

```
suffix ".java"
∧ predicate "javaImportPredicate.sh␣org.antlr.runtime"
  → dependsOn "ANTLR" .
```

We may later decide to re-implement the predicate at a syntax-aware level as opposed to regular expression matching.

D. Concept links

We may also want to annotate files with any software concepts of the *101companies* chrestomathy. For instance, we may want to express that certain files define a parser, a GUI, or contribute to a MVC architecture. Here is a concrete example where files of a specific contribution, *antlrObjects*, are tagged with the **concept**s *Parser* and *Lexer*:

```
filename "antlrObjects/org/softlang/parser/CompanyParser.java"
  → concept "Parser" .
filename "antlrObjects/org/softlang/parser/CompanyLexer.java"
  → concept "Lexer" .
```

More general rules may also be conceivable here.

E. Links related to the 101companies domain

We can also assign **feature**s of the *101companies* system as well as **term**s of the domain to files and fragments. For instance:

```
filename "javaStatic/org/softlang/behavior/Total.java" →
    feature "Type-driven␣query"
    term "Total" .
```

F. Method assignments

The preparation of information, as needed for the exploration use case requires several methods, as discussed in §IV. The assigned methods may be invoked by functionality such as the *101explorer*, which operates on the matched file system. There are rules for method assignment for many suffixes and 'special' filenames. For instance:

```
suffix ".wsdl" → renderer "php␣geshiRenderer.php␣xml" .
basename "Makefile" → renderer "php␣geshiRenderer.php␣text" .
```

That is, renderer methods are assigned to '.wsdl' files and makefiles. The Generic Syntax Highlighter, *GeSHi*[7] is leveraged here; this is a PHP package supporting HTML generation for source files for more than 200 languages with awareness of keywords, strings, comments, and others. WSDL files are rendered as XML files; see the 'xml' argument being passed; makefiles are rendered as plain text files; see 'text'.

Eventually, we may also want to use more advanced renderers than *GeSHI*. For instance, we could use a more WSDL-aware renderer which supports navigation for some of the WSDL elements.

In the following example, we register Python programs for validating *Java* source code and extracting facts from the code.

```
suffix ".java" →
  extractor "JFactExtractor.py"
  validator "JValidator.py" .
```

Extractors extract facts from source code, e.g., the declared classes or the imported packages in the case of *Java*. The facts can be used in various ways, e.g., for the purpose of establishing technology-related links. Validators are meant to validate assumptions about files, e.g., to pertain to certain languages. Validators can also be remotely invoked. For instance, '.html' files may be validated by a script which wraps an online service provided by the W3C consortium.

Validation may seem very similar to the predicate form of conditions; see §V-A. However, predicates serve for matching conditions whereas validators serve for the validation of committed matches. Unsuccessful matching (based on predicates) is to be expected; unsuccessful validation pinpoints an issue with software artifacts or rules, and hence, user intervention may be required.

G. Fragment scope

In all examples, so far, we really meant to annotate complete files. In general, it may be necessary to limit the scope of metadata to apply only to file fragments. To this end, a fragment designator has to be used as a scope of assignments. Consider, for example, the data model for companies, as defined by a trivial *Haskell*-based contribution; one file contains all the data types for companies, departments, and employees:

```
module Company where
data Company = Company Name [Department]
data Department = Department Name Manager [SubUnit]
data Employee = Employee Name Address Salary
```

We would like to point to all the specific domain terms 'Company', 'Department', and 'Employee'. The following rule involves fragment designation to link the appropriate **fragment**, i.e., the data type 'Company', to the **term** 'Company':

```
filename "haskell/Company.hs"
  → fragment { "data" : "Company" } term "Company" .
```

We rely on language-aware support for fragment location. In the example, we rely on a *Haskell*-specific locator, which is known due to the following method assignment:

[7] http://qbnz.com/highlighter/

```
suffix ".hs"
  → locator "HsFragmentLocator.py" .
```

There is also a more lexical and generic approach to fragment selection based on GeFLo[8], a *101companies*-specific technology for generic fragment location, which, in turn, is based on GeSHi.

H. Summary of 101meta usage

The paper's website provides access to various derived resources; some of them also directly illustrate and measure the use of *101meta* in the chrestomathy. Here are some illustrative numbers, recorded at the time of writing:

- Examined 2910 repository files.
- Gathered 307 *101meta* rules.
- Assigned metadata to 2030 files.
- Performed 6778 metadata assignments.
- Mapped 41 suffixes (pertaining to languages).
- Mapped 57 *Java* packages (accounting for APIs).

Rule execution for matching and subsequent phases for validation, rendering, etc. are performed continuously on a worker machine of the *101companies* project.

VI. The *101ecosystem*

It remains to integrate the analysis technique of the previous section into a framework that supports the different stakeholders and objectives of a software chrestomathy. The UML diagram of Figure 5 describes the resulting ecosystem of the *101companies* project (see on the left) and examples of external resources, consumers or producers machines (see on the right). The ecosystem is an instance of the notion of metaware environment with meta-usecases according to [6]. Each node (3D box) in the diagram corresponds to a different machine (virtual or not).

The *101ecosystem* consists of three levels corresponding to the flow of data from top to bottom. At the *repository level*, the *101repo* and *101wiki* repositories of the chrestomathy are located. The *101meta* rules reside in *101repo* directories for specific languages, technologies, or contributions.

At the *computation level*, a build server (the so-called *101worker* machine) continuously executes modules to process the repositories so that information is extracted, computed, validated, and prepared for presentation. For instance, the module *match101meta* executes all gathered *101meta* rules and performs the assignment of metadata for *101repo*. The use of a designated build machine like this is a common architectural pattern.

At the *service level*, the results of the computations are served through endpoints of different kinds. For instance, the repository is surfaced in a form that is ready for browsing; results of matching are surfaced in a form that allows for human validation and provides assistance in producing additional rules; the repositories are surfaced in different fact-extraction formats to facilitate query techniques, e.g., based on SPARQL for RDF.

[8]http://101companies.org/index.php/Technology:GeFLo

Let us consider the scenarios (on the right) of the figure; they are based on fictional characters playing different roles.

Chan takes advantage of the knowledge contained in the chrestomathy and uses the *101explorer* to visualize information.

Vadim actively contributes to the project by providing a new implementation of the *101companies* system, demonstrating how the languages and technologies, that he has designed, could be used to solve software development problems.

Ahmed is interested in reverse engineering technologies. He is actively developing his own analysis tools and tests them on the chrestomathy taking advantage of direct access to extracted facts by either downloading dumps in different formats or querying through the SPARQL endpoint. His tools (or parts thereof) could later be reused by the *101ecosystem* for the benefit of the community.

VII. Related work

Derivation of abstractions: Program comprehension and reverse engineering routinely involves the (semi-) automated derivation of models as abstractions over the source code. The Rigi system [13], [14] serves here as a seminal example: the system implements automated techniques to compute and maintain architectural documentation from source code using the Rigi Command Language (RCL).

Our approach mainly aims at establishing links between documented, conceptual entities (i.e., languages, technologies, software concepts, domain terms, and product features) and implemented, physical entities (i.e., files and fragments thereof). Such link establishment can be seen as a sort of source-code summarization in the sense of [15].

Consistency between models and source code: Software reflexion models [1], [2] also help software engineers to establish links between (high-level or design) models and source code. The reflexion approach is concerned with traditional (as opposed to linguistic) architecture and it focuses on summarizing differences and making the models work as a lens on the source code.

Multi-language analyses: Previous research has also aimed at tools and methods that apply to multiple languages without, though, addressing linguistic architecture. [16] describes a fact extractor and an analysis to understand cross-language dependencies in projects using scripting languages. [17] provides a web-based experience for analyzing C, C++, and Java programs. [18] targets multi-language systems by using a common, graph-based conceptual model for the involved languages. [19] focuses on the formalization, management, exploration, and presentation of multi-language program dependencies. Moose [20] serves versatile exploration of source code for all languages with an import path to its FAMIX metamodel.

File extension usage in OSS: [21] examines the use of popular file extensions to analyse language usage along the evolution history of some open-source projects. The approach of this paper enables more diverse analyses of projects, e.g., in terms of different aspects of technology usage, and it assigns

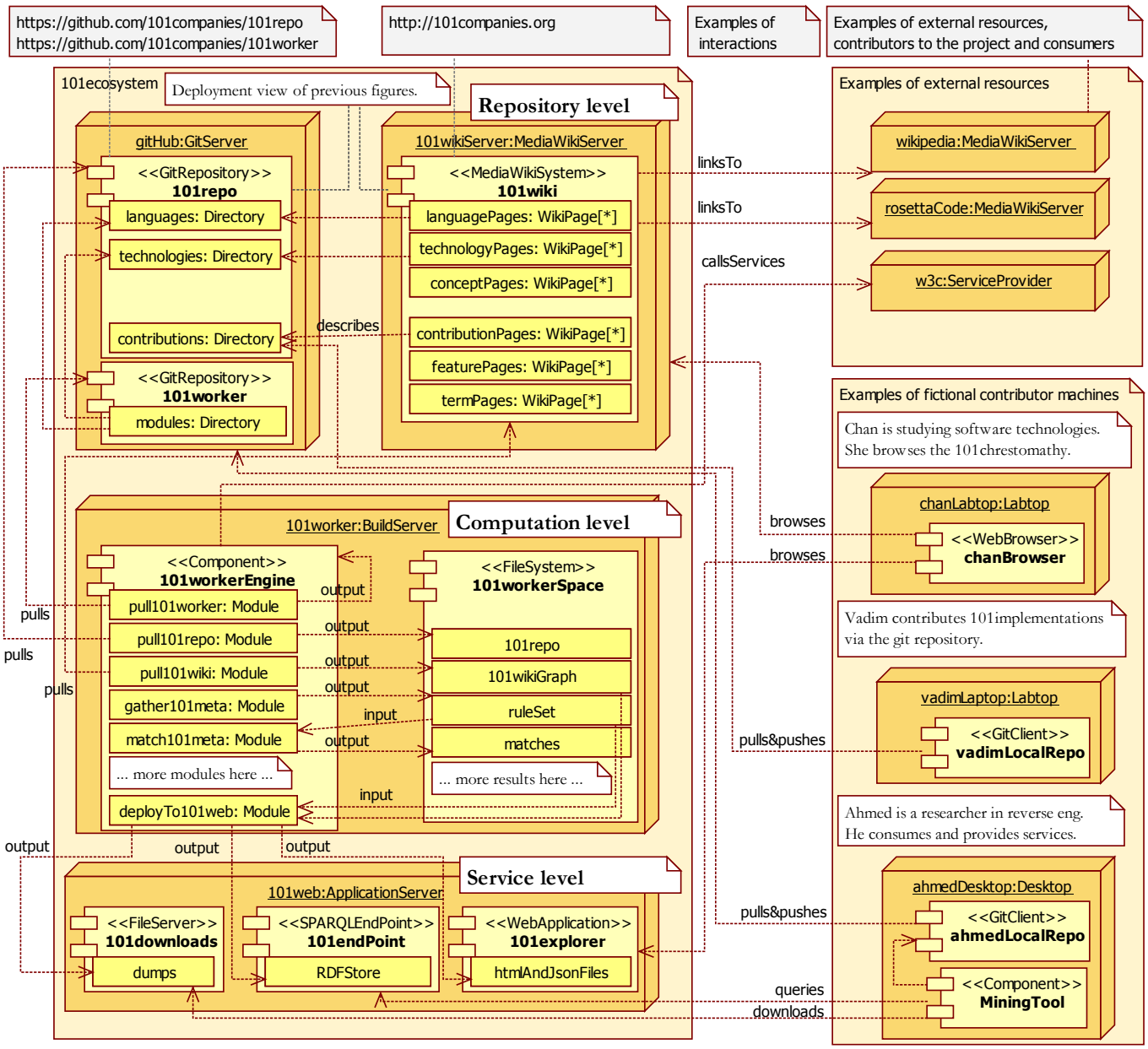

Fig. 5. Architecture of the *101ecosystem* (on the left) with examples for providers, consumers, resources (on the right).

methods to files, thereby supporting functionality such as exploration or fact extraction.

Embedding approaches: Arguably, the link extraction and consistency problem would not exist, if source code and documentation were embedded into the same artifacts either based on *literate programming* [22] (such that source-code fragments are embedded into the program documentation, subject to extraction for compilation etc.) or the opposite approach of *documentation embedding* (such that documentation fragments are embedded into the source-code artifacts, subject to extraction for building the documentation). The latter approach is used by popular technologies such as Javadoc[9] and Doxygen[10].

These two embedding approaches are essentially program-centric in the sense that the integrated artifacts are aligned with program structure. A multi-language, multi-technology software product involves additional abstractions, e.g., ontologies of software development, domain glossaries, feature models, and functional and non-functional requirements. Embedding all such information in a single kind of artifact appears to be unpractical in terms of both logistics (as different stakeholders are involved) and concise representation (as n:m relationships would imply duplication during embedding). In fact, some

[9]http://java.sun.com/j2se/javadoc/
[10]http://www.doxygen.org/

embedding approaches also integrate the notion of link (see `@see X` in Javadoc). Our work shows how to deal with a highly heterogeneous situation in terms of kinds of artifacts, languages, technologies, concepts, and kinds of links.

Metadata assignment: Metadata, specifically in the sense of annotations, is often used to support program comprehension and software maintenance [23], [24] potentially even enabling sharing among developers. Popular tools such as the TagSEA system[11] or the Eclipse Resource Tagger[12] support tagging for the benefit of navigation and location finding. Our work is inspired by metadata approaches that describe metadata external to the addressed artifacts [25], [26].

VIII. Conclusion

A software chrestomathy involves entities that go beyond 'conservative' source-code and documentation artifacts, i.e., languages, technologies, system features, software concepts, etc. Hence, a chrestomathy should be enriched 'semantically' so that all the artifacts are linked to the relevant concepts and rich exploration is enabled and the linked chrestomathy is amenable to some limited consistency checks. We have described a corresponding approach for enriching the *101companies* chrestomathy; it relies on the rule-based language *101meta* for metadata assignment. Also, the approach enriches the repositories of the chrestomathy with derived information resources and services as they are of interest for consumers of and contributors to the chrestomathy, culminating in the *101ecosystem*.

We conclude with a list of future work topics. Authoring of rules shall be supported by interactive tools that make it easy to fill in parts of the rule based on the context of exploration. The community process towards improving coverage of the rules and their actual validation shall be simplified and contributors shall receive credit more clearly. Metadata assignment shall be further automated, e.g., by means of data mining for program identifiers. We expect to integrate the *101ecosystem* with other technology portals, e.g., stackoverflow. We will also drive the process of integrating existing reverse engineering techniques into the ecosystem, thereby turning the *101companies* project into a directly useful platform for teaching reverse engineering.

References

[1] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," in *Proceedings of SIGSOFT FSE 1995 (ACM SIGSOFT Symposium on Foundations of Software Engineering)*. ACM, 1995, pp. 18–28.

[2] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Trans. Software Eng.*, vol. 27, pp. 364–380, 2001.

[3] J.-M. Favre, R. Lämmel, and A. Varanovich, "Modeling the linguistic architecture of software products," in *Proceedings of MODELS 2012 (Model Driven Engineering Languages and Systems)*, ser. LNCS. Springer, 2012, 17 pages. To appear.

[4] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich, "101companies: A Community Project on Software Technologies and Software Languages," in *Proceedings of TOOLS 2012 (Int'l Conference on Objects, Models, Components, Patterns)*, ser. LNCS, vol. 7304. Springer, 2012, pp. 58–74.

[11] http://tagsea.sourceforge.net/

[12] http://taggerplugin.sourceforge.net/

[5] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-Driven Software Architecture Reconstruction," in *Proceedings of WICSA 2004 (Working IEEE / IFIP Conference on Software Architecture)*. IEEE, 2004, pp. 122–134.

[6] J.-M. Favre, "CacOphoNy: Metamodel-Driven Architecture Recovery," in *Proceedings of WCRE 2004 (Working Conference on Reverse Engineering)*. IEEE, 2004, pp. 204–213.

[7] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri, "Revealer: A Lexical Pattern Matcher for Architecture Recovery," in *Proceedings of WCRE 2002 (Working Conference on Reverse Engineering)*. IEEE, 2002, pp. 170–.

[8] S. Kang, S. Lee, and D. Lee, "A framework for tool-based software architecture reconstruction," *Int'l Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 2, pp. 283–30, 2009.

[9] C. S. Liam OBrien and C. Verhoef, "Software architecture reconstruction: Practice needs and current approaches," Carnegie Mellon, Tech. Rep., 2002.

[10] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Trans. Software Eng.*, vol. 35, no. 4, pp. 573–591, 2009.

[11] M. Nielsen, *Reinventing Discovery: The New Era of Networked Science*. Princeton University Press, 2011.

[12] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, ser. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[13] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller, "Programmable reverse engineering," *Int'l Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, pp. 501–520, 1994.

[14] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong, "Domain-Retargetable Reverse Engineering," in *Proceedings of ICSM 1993 (Int'l Conference on Software Maintenance)*. IEEE, 1993, pp. 142–151.

[15] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *Proceedings of WCRE 2010 (Working Conference on Reverse Engineering)*. IEEE, 2010, pp. 35–44.

[16] D. L. Moise, K. Wong, H. J. Hoover, and D. Hou, "Reverse Engineering Scripting Language Extensions," in *14th Int'l Conference on Program Comprehension (ICPC 2006). Proceedings*. IEEE, 2006, pp. 295–306.

[17] S. Mancoridis, T. S. Souder, Y.-F. Chen, E. R. Gansner, and J. L. Korn, "REportal: A Web-Based Portal Site for Reverse Engineering," in *Proceedings of WCRE 2001 (Working Conference on Reverse Engineering)*. IEEE, 2001, pp. 221–230.

[18] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, "Program Comprehension in Multi-Language Systems," in *Proceedings of WCRE 1998 (Working Conference on Reverse Engineering)*, 1998, pp. 135–143.

[19] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and Maintenance of Large-Scale Multi-Language Software Applications," in *Proceedings of ICSM 2006 (Int'l Conference on Software Maintenance)*, 2006, pp. 497–500.

[20] O. Nierstrasz and M. Lungu, "Agile software assessment (Invited paper)," in *Proceedings of ICPC 2012 (Int'l Conference on Program Comprehension)*. IEEE, 2012, pp. 3–10.

[21] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proceedings of MSR 2011 (Mining Software Repositories)*. IEEE, 2011, pp. 13–22.

[22] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, pp. 97–111, 1984.

[23] M.-A. D. Storey, L.-T. Cheng, R. I. Bull, and P. C. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *Proceedings of CSCW 2006 (Computer Supported Cooperative Work)*. ACM, 2006, pp. 195–198.

[24] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz, "Enriching Reverse Engineering with Annotations," in *Proceedings of MoDELS 2008 (Model Driven Engineering Languages and Systems)*, ser. LNCS, vol. 5301. Springer, 2008, pp. 660–674.

[25] L. Silva, S. Domingues, and M. T. de Oliveira Valente, "Non-invasive and non-scattered annotations for more robust pointcuts," in *Proceedings of ICSM 2008 (Int'l Conference on Software Maintenance)*. IEEE, 2008, pp. 67–76.

[26] E. Tilevich and M. Song, "Reusable enterprise metadata with pattern-based structural expressions," in *Proceedings of AOSD 2010 (Aspect-Oriented Software Development)*. ACM, 2010, pp. 25–36.