EMF Patterns of Usage on GitHub

Johannes Härtel, Marcel Heinz and Ralf Lämmel

Software Languages Team, http://www.softlang.org/
University of Koblenz-Landau, Universitätsstraße 1, 56072 Koblenz, Germany

Abstract. Mining software repositories is a common activity in software engineering with diverse use cases such as understanding project quality, technology usage, and developer profiles. Such mining activities involve, more often than not, a phase for data extraction from the source code in the repository with recurring tasks such as processing the folder structure (possibly on the timeline), classifying repository artifacts (e.g., in terms of the languages or technologies used), and extracting facts from the artifacts by parsing or otherwise. We describe a new approach for such data extraction; its key pillar is a declarative rule-based language for the uniform, inference-based extraction of facts from the repository (the file system), the artifacts in the repository (their content), and previously extracted facts. All inferred facts are maintained in a triple store. We describe a case study for the purpose of understanding the usage of *EMF*. To this end, we describe an emerging catalog of patterns of using *EMF* in repositories and we detect these patterns on *GitHub*. In our implementation, we use *Apache Jena* for which we provide dedicated language support tailored towards mining software repositories.

1 Introduction

Our long-term research objective is to apply megamodeling [5, 4] to the problem of documenting software-technology usage in software projects. In our previous work [14, 29, 19, 17, 9], we focused on case studies, basic aspects of language support for such megamodeling, some forms of verification of a megamodel to correspond to a proper system abstraction, the axiomatization of the involved megamodeling expressiveness, the methodology for discovering megamodels, and surveying related concepts in the literature. We also use the term (models of) 'linguistic architecture' for such megamodels.

In this paper, we apply a mining- (or reverse engineering-) oriented view on documenting (or modeling) usage of technologies. We aim at extracting (inferring) facts uniformly from a software repository such that these facts classify artifacts in the repository and describe relationships, for example, related to dependencies, conformance, and correspondence. In particular, we aim at detecting patterns of technology usage. This problem is somewhat similar to design-pattern detection [24, 34, 25] and architecture recovery [16, 18, 27, 2].

In the case study of this paper, we are concerned with *EMF*. We aim to better understand how *EMF* is used 'in the wild' in *GitHub* projects. To this end, we also describe an emerging catalog of patterns for *EMF*. There are, for example, patterns dealing with the more or less consistent and complete presence of interrelated artifacts: metamodel versus derived *Java* code versus generator model. In this paper, we do not study the evolution history of projects [40].

Our approach is original in that we use a declarative rule-based language for the uniform, inference-based extraction of facts from the repository (the folder structure), the artifacts in the repository (their content), and previously extracted facts. All inferred facts are maintained in a triple store. To give the reader an idea, consider the following trivial rule drawn from the case study:

```
(?x sl:manifestsAs sl:File) (?x sl:elementOf sl:XML) Extension(?x,"ecore") →
   (?x sl:elementOf sl:Ecore).
```

Listing 1. Sample rule classifying Ecore files.

The body of the rule (i.e., the condition left to '→') quantifies over artifacts `?x` that are files with extension 'ecore' and readily known to be of 'type' (language) *XML*. For each such artifact, the head of the rule (right to '→') states that the artifact is also of 'type' (language) 'Ecore'. Thus, the rule infers triples for artifacts to be classified as metamodels.

Our rule-based approach is declarative and modular, when compared to the common use of problem-specific custom functionality for processing folder structure and file content (e.g., [21, 9]). Our approach leverages an extensible suite of accessor primitives for interacting with standardized formats and structures such as *Java*, *XML*, and the file system (the folder structure) in a uniform manner. The rule-based approach helps in manifesting only the facts that are actually needed, as opposed to operating on complete ASTs or similar structures (as in, e.g., [34, 35, 1, 39]); it is up to the rules and the accessor primitives to selectively extract and infer more facts. Such inference is similar to the event-condition-action paradigm [10].

Summary of the paper's contributions

- We develop a rule-based approach for uniform, inference-based data extraction from source-code repositories. While we leverage existing techniques known in the semantic web context and as specifically supported by *Apache Jena*, we provide dedicated language support on top of — tailored towards mining software repositories.
- We initiate work towards a structured catalog of *EMF* repository patterns. Each pattern captures a particular situation in a repository such as the presence of a certain kind of artifact and a potential or definite symptom of incompleteness or inconsistency (e.g., a missing or an unsynchronized artifact). This catalog calls for future work.
- We design and execute a case study for mining instances of *EMF* repository patterns in projects on *GitHub*. In this large-scale case study, we examine 5759 repositories. In this paper, we limit ourselves to only studying the most recent version of each project, leaving an evolutionary analysis to future work.

Roadmap of the paper Section 2 develops the rule-based approach for data extraction in mining software repositories. Section 3 develops the catalog of *EMF* repository patterns. Section 4 describes the design of the case study for *EMF* and the results. Section 5 discusses threats to validity. Section 6 discusses related work. Section 7 concludes the paper. The rules and the dataset for the case study and the implementation of the rule-based language are available online.[1]

2 Rule-Based Data Extraction from Repositories

In our approach, the result of data extraction is a 'model' — a set of triples as inferred by the successful application of rules. A rule is of the form '$body \rightarrow head.$' where *body* is the condition part and *head* corresponds to the inferred triples. That is, a rule matches the body against the current set of triples and for each resulting match, the head adds new triples to the model. Inference is a monotone process of inferring triples until a fixed point is reached, i.e., no more successful rule applications for new triples are applicable. The rules may use 'primitives', as discussed below, to access the repository (the folder structure and the content of files). The rule-based approach provides full control over materializing just the 'repository content of interest' in the model. In this section, we may occasionally use rules from the case study for illustrative purposes.

2.1 The Triple Model

Fact extraction infers triples or, in fact, labeled edges of a model (a graph). Nodes are URIs (Unified Resource Identifiers) or literals (such as strings). Each triple consists of a *subject* node (a URI), a *predicate* (a URI) and an *object* node (a URI or literal) where the predicate can be viewed as the edge label. In the earlier rule in Listing 1, we use `sl:File` as an object for classifying a repository artifact `?x` in the subject position with `sl:manifestsAs` as the predicate for the form of classification needed, i.e., an artifact to manifest as, for example, a folder or file. `sl` (for 'software language') represents a custom prefix.

Fact extraction starts from a graph with the following triple:

```
repository:/ sl:manifestsAs sl:Folder.
```

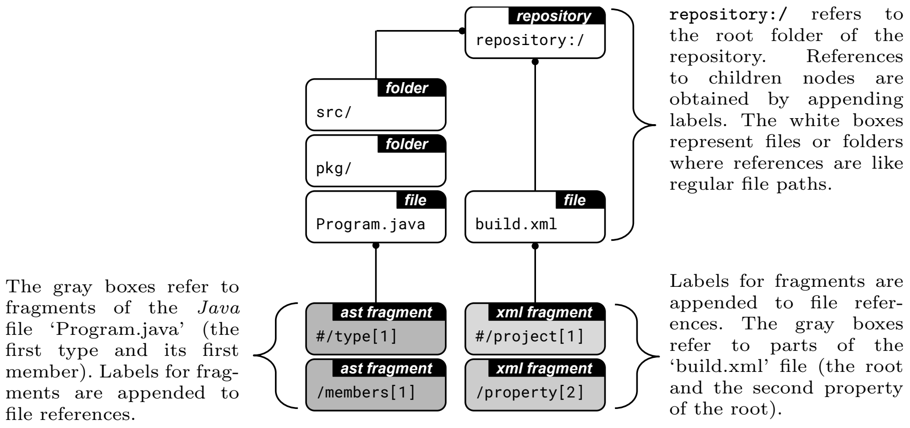
Listing 2. The initial model containing one triple.

The subject URI `repository:/` is special in that it refers to the root of the actual repository folder. Inference discovers folders and files and content thereof, as we discuss below.

2.2 A Scheme for Referencing Repository Content

Figure 1 illustrates the straightforward scheme that we assume for referring to repository content. This is the foundation for treating folders, files, and content particles (fragments of content) for files of different languages in a uniform

[1] https://github.com/softlang/qegal



`repository:/` refers to the root folder of the repository. References to children nodes are obtained by appending labels. The white boxes represent files or folders where references are like regular file paths.

The gray boxes refer to fragments of the *Java* file ‘Program.java’ (the first type and its first member). Labels for fragments are appended to file references.

Labels for fragments are appended to file references. The gray boxes refer to parts of the ‘build.xml’ file (the root and the second property of the root).

Fig. 1. URIs for referring to repository content for a small sample project containing an *ANT* and a *Java* file.

manner. Whether or not all the conceivable nodes are materialized in the model depends on the fact extractor, i.e., on the design of the rules. Typically, we materialize the catalog of the file system completely, but we materialize file content selectively, as we will discuss in more detail in a second.

2.3 Repository Accessors

The rule-based approach relies on primitives for accessing the repository (folder structure and content of files). In particular, the body of a rule may use such primitives to match or bind repository data such as file names and file content in terms of different representations (e.g., ASTs). Primitives may be also used in heads of rules for the purpose of data manipulation or for expanding given bindings into sets of inferred triples.

The primitives needed in the case study are shown in Table 1 with some omissions for brevity. The primitives are free of side effects to the repository. A primitive takes one or more arguments. Each argument is either a URI, a literal, or a placeholder to be bound. The application of a primitive may fail, if the given arguments are not in the corresponding relationship or placeholders cannot be bound to valid results.

2.4 Materialization of the Folder Structure

The following rule is the starting point for decomposing the repository folder:

```
(?x, sl:manifestsAs, sl:Folder) → DecFs(?x, "/*", sl:partOf, ?x).
```

Listing 3. Recursive folder decomposition into parts.

For the initial model (Listing 2), the placeholder `?x` in the body of the rule in Listing 3 is matched with `repository:/`. The head uses the `DecFs(folder, subj, pred, obj)` primitive, as described in Table 1, to decompose file paths,

Table 1. Primitives for accessing folder structure (in general) and file content for *XML* and *Java* (in the case study).

Primitive	**Parameters and Description**
`IsFile`	`(artifact)` Matches if the `artifact` URI can be accessed as a file.
`IsFolder`	`(artifact)` Matches if the `artifact` URI can be accessed as a folder.
`Extension`	`(file, extension)` Matches if the `file` URI has a given extension. (When `extension` is a placeholder, it will be bound.)
`XmlWellformed`	`(file)` Parses the content referred to by the `file` URI and matches if the content is well-formed *XML*.
`Children`	(uri, $\texttt{part}_1$, ..., $\texttt{part}_n$) Decompose a `uri` into its parts split at '/'; parts filled in are matched; variable parts are bound.
`DecFs`	`(folder, xpath, result)` Decompose references to file system by applying an XPath expression `xpath` on the repository starting from the given `folder`; assigns the URI of the first result to `result`.
`DecFs`	`(folder, subj, pred, obj)` A variation on the previous primitive. It infers a *set* of triples, when used in the head of a rule. The inferred triples vary in the subject based on the argument `subj` which corresponds to the XPath expression in the basic form of `DecFs`. The arguments `pred` and `obj` are assigned to regular URIs. For instance, `DecFs(repository:/,"/*", sl:partOf, repository:/)` adds a `sl:partOf` triple for all first-level repository children (XPath '/*' for subject) of the repository (for the object).
`DecJava`, `DecXml`	Variations on `DecFs` working on *Java* ASTs or *XML* trees as opposed to the file system.
`StrXml`, `StrJava`, `UriXml`, `UriJava`	Variations on the decomposition primitives for use in rule bodies, as described above. These variations perform data lookup as opposed to constructing a reference. The `Str...` primitives look up a string (e.g., an attribute in XML) and return it as a result. Likewise, the `Uri...` primitives look up a string which is a URI.

to compose subject URIs, and to infer triples with the `sl:partOf` predicate and `?x` as the object. Listing 4 presents the resulting triples.

```
repository:/ sl:manifestsAs sl:Folder. // Initial repository root classification.
repository:/src/ sl:partOf repository:/. // src is part of the repository.
repository:/build.xml sl:partOf repository:/. // build.xml is part of the repository.
```

Listing 4. Evolved model after applying the rule for folder decomposition.

We should enable the recursive application of the previous rule. To this end, we need to infer triples with the manifestation types `sl:Folder` and `sl:File` of discovered artifacts. The following rules match all (newly added) `sl:partOf` triples, check whether the part is a folder or a file (using the corresponding primitives), and, if so, add suitable triples with the `sl:manifestsAs` predicate.

```
(?x, sl:partOf, ?parent) (?parent, sl:manifestsAs, sl:Folder) IsFolder(?x) →
   (?x, sl:manifestsAs, sl:Folder). // Classifies folders.
(?x, sl:partOf, ?parent) (?parent, sl:manifestsAs, sl:Folder) IsFile(?x) →
   (?x, sl:manifestsAs, sl:File). // Classifies files.
```

Listing 5. Classifying files and folders.

We would like to classify files by languages, as this is a prerequisite for diving deeper into the content, e.g., at the level of parse trees or ASTs. The following rules classify files by a language adding an `sl:elementOf` relationship between the file and the language at hand. We use `(?x sl:manifestsAs sl:File)` to select potential candidates `?x`. Listing 6 illustrates the rules for language classification.

```
(?x sl:manifestsAs sl:File) Extension(?x,"java") → (?x sl:elementOf sl:Java).
(?x sl:manifestsAs sl:File) XmlWellformed(?x) → (?x sl:elementOf sl:XML).
(?x sl:manifestsAs sl:File) (?x sl:elementOf sl:XML) Extension(?x,"ecore") →
   (?x sl:elementOf sl:Ecore).
(?x, sl:manifestsAs, sl:File) Children(?x, _, "META−INF", "MANIFEST.MF") →
   (?x, sl:elementOf, sl:Manifest).
(?x, sl:manifestsAs, sl:File) (?x, sl:elementOf, sl:XML) Children(?x, _, "build.xml") →
   (?x, sl:elementOf, sl:Ant).
(?x, sl:manifestsAs, sl:File) Children(?x, _,"build.gradle") →
   (?x, sl:elementOf, sl:Gradle).
(?x, sl:manifestsAs, sl:File) (?x, sl:elementOf, sl:XML) Children(?x, _, "pom.xml") →
   (?x, sl:elementOf, sl:Pom).
```

Listing 6. Rules for basic language classification.

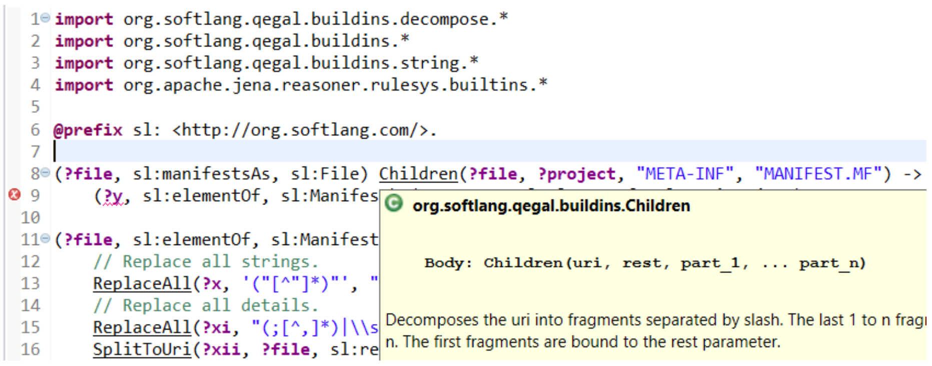
2.5 Pluggable primitives

In our implementation, we rely on the extensibility of the rule engine. That is, *Apache Jena* allows us to plug Java code for new primitives into the engine. In the common semantic web-like use case, primitives are used for basic string or data manipulation. In our mining context, primitives correspond to significant functionality involving repository access, parsing, and more complex analyses. Consider the following implementation of the `Extension` primitive:

```java
public class Extension extends QegalBuiltin {
   @Override
   public boolean trackedBodyCall(Node[] args, int length, RuleContext context) {
      BindingEnvironment env = context.getEnv();
      String file = getArg(0, args, context).getURI();
      Node extension = getArg(1, args, context);
      return env.bind(extension, NodeFactory.createLiteral(iolayer.extension(file)));
   }
}
```

Listing 7. Implementation of the Extension builtin.

This primitive for extension matching or extraction would be typically used in the body of a rule and thus, we need to implement a method `trackedBodyCall` which receives the bound or free arguments `args` and returns true if the primitive completes successfully, i.e., matching or binding succeeds. `Extension(file, extension)` takes two arguments: the `file` argument which must be given and the `extension` argument which is matched if present or bound if it is a placeholder. That is, the method `env.bind(current, expected)` returns true if the `current` and `expected` assignments match or, in the case that `current` is an open placeholder, it binds it to the `expected` value and returns true. For brevity, we omit the discussion of implementing primitives for head usage.



```
import org.softlang.qegal.buildins.decompose.*
import org.softlang.qegal.buildins.*
import org.softlang.qegal.buildins.string.*
import org.apache.jena.reasoner.rulesys.builtins.*

@prefix sl: <http://org.softlang.com/>.

(?file, sl:manifestsAs, sl:File) Children(?file, ?project, "META-INF", "MANIFEST.MF") ->
    (?y, sl:elementOf, sl:Manifes

(?file, sl:elementOf, sl:Manifest
    // Replace all strings.
    ReplaceAll(?x, '("[^"]*)"', "
    // Replace all details.
    ReplaceAll(?xi, "(;[^,]*)|\\s
    SplitToUri(?xii, ?file, sl:re
```

```
org.softlang.qegal.buildins.Children

    Body: Children(uri, rest, part_1, ... part_n)

Decomposes the uri into fragments separated by slash. The last 1 to n fragi
n. The first fragments are bound to the rest parameter.
```

Fig. 2. IDE support for working with rules and primitives.

2.6 Dedicated Language Support

Our implementation leverages the *Apache Jena*[2] implementation for rule-based inference and triple processing. Our experience with the rule-based approach in case studies such as the one of Section 4 led us to advance the Jena approach by adding language support addressing the complexities of mining software repositories. In addition to the specific primitives needed, as discussed earlier, there are these aspects:

IDE support Based on XText, [3][3] provide editing, syntax highlighting, auto-completion, code navigation, and other IDE support (see Fig. 2 for an illustration) also subject to JVM integration for the pluggable primitives of the rule-based language;

Logging and profiling Log the execution of primitives with appropriate context and runtimes to enable debugging of the rule-based system and to check for performance hogs, thereby guiding optimization of primitives and rule set;

Exception handling Recover from and log exceptions thrown by accessor primitives to enable completion of repository processing and post-mortem analysis, subject to a dedicated interface for primitives and housekeeping;

Virtualized access Be able to switch between actual file-system-based access to artifacts (development mode) and immediate repository access without manifestation on the local file system (production mode);

Testing Use a combination of parametrized and instance-based tests on white- and blacklisted repositories and the derived models, also using redundant repository processing (e.g., based on grep) to obtain reasonable baselines.

3 Towards an EMF Repository Pattern Catalog

EMF can be used in different ways in projects, subject to the *presence* of different types of artifacts, possibly with different *multiplicities* and in different

[2] https://jena.apache.org/
[3] http://www.eclipse.org/Xtext/

Table 2. An *EMF* Repository Pattern Catalog (Some of the corresponding detection rules are discussed in Section 4. All of the rules are available online.)

Id	**Cls.**	**Artifacts**	**Description and cause**
Single artifact patterns			
E	Pres.	– Ecore Pkg.	The presence of an Ecore Pkg. in '.ecore' files as root or subpackage.
J	Pres.	– Java Pkg.	The presence of a Java Pkg.
G	Pres.	– Genmodel Pkg.	The presence of a Genmodel Pkg. in '.genmodel' files as root or subpackage.
C	Pres.	– Customized Java Pkg.	The presence of a Java Pkg. with customized interface or implementation.
Double artifact patterns			
EJ1	Pot. Incomp.	– Ecore Pkg. – Java Pkg. (m[a]) [a] Missing	A Java Pkg. cannot be found for a given nsURI as extracted from some Ecore Pkg. This is only a potential incompleteness, because a Java Pkg. could be potentially derived, if no customization is intended.
EJ2	Def. Incomp.	– Ecore Pkg. (m) – Java Pkg.	An Ecore Pkg. cannot be found for a given nsURI as extracted from some Java Pkg. This is a definite incompleteness because the Java Pkg. is derived and thus, the underlying primary artifact (the Ecore Pkg.) should also be in the repository.
EJ3	Pres.	– Ecore Pkg. – Java Pkg.	The presence of a Java Pkg. and Ecore Pkg. with the same nsURI. One Ecore Pkg. can correspond to many Java Packages.
EE	Def. Incons.	– Ecore Pkg. – Ecore Pkg.	An Ecore Pkg. with at least one competing Pkg. with the same nsURI.
EJc1	Def. Incons.	– Ecore Pkg. – Java Pkg.	A Java class that is part of the Java Pkg. with a corresponding Ecore Pkg., but without a corresponding Ecore classifier (based on name comparison). For instance, one may have forgotten to remove a Java class derived from an earlier version of the metamodel.
EJc2	Def. Incons.	– Ecore Pkg. – Java Pkg.	An Ecore classifier contained in an Ecore Pkg. with a corresponding Java Pkg., but without a corresponding Java classifier (based on name comparison). The Java Pkg. is thus out of sync with the Ecore Pkg. in the repository.
Triple artifact patterns			
EJJ	Pres.	– Ecore Pkg. – Java Pkg. – Java Pkg.	An Ecore Pkg. with at least two corresponding Java Packages.
EJG	Pot. Incomp.	– Ecore Pkg. – Java Pkg. – Generator Pkg. (m)	For a corresponding pair of Java Pkg. and Ecore Pkg., a corresponding Generator Pkg. cannot be found.

combinations. In this paper, we begin work towards a catalog for *EMF* which covers these basic 'artifact' types: *Ecore Package*, as identified in '.ecore' files where one such file can possibly define several such packages; *Java Package* – an actual Java package containing derived classes according to a metamodel, a factory, and a package description; and *Generator Package* as identified in '.genmodel' files.

Artifacts of these types can be related in certain ways in a project. In fact, by checking on certain relationships, e.g., by determining the *absence* of certain artifacts or elements thereof, we may infer cases of *incompleteness* or *inconsistency*, where these are either *potential* or *definite* problems of usage or, in fact, of maintaining *EMF* usage in the repository. Table 2 lists patterns organized along these different dimensions (artifact type, presence, incompleteness, inconsistency, potential versus definite). We group by cardinalities of artifacts: single, double, and triple artifact patterns. For brevity, we exclude patterns related to XMI-based persistence in this paper. Generally, further work is needed to arrive at a more comprehensive catalog for *EMF*.

4 An MSR Case Study on EMF

We located projects with traces of *EMF* usage on *GitHub*. We assessed these projects in terms of some basic architectural characteristics to prepare a selection of well-understood project layouts for which a mining process is assumed to provide more comprehensible insights. Eventually, we detected *EMF* repository patterns as introduced in Section 3. We describe these phases here and summarize our findings.

4.1 Locating Repositories

We used the *GitHub* search API to locate all recently indexed *Ecore*, *Generator Model* and *Java Model* files on *GitHub* as an indication of *EMF* usage in repositories. The corresponding queries are listed in Table 3.

Table 3. Queries for locating repositories through *GitHub* API.

Evidence	**Query**	**Extension**
Java Model	"extends EObject {"	java
Ecore Model	GenModel	ecore
Generator Model	EClass	genmodel

(For what it matters, the API search limit is circumvented by recursive query segmentation which splits a query by setting an upper and lower bound in file size based on the returned number of total results. This process may miss some results.) The search API only considers heads of default branches and files smaller than 384 KB. A list of 5759 *GitHub* repositories was extracted from the query results.

4.2 Selection of Repositories

We applied the rule-based mining approach to recover the repository layout in terms of usage of build systems, project dependencies, and other aspects. We developed the following classifiers for repositories as an extension of the pattern catalog of Section 3:

Homogeneous versus **heterogeneous build system** We search for traces of Manifest, POM, Gradle, and ANT, as modeled by the rules in Listing 6. In the homogeneous case, only one such technology is used; otherwise we apply the heterogeneous classifier. We assume that the heterogeneous situation is harder to understand in terms of project dependencies.

Single component versus **multiple components** Based on an analysis of project dependencies, as described in more detail below, we determine the number of components. We assume that repositories with multiple components are special. Such a repository may be, for example, a 'zoo' [28]. Note that a single component can still imply the presence of multiple (dependent) projects.

Variants This classifier applies when we locate different versions of the same project in a repository based on the analysis of project dependencies. We assume again that repositories with variants are special. Such a repository may capture, for example, versions in a migration process.

EMF's default is the use of Manifest files for defining OSGi projects and dependencies. We decided to only include homogeneous repositories using Manifest files for mining. The analysis of dependencies is based on 'Bundle-SymbolicName' elements in Manifest files. Listing 8 presents the rules for inferring the occurrence of declarations (predicate `sl:decOccures`) and references (predicate `sl:refOccures`) and OSGi dependencies between Manifest files (predicate `sl:dependsOn`):

```
// Extraction of Bundle−SymbolicName declaration.
(?file, sl:elementOf, sl:Manifest) StrManifest(?file, "Bundle−SymbolicName", ?x)
    ReplaceAllToUri(?x, "(;[^,]∗)|\\s", "", ?declaration) → // Replace all details.
    (?file, sl:decOccurs, ?declaration).
// Extraction of Bundle−SymbolicName references.
(?file, sl:elementOf, sl:Manifest) StrManifest(?file, "Require−Bundle", ?x)
    ReplaceAll(?x, '("[^"]∗)"', "", ?xi) // Replace all strings.
    ReplaceAll(?xi, "(;[^,]∗)|\\s", "", ?references) → // Replace all details.
    SplitToUri(?references, ?file, sl:refOccurs, ',').
// Creating dependency structure
(?a, sl:elementOf, sl:Manifest) (?b, sl:elementOf, sl:Manifest)
    (?a, sl:decOccurs, ?deca) (?a, sl:refOccurs, ?decb) (?b, sl:decOccurs, ?decb) →
    (?deca, sl:dependsOn, ?decb).
```

Listing 8. Rules for extracting OSGi declarations, references and dependencies.

The primitive `StrManifest(file, property, value)` is a specialized decomposition of a Manifest file, comparable to `StrJava` in Section 2.3; it binds `value` to a Manifest property in literal form. In the rules shown above, it binds `?x` to the required or defined bundles in string representation. The chains of `ReplaceAllToUri`, `ReplaceAll` and `SplitToUri` process `?x` in that it can be added to the model as declaration or reference. We exclude repositories with duplicated declarations (`sl:decOccurs`) for the same URI (classifier *Variants*). We apply an algorithm for detecting connected components to the `sl:dependsOn` triples, as inferred by the last rule shown above. We exclude repositories with multiple components.

The results of the selection steps are depicted in Fig. 3. In what follows, we only consider repositories with a single component, Manifest usage only, and no variants. We refer to these repositories as 'Vanilla *EMF* repositories'. There are 1438 such projects. These are the projects considered for mining below.

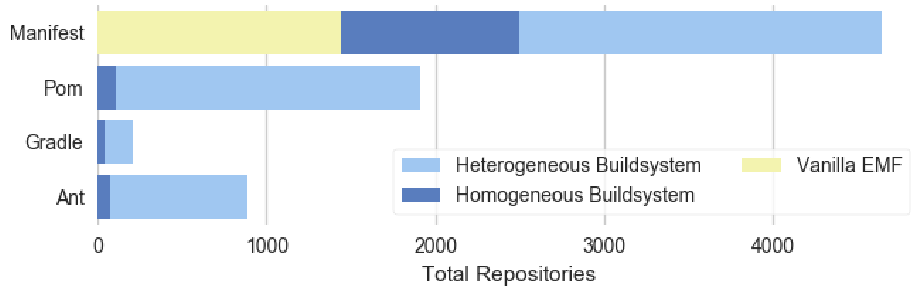


Fig. 3. Number of repositories with a particular build system further partitioned into homogeneous versus heterogeneous case.

4.3 Pattern Detection

For brevity, we only discuss here the detection of correspondence between Java and Ecore models; we show the decomposition of an Ecore model; we omit the rules for *Java* model detection in the set of available *Java* files; we also omit handling Ecore sub-packages.

Consider the beginning of a small Ecore sample file as follows:

```
<ecore:EPackage ... name="fsml" nsURI="http://www.softlang.org/metalib/emf/Fsml"
      nsPrefix="fsml">
   <eClassifiers xsi:type="ecore:EClass" name="FSM">
         ...
```

Listing 9. The first lines of a sample Ecore file.

The following rules decompose an Ecore model into root package and nested classifiers:

```
// Decomposition of the Ecore file into ...
(?x, sl:elementOf, sl:Ecore) → // ... the root package.
   DecXml(?x, "/ecore:EPackage", sl:partOf,?x)
   DecXml(?x, "/ecore:EPackage", sl:elementOf, sl:EcorePackageXMI).
(?x, sl:elementOf, sl:EcorePackageXMI) → // ... the nested classifiers in a package.
   DecXml(?x, "/eClassifiers", sl:partOf, ?x)
   DecXml(?x, "/eClassifiers", sl:elementOf, sl:EcoreClassifierXMI).
// Extracting URI and nsURI, necessary for detecting correspondence.
(?x, sl:elementOf, sl:EcorePackageXMI) →
   UriXml(?x, ?x, sl:nsUri, "/@nsURI"). // NsUri for a package as URI.
(?classifier, sl:elementOf, sl:EcoreClassifierXMI)
   (?classifier, sl:partOf, ?package) (?package, sl:nsUri, ?nsUri) // Get package's nsURI.
   StrXml(?classifier, "/@name", ?classifierName) // Get the classifier's name as string.
   UriConcat(?nsUri, '#//', ?classifierName, ?uri) → // Build a compound ?uri.
   (?classifier, sl:uri, ?uri). // Uri for a classifier, i.e., nsURI with appended name.
```

Listing 10. Decomposing Ecore into classifiers appending a nsURI.

That is, a `sl:partOf` relationship is inserted along the nesting structure and fragments are classified by `sl:EcorePackageXMI` and `sl:EcoreClassifierXMI`

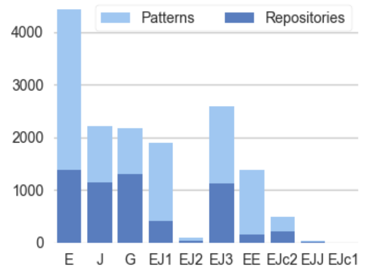


Fig. 4. Overall pattern sum and the number of repositories a pattern occurs in.

	E	J	G	EJ1	EJ2	EJ3	EE	EJc2	EJJ
Sum	4427	2217	2181	1894	96	2598	1376	496	45
Repo	1389	1152	1294	404	43	1127	157	223	18
mean	3.1	1.5	1.5	1.3	0.1	1.8	1.0	0.3	0.0
std	9.8	3.2	2.2	6.2	0.5	7.3	8.3	1.6	0.5
25%	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
50%	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
75%	2.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0
max	261	71	28	103	10	248	251	26	16
cv	3.2	2.1	1.5	4.7	8.2	4.0	8.6	4.5	15.2

Fig. 5. The distribution of detecting a pattern in the repositories (where the minimum is always 0.0). Row 'cv' lists the coefficient of variation.

respectively. This decomposition is handled by the first two rules using the DecXml to construct URIs in the repository referencing scheme. In contrast, the last two rules extract the attributes 'nsURI' and 'name' using UriXml and StrXml. The primitives return the recovered content directly as string or URI, as we need the actual attribute values 'FSM' (name) and 'http://www.softlang.org/metalib/emf/Fsml' (nsURI).

The following rules establish the correspondence between the elements of `sl:EcorePackageXMI` and `sl:EcorePackageJava` by matching the nsURI.

```
// Correspondence between XMI and Java Packages.
(?xmi, sl:elementOf, sl:EcorePackageXMI) (?java, sl:elementOf, sl:EcorePackageJava)
   (?xmi, sl:nsUri, ?nsUri) (?java, sl:nsUri, ?nsUri) →
   (?xmi, sl:correspondsTo, ?java).
// Correspondence between XMI and Java Classifiers.
(?xmiClassifier, sl:uri, ?uri) (?javaClassifier, sl:uri, ?uri)
   (?xmiClassifier, sl:elementOf, sl:EcoreClassifierXMI)
   (?javaClassifier, sl:elementOf, sl:EcoreClassifierJava) →
   (?xmiClassifier, sl:correspondsTo, ?javaClassifier).
```

Listing 11. Rules recovering the correspondence.

The *Java* model extraction underlying the latter classification is based on accessing the nsURI property in the *Java* AST by a similar primitive UriJava (not shown here). Corresponding classifiers are aligned by comparing the nsURI concatenated with the classifier name.

4.4 Results

The results of the case study applied on 1438 Vanilla *EMF* repositories are shown in Fig. 4, Fig. 5, and Fig. 6. The discussion is not comprehensive. Overall, the online corpus features additional results. At the most basic level, we show numbers of repositories per pattern and numbers of pattern instances (Fig. 4). The median pattern occurrence of Ecore (E), Java (J) and Genmodel (G) packages and the regular correspondence (EJ3) in a repository is 1 (Fig. 5). This indicates that common usage is concerned with only one package. The coefficient of

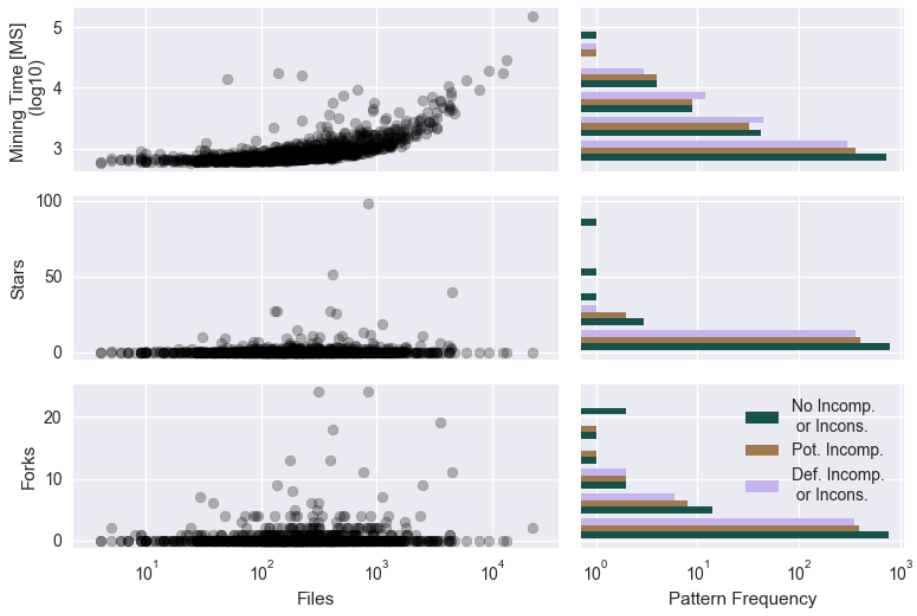


Fig. 6. Different repository and mining characteristics shown with respect to the amount of files in a repository. The right column shows how often different types of patterns occur for some histogram buckets and for the different characteristics.

variation for measuring the relative variability 'cv' is the highest for EJ2, EE and EJJ — the first two patterns indicate problems; the latter pattern represents a very rare case (1% of the Vanilla repositories). We expect corner cases and problems to have high variations in pattern occurrence. Having no package correspondence (EJ1) or a regular package correspondence (EJ3) can both be considered as common usage. Forgetting to remove Java classifiers (EJc2) can also be considered a common usage despite being a definite inconsistency. On the contrary, we detected no repository missing a Java classifier for an Ecore classifier (EJc1). In Fig. 6, we examine the projects on scales for different characteristics (forks, stars, and mining time) to see how the size of projects relates to these characteristics and also how the relative frequency of pattern-based problems or the absence thereof relates to these characteristics. For instance, we can observe (right-bottom and right-middle charts in Fig. 6) that definite incompleteness and inconsistency is of much less or no concern with increasingly more forked or stared repositories.

5 Threats to Validity

The initialization and filtering of the repository list towards 'Vanilla EMF' can be seen as a threat to external validity. We might miss 'important' repositories and thereby produce biased results. Further, due to the complexity of EMF and the diversity of possible repository layouts, we might potentially miss particular cases in the rules. We extensively tested our rules in an instance- and parameter-based manner to cover this internal threat, but some rules are incomplete (e.g., regarding the considered build systems) or approximative (e.g., in assuming a very strict naming scheme for generated classifiers). Our pattern catalog and the rating of patterns, i.e., *potential* or *definite incompleteness* or *inconsistency*, are potentially subjective, even though we have extensive experience with *EMF*.

6 Related Work

The reported research is original in terms of (i) leveraging an inference- and rule-based approach (as opposed to using any computational approach which would operate on more 'complete' representations) and (ii) developing a pattern catalog for EMF usage in repositories. However, there is related research in the broader areas of mining software repositories, program comprehension, and reverse engineering which we group accordingly below.

Pattern Detection This may concern design or API-usage patterns. For instance, in [36], API-usage patterns are mined based on structural, semantic, and co-usage similarity for the accessed API methods and fields; in [34], logic metaprogramming is used to detect patterns in a logic layer on top of Java ASTs using JDT.

Classification of Artifacts In [21], language usage trends are analyzed in 22 open source software projects by counting files with language-specific file extensions, e.g., '.py', and certain file-name patterns, e.g., 'README'. In [26], the use of different Eclipse-based MDE technologies is examined on *GitHub* repositories. The approach combines search based on technology-specific extensions and string-based content search, just like in our case study (Table 3). In [33], a large dataset of UML models is collected from *GitHub* by script-based inspection of downloaded images (e.g., '.jpg'), standard UML formats (e.g., '.uml') and tool specific formats (e.g., '.argo'). In [32], the number of languages used and their relatedness to each other is analyzed in a random set of 1150 *GitHub* repositories relying only on metadata and version history.

Linking Artifacts In [41], traces between XML documents are analyzed using an imperative rule language and XML technology such as XPath. This work exceeds ours by considering references encoded in text fragments that are then mined using NLP-techniques. In [8], the distribution frequency at *GitHub* is empirically compared to CRAN for R packages. Further, package dependencies between projects' 'DESCRIPTION' files are mined to explore inter-repository dependency problems. In [30], a system's ground truth architecture is recovered by analyzing dependencies in the build configuration. To raise accuracy, the folder layout is considered. Arguably, our work relates to traceability recovery [7, 22, 15, 31, 38].

Fact extraction In [20], metrics are computed in multi-language repositories by reusing existing parser technology that is part of Eclipse. In [37], API usage in Java-based *GitHub* repositories is analyzed by parsing the code and resolving method calls to APIs. Their approach exceeds ours in using JDT type resolution which we may want to incorporate into our rule-based language model. In [1], OO-specific ASTs (i.e., Java) are converted into RDF triples. This enables data extraction through SPARQL-queries on a triple store. In [39], a rule-based approach similar to ours is presented that relies on parsing code into a knowledge-discovery-model (KDM). It is used to mine dependencies in Java EE application. While we also employ AST structures, we rely on selective fact extraction as opposed to full materialization of the involved artifacts. In [11–13], the infrastructure, the domain specific language and applications of Boa, a framework for structured data extraction targeting repositories, is described. While the actual purpose and the surrounding infrastructure of our approach is highly comparable, the computation substantially differs in that Boa's fact extraction is not driven by previously inferred facts. The Boa language is compiled to a map-reduce framework to be executed in parallel. Such distribution is difficult to align with our rule-based inference mechanism.

Analysis of Changes In [6], the frequency of code changes in Java-projects using Hibernate is traced while differentiating between data model, mapping, performance configurations, and query calls. In [23], the dependencies between projects considering build files specific to JavaScript, Ruby, and Rust are examined and their evolution is traced. In our future work, we will take version history into account.

7 Conclusion

In this paper, we have provided some insight into basic variation points and potential completeness and consistency problems with using EMF and detecting or maintaining such usage in repositories. We have used a rule-based approach to detect patterns of usage.

In future work, we would like to analyze evolution of repositories in terms of layout and pattern instances. Further, we would like to increase precision with regard to some aspects of correspondence, completeness, and consistency by integrating type resolution with Java classpath recovery. Moreover, we also work on a more profound generalization of referring to and accessing 'arbitrary' code or model elements across technological spaces: codename URA (unified resource accessor). Ultimately, we would like to move from (small) patterns of technology usage to inferring usage of more complex and modular megamodels for technology documentation [17].

References

1. M. Atzeni and M. Atzori. CodeOntology: RDF-ization of source code. In *Proc. ISWC*, 2017.
2. B. J. Berger, K. Sohr, and R. Koschke. Extracting and analyzing the implemented security architecture of business applications. In *Proc. CSMR*, pages 285–294. IEEE, 2013.

3. L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.
4. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Proc. MDAFA 2003 and MDAFA 2004*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.
5. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proc. OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.
6. T. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. N. Nasser, and P. Flora. An empirical study on the practice of maintaining object-relational mapping code in Java systems. In *Proc. MSR 2016*, pages 165–176, 2016.
7. J. Cleland-Huang, O. Gotel, and A. Zisman, editors. *Software and Systems Traceability*. Springer, 2012.
8. A. Decan, T. Mens, M. Claes, and P. Grosjean. When GitHub meets CRAN: an analysis of inter-repository package dependency problems. In *SANER*, pages 493–504, 2016.
9. J. Di Rocco, D. Di Ruscio, J. Härtel, L. Iovino, R. Lämmel, and A. Pierantonio. Systematic recovery of MDE technology usage. LNCS. Springer, 2018.
10. K. R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of ADBMS features. In *Proc. RIDS*, volume 985 of *LNCS*, pages 3–20. Springer, 1995.
11. R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE Computer Society, 2013.
12. R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, 2015.
13. R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *ICSE*, pages 779–790. ACM, 2014.
14. J. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.
15. I. Galvão and A. Goknil. Survey of traceability approaches in model-driven engineering. In *Proc. EDOC*, pages 313–326. IEEE, 2007.
16. M. Han, C. Hofmeister, and R. L. Nord. Reconstructing software architecture for J2EE web applications. In *Proc. WCRE*, pages 67–79. IEEE, 2003.
17. J. Härtel, L. Härtel, R. Lämmel, A. Varanovich, and M. Heinz. Interconnected linguistic architecture. *Programming Journal*, 1(1):3, 2017.
18. A. E. Hassan, Z. M. Jiang, and R. C. Holt. Source versus object code extraction for recovering software architecture. In *Proc. WCRE*, pages 67–76. IEEE, 2005.
19. M. Heinz, R. Lämmel, and A. Varanovich. Axioms of linguistic architecture. In *Proc. MODELSWARD 2017*, 2017.
20. A. Janes, D. Piatov, A. Sillitti, and G. Succi. How to calculate software metrics for multiple languages using open source parsers. In *Proc. OSS*, pages 264–270, 2013.
21. S. Karus and H. C. Gall. A study of language usage evolution in open source software. In *MSR*, pages 13–22. ACM, 2011.
22. E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. I. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian,

S. Hossein, and D. Hearn. TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proc. ICSE*, pages 1375–1378. IEEE, 2012.
23. R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In *MSR*, pages 102–112, 2017.
24. G. Kniesel and A. Binun. Standing on the shoulders of giants - A data fusion approach to design pattern detection. In *Proc. ICPC*, pages 208–217. IEEE, 2009.
25. G. Kniesel, A. Binun, P. Hegedüs, L. J. Fülöp, A. Chatzigeorgiou, Y. Guéhéneuc, and N. Tsantalis. DPDX–Towards a common result exchange format for design pattern detection tools. In *Proc. CSMR*, pages 232–235. IEEE, 2010.
26. D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige. Assessing the use of Eclipse MDE technologies in open-source software projects. In *Proc. MODELS*, pages 20–29, 2015.
27. R. Koschke. Architecture reconstruction. In *ISSSE 2006-2008, Revised Tutorial Lectures*, volume 5413 of *LNCS*, pages 140–173. Springer, 2009.
28. A. Kusel, J. Schoenboeck, M. Wimmer, W. Retschitzegger, W. Schwinger, and G. Kappel. Reality check for model transformation reuse: The ATL transformation zoo case study. In *Proc. AMT 2013*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.
29. R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.
30. T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proc. ICSE*, pages 69–78, 2015.
31. P. Mäder and A. Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empir. Softw. Eng.*, 20(2):413–441, 2015.
32. P. Mayer and A. Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proc. EASE*, pages 4:1–4:10, 2015.
33. G. Robles, T. Ho-Quang, R. Hebig, M. R. V. Chaudron, and M. A. Fernández. An extensive dataset of UML models in GitHub. In *Proc. MSR*, pages 519–522, 2017.
34. C. D. Roover. A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In *Proc. ICSM*, pages 556–561. IEEE, 2011.
35. C. D. Roover, R. Lämmel, and E. Pek. Multi-dimensional exploration of API usage. In *Proc. ICPC 2013*, pages 152–161. IEEE, 2013.
36. M. A. Saied and H. A. Sahraoui. A cooperative approach for combining client-based and library-based API usage pattern mining. In *Proc. ICPC*, pages 1–10, 2016.
37. A. A. Sawant and A. Bacchelli. A dataset for API usage. In *Proc. MSR*, pages 506–509, 2015.
38. A. Seibel, R. Hebig, and H. Giese. Traceability in model-driven engineering: Efficient and scalable traceability maintenance. In *Software and Systems Traceability.*, pages 215–240. Springer, 2012.
39. A. Shatnawi, H. Mili, G. El-Boussaidi, A. Boubaker, Y. Guéhéneuc, N. Moha, J. Privat, and M. Abdellatif. Analyzing program dependencies in Java EE applications. In *Proc. MSR*, 2017.
40. R. Stevens, C. D. Roover, C. Noguera, A. Kellens, and V. Jonckers. A logic foundation for a general-purpose history querying tool. *Sci. Comput. Program.*, 96:107–120, 2014.
41. A. Zisman. Using rules for traceability creation. In *Software and Systems Traceability*, pages 147–170. Springer, 2012.