Project Number 732223

D2.5 Dependency Analysis Components

Version 1.0
28 December 2018
Final

Public Distribution

Centrum Wiskunde & Informatica (CWI)

Project Partners: Athens University of Economics & Business, Bitergia, Castalia Solutions, Centrum Wiskunde & Informatica, Eclipse Foundation Europe, Edge Hill University, FrontEndART, OW2, SOFTEAM, The Open Group, University of L′Aquila, University of York, Unparallel Innovation

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L′Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	First draft	04 December 2018
0.2	First internal release	16 December 2018
1.0	Final release	28 December 2018

Table of Contents

Executive Summary

This document presents the final version of the software components developed in the context of **Task 2.1**, **Task 2.2**, and **Task 2.3** for the CROSSMINER platform:

Task 2.1 Inference of project build configuration.

Task 2.2 Modeling framework semantics.

Task 2.3 Dependency analysis.

This document is the latest iteration on the dependency components of the CROSSMINER platform. It updates and supersedes the previous deliverables **D2.2 – Framework Modelling Components** and **D2.4 – Dependency Inference Components**. It focuses on the *software* we develop for **Task 2.1**, **Task 2.2**, and **Task 2.3**. For more information on the research questions we address, the way we infer and analyze dependencies from meta-data and bytecode for Apache Maven and OSGi in the context of the CROSSMINER project, and an empirical study of OSGi best practices in the Eclipse ecosystem, we refer the reader to the companion deliverable **D2.3 – Dependency Inference and Analysis – Final Progress Report**, which also contains the list of requirements emerging from CROSSMINER partners that are addressed by our tool, and our publication at the *15th International Conference on Mining Software Repositories* (MSR'18) [3].

The software we develop in this context aims at automatically inferring the dependencies of the software artifacts analyzed by the CROSSMINER platform, by extracting information from their build configuration, meta-data, and source code. Specifically, we cover two of the most popular frameworks for dependencies management in the Java ecosystem: OSGi and Apache Maven (**Task 2.1** and **Task 2.2**). More specifically, we pay special attention to the way OSGi is used within the Eclipse ecosystem, as part of the Eclipse plug-in model. To avoid over-generalizing our OSGi results, we adapt our model to the specificities of the Eclipse plug-in model (**Task 2.3**), as required for instance in the Eclipse Europe Foundation use case.

Building on its success to define various metric providers in the context of the predecessor project OSSMETER, we rely on Rascal [2] to implement all the components necessary for analyzing OSGi and Apache Maven meta-data and dependencies: parsers, model builders, metrics, analyzers, and refactoring tools.

In this document:

- We give a general overview of the software we developed in Section 1;
- We present the architecture of the OSGi analyzer in Section 2;
- We detail the integration of the OSGi and Apache Maven miners and analyzers within the CROSSMINER platform in Section 3;
- We present the visualizations associated to these metrics in Section 4;
- We conclude in Section 5.

1 Overview

This document presents the software implementation resulting from **Task 2.1**, **Task 2.2**, and **Task 2.3**. The software output of these tasks consists of several projects publicly available on the CROSSMINER organization's GitHub account (`https://github.com/crossminer/`). They realize some of the CROSSMINER components assigned to **WP2 – Mining Source Code** and described in **WP8 – Platform Integration and Evaluation**:

Meta-data Miner and Dependency Miner The meta-data miner and dependency miner are realized by two components in the main repository (`https://github.com/crossminer/scava/`): one for Apache Maven (the plugin `org.eclipse.scava.dependency.model.maven`) and one for OSGi (the plugin `org.eclipse.scava.dependency.model.osgi`); these projects are the backbone of the dependency metric providers;

Dependency Metrics The main repository also contains a number of predefined metrics in the plug-ins `org.eclipse.scava.metricprovider.trans.rascal.dependency.osgi` and `org.eclipse.scava.metricprovider.trans.rascal.dependency.maven`; these metrics are meant to showcase the capabilities of our tool and to implement specific metrics required in the use cases; the set of metrics, however, is extensible: defining new domain- or use case-specific metrics alongside the predefined ones is straightforward;

OSGi Analyzer An OSGi analyzer, which can be used independently from the CROSSMINER platform, is available in a separate repository (`https://github.com/crossminer/osgi-analysis-rascal/`); it allows us to analyze large corpora of OSGi projects separately without having to run every other metric of the CROSSMINER platform.

2 Architecture of the OSGi Analysis Tool

Our analysis tool aims at extracting factual and actionable information related to dependency management from OSGi bundles. It can be employed to analyze large corpora of OSGi bundles independently from the CROSSMINER platform whenever necessary. Nonetheless, it is also fully integrated with the CROSSMINER platform, as presented in Section 3. It is fully implemented in Rascal, a one-stop shop for meta-programming that supports source code analysis, transformation, and generation [2]. Rascal is a functional programming language where data is immutable that offers many common functional programming concepts such as pattern matching, algebraic data types, higher-order functions, and comprehensions.

In OSGi, the primary unit of modularization is a *bundle*. A bundle is a cohesive set of Java packages and classes (and possibly other arbitrary resources) that together provide some meaningful functionality to other bundles [4]. A bundle is typically deployed in the form of a Java archive file (JAR) that embeds a *Manifest file* describing its content, its meta-data (e.g., version, platform requirements, execution environment), and its dependencies towards other bundles. We refer the interested reader to **D2.3: Dependency Inference and Analysis – Final Progress Report** for more information on OSGi bundles and manifests. The main input of the tool is thus a set of JAR files corresponding to OSGi bundles containing Manifest files and the associated Java bytecode. The analysis process consists of four main steps, which are implemented in four separate components:

1. In the first step, the *Parsing* component takes as input the OSGi bundles in the form of JARs and turns them into an exploitable parse tree that can be manipulated in Rascal;

2. In the second step, the *Builder* component turns the parse tree into a dedicated OSGi M^3 model that stores information about the artefacts in the form of attributed relations;

3. In the third step, the *Analysis* component defines a set of metrics that turn the raw information stored in the OSGi M^3 model into actionable information to answer dependency-related questions;

4. In the fourth and last step, the *Refactoring* component may automatically transform the analyzed Manifest files to make them comply to the OSGi best practices we identified.

We introduce each of these components, along with illustrative examples, in the remainder of this section.

2.1 The *Parsing* Component

The *Parsing* component defines the syntax of the meta-data files we analyze. For the purpose of OSGi analysis, we implemented a grammar in Rascal that is able to parse the headers of interest in OSGi Manifest files. An excerpt of this grammar is given in Listing 1. It essentially specifies a set of production rules defining certain parts of the syntax of Manifest files, as formalized in the OSGi Specification Release 6 [4]. Manifest files often contain vendor-specific and implementation-specific headers (e.g., the **`Eclipse-PlatformFilter`** header in Eclipse Equinox). They are simply ignored in the grammar to focus only on the headers of interest related to dependency management (**`Require-Bundle`**, **`Import-Package`**, etc.). From this grammar specification, Rascal automatically generates a parser for Manifest files. Given a particular Manifest, the parser then produces a parse tree that can be further processed directly within Rascal.

```
start syntax Manifest
 = headers: Header* headers;

syntax Header
 = bundleSymbolicName: HeaderBundleSymbolicName bundleSymbolicName
 | bundleVersion: HeaderBundleVersion bundleVersion
 | dynamicImportPackage : HeaderDynamicImportPackage dynamicImportPackage
 | exportPackage: HeaderExportPackage exportPackage
 | importPackage: HeaderImportPackage importPackage
 | requireBundle: HeaderRequireBundle requireBundle
 | customHeader: HeaderCustom
 ;

syntax HeaderBundleSymbolicName
 = 'Bundle-SymbolicName' ':' QualifiedName name
   (';' {BundleSymbolicNameParameter ';'}+)?;

syntax BundleSymbolicNameParameter
 = singleton: 'singleton' ':=' Boolean singleton
 | fragmentAttachment: 'fragment-attachment' ':=' DirectiveFragmentAttachment
   fragmentAttachment
 | mandatory: 'mandatory' ':=' AttributeExpression mandatory
 ;

syntax HeaderBundleVersion
 = 'Bundle-Version' ':' Version version;

syntax HeaderDynamicImportPackage
 = 'DynamicImport-Package' ':' {DynamicImportDescription ','}+ descriptions;

syntax DynamicImportDescription
 = {WildCardNames ';'}+ dynamicImports (';' {DynamicImportPackageParameter ';'}+)?;

syntax WildCardNames
 = {WildCardName ';'}+ wildCardNames;

syntax WildCardName
 = packageName: QualifiedName
 | packageWildCard: QualifiedName '.*'
 | globalWildCard: '*';

syntax DynamicImportPackageParameter
 = version: 'version' '=' QuotedHybridVersion version
 | bundleSymbolicName: 'bundle-symbolic-name' '=' QualifiedName bundleSymbolicName
 | bundleVersion: 'bundle-version' '=' QuotedHybridVersion bundleVersion;

syntax HeaderExportPackage
 = 'Export-Package' ':' {ExportPackage ','}+ packages;

syntax ExportPackage
 = {QualifiedName ';'}+ packageNames (';' {ExportPackageParameter ';'}+)?;

[... 448 lines ommitted ...]
```

Listing 1: An excerpt of the OSGi Manifest grammar in Rascal.

2.2 The *Builder* Component

The *Builder* component is in charge of creating a dedicated OSGi M^3 model from the result of parsing a set of OSGi Manifest files. The OSGi M^3 model is a dedicated "metamodel" that represents information about a corpus of bundles in the form of a set of attributed relations (denoted **rel** in Rascal). Its syntax, given by an *Algebraic Data Type*, is shown in Listing 2. Table 1 gives a description of each of the relations it contains.

Essentially, the idea is to map physical bundle locations (the JAR files given as input to the tool) to a logical location that uniquely identifies a bundle using its symbolic name and precise version. Then, the other relations of the M^3 model use these logical locations to create relations between bundles based on the information extracted from the Manifest files. These relations typically carry a **map** of parameters, for instance to store the precise *bundle-version* specified in a **Require-Bundle** header.

```
data OSGiModel = osgiModel (
loc id,
  rel[loc logical, loc physical, map[str,str] params] locations = {},
  rel[loc bundle, loc reqBundle, map[str,str] params] requiredBundles = {},
  rel[loc bundle, loc impPackage, map[str,str] params] importedPackages = {},
  rel[loc bundle, loc expPackage, map[str,str] params] exportedPackages = {},
  rel[loc bundle, loc dynImpPackage, map[str,str] params] dynamicImportedPackages = {},
  rel[loc bundle, loc impPackage] importedPackagesBC = {},
  rel[loc bundle, loc package] bundlePackagesBC = {},
  rel[loc bundle, set[Header] header] headers = {}
);
```

Listing 2: The OSGi M^3 model in Rascal.

It is important to note that some of these relations define links between OSGi Manifest files and the Java bytecode they are attached to. In order to do so, our tool relies on the Java M^3 model that has been developed in the context of the predecessor OSSMETER project [1]. The Java M^3 model stores information about the bundles' code (for instance inheritance relations between classes, overriding relation between methods, etc.). The `importedPackagesBC` relation, for example, associates a logical bundle location (`loc bundle`) to a set of Java packages extracted from the M^3 model and uniquely identified by a Rascal location (`loc package`). In this particular case, this information is used to look for superfluous bundle dependencies, i.e., dependencies that are declared in the Manifest but not used in the actual code of a bundle.

Table 1: Relations of the OSGi M^3 model.

Relation	Description
`rel[loc,loc,map] locations`	Links logical URLs, used as bundle identifiers, to their physical location. The bundle's version is included in a map.
`rel[loc,loc,map] requiredBundles`	Links bundle logical locations to required bundle logical locations. Main **Require-Bundle** attributes are set in a map.
`rel[loc,loc,map] importedPackages`	Links bundle logical locations to imported package logical locations. Main **Import-Package** attributes are set in a map.
`rel[loc,loc,map] exportedPackages`	Links bundle logical locations to exported package logical locations. Main **Export-Package** attributes are set in a map.
`rel[loc,loc,map] dynamicImportedPackages`	Links bundle logical locations to dynamically imported package logical locations. Main **DynamicImport-Package** attributes are set in a map.
`rel[loc bundle, set[Header] header] headers`	Stores all other headers.

```
public int getRequiredBundlesSize(OSGiM3Model model)
  = size(model.requiredBundles);
```

Listing 3: Using Rascal to compute the number of `Require-Bundle` relations in an OSGi M^3 model.

Another important aspect of the OSGi M^3 model is that it can be computed once for a given corpus and then serialized separately. This avoids having to reconstruct a new OSGi M^3 model from scratch (with the overhead of the parsing and building phases) every time a new metric is added or modified.

From this model, it is straightforward to implement simple high-level metrics that return factual information about the analyzed bundles. Listing 3, for instance, depicts a simple function that, given an OSGi M^3 model, returns the number of `Require-Bundle` relations in the model. Some of the metrics we defined for OSGi are detailed in Section 3.

2.3 The *Analysis* Component

The *Analysis* component takes the result of the *Builder* component, i.e., an OSGi M^3 model, and turns it into factual information related to dependency management in OSGi. Roughly, it turns raw data into meaningful information that answers specific questions for the developers. In **D2.3: Dependency Inference and Analysis – Final Progress Report**, we use the *Analysis* component to define metrics related to best practices in OSGi.

2.4 The *Refactoring* Component

We implemented a number of refactorings atop our analysis of best practices in the OSGi ecosystem. These refactorings are currently being integrated within the CROSSMINER platform. When a smell is detected in the dependency meta-data, the analysis component formulates a recommendation to which is associated a refactoring. We are currently putting every component together (the dependency miners, the Knowledge Base, and the IDE) to automatically refactor the dependency meta-data in the CROSSMINER IDE at M30.

For instance, an excerpt of the refactoring transforming all `Require-Bundle` into a set of `Import-Package`—corresponding to the best practice [B1] as described in **D2.3: Dependency Inference and Analysis – Final Progress Report**—is given in Listing 4 and available online (`https://github.com/crossminer/osgi-analysis-rascal/blob/master/code/DependenciesAnalyzer/src/org/analyzer/osgi/analysis/smells/requireBundle/Modifier.rsc`). The `modifyManifests` function considers an OSGi M^3 to perform the corresponding refactoring (Line 1). First, it identifies and considers bundles that do not provide an implementation (aka. extension) to a given plug-in API (aka. extension point) (Lines 2-6). Extensions and extension points are included as part of the capabilities offered by the Plug-in Development Environment (PDE)[1] of Eclipse. Requiring a bundle that offers a target extension point is mandatory in the case of extension bundles; thus, the `Require-Bundle` header is not modified in these cases.

Afterwards, for a given non-extension bundle we check which are its corresponding mandatory required bundles (i.e., bundles that export split packages, the OSGi system bundle, or unresolved bundles) (Line 7). For the remaining bundles defined in the `Require-Bundle` header we extract the exported packages that are actually being used in the code (Line 8). This operation is supported by the `bundleToPackageDependencies` function (Lines 17-25), where for a given bundle b and each one of its required bundles r, we intersect the set of exported

[1]PDE includes OSGi tooling as well as additional functionality to manage plug-ins, fragments, features, update sites, and Rich Client Platform (RCP) products. More information available at `https://www.eclipse.org/pde/`.

```
1  void modifyManifests(OSGiModel model) {
2   Extension ext = getExtensionBundles(model);
3   nonExtensionBundles = getComplementExtensionReqBundles(model,ext);
4
5   for(<logical,physical,params> ← model.locations,
6    size(nonExtensionBundles[logical]) > 0) {
7    mandatoryReqBundles = getMandatoryRequiredBundles(logical,model);
8    importedPackages = bundleToPackageDependencies(logical,mandatoryReqBundles,model);
9
10   mandatoryReqBundlesStr = requireBundleToStr(logical,mandatoryReqBundles,model);
11   importedPackagesStr = importPackageToStr(logical,importedPackages,model);
12   changeManifest(physical, importPackage = importedPackagesStr,
13    requireBundle = mandatoryReqBundlesStr);
14  }
15 }
16
17 set[loc] bundleToPackageDependencies(loc bundle, set[loc] mandatoryReqBundles,
18 OSGiModel model) {
19  flatExportedPackages = toBinaryRelation(model.exportedPackages);
20  flatImportedPackages = toBinaryRelation(model.importedPackages);
21  importedPackages = {*((flatExportedPackages[b] & model.importedPackagesBC[bundle])
22   - flatImportedPackages[bundle]) |
23    <b,p> ← model.requiredBundles[bundle], b notin mandatoryReqBundles};
24  return importedPackages;
25 }
```

Listing 4: Refactoring `Require-Bundle`s into corresponding `Import-Package`s in Rascal.

packages of r with the set of packages available in the bytecode of b and we remove the set of packages that are already defined as part of the `Import-Package` header of b. Finally, the `Require-Bundle` and `Import-Package` headers are recomputed (Lines 10-11), and the corresponding manifest file is refactored (Lines 12-13).

2.5 Smells Detection

The OSGi analysis tool is able to automatically detect OSGi smells from an analysis of project meta-data. Specifically, the six smells used in the Eclipse ecosystem evaluation presented in **D2.3 – Dependency Inference and Analysis – Final Progress Report** are readily implemented in the platform. For each of them, a *detector* which detects the smell and an *automatic refactoring tool* which automatically transforms the meta-data to conform to the best practice (see Section 2.4) are defined.

3 CROSSMINER Components

The integration of our meta-data and dependency miners for Apache Maven and OSGi relies on two main plug-ins integrated within the CROSSMINER platform: one for OSGi and one for Apache Maven. Atop these plug-ins, we defined a number of metric providers that leverage the results of the dependency analysis to compute high-level metrics that can directly be interpreted by the CROSSMINER users.

Most importantly, the analysis facilities presented in Section 2 are readily available to any dependency metric providers.

3.1 Rascal Dependency

The OSGi and the Apache Maven CROSSMINER plug-ins have a dependency on the Rascal bundle. This dependency is included as a required bundle in their Manifest files (`rascal_bundle` v0.10.0), and we added the Rascal repository to the `pom.xml` file of the `org.eclipse.scava.configuration` project. For development and test purposes, developers can install the Rascal plugin in the Eclipse IDE by pointing to the corresponding update site[2].

3.2 OSGi Dependency Miner

The `org.eclipse.scava.dependency.model.osgi` project encapsulates the OSGi analysis tool described in Section 2 in the form of a CROSSMINER plug-in. Besides, it exposes a single API function that allows any metric provider to retrieve the OSGi M^3 model of the project that is currently analyzed. This function is depicted in Listing 5. Given a location pointing to the working copy of the project (Line 1), the function first retrieves all the `MANIFEST.MF` files found in the current project (Line 2) and builds the corresponding OSGi M^3 model (Line 3). It then returns a composed M^3 model that gathers all dependency-related information from all the Manifests (Line 4).

```
1 OSGiModel getOSGiModelFromWorkingCopy(loc workingCopy) =
2   manifestFiles = manifestLocations(workingCopy,{});
3   models = {createOSGimodel(workingCopy, f) | f ← manifestFiles};
4   return composeOSGiModels(workingCopy, models);
5 }
```

Listing 5: Extracting the composed OSGi M^3 model of the current project.

This public API is intended to be used by all metric providers requiring accessing information related to the dependencies of a project. We implemented a number of predefined metrics for OSGi in the metric provider plug-in `org.eclipse.scava.metricprovider.trans.rascal.dependency.osgi`. Just as any other metric provider, this plug-in subscribes to the extension point `scava.rascal.metricprovider`. Other components requiring some of the information computed by these metrics can gather this data from the non-relational database of the project. Doing so, it is identified and invoked by the CROSSMINER platform when analyzing projects.

Listing 6 presents some of the metrics that have been defined in this plug-in. As mentioned earlier, each metric explicitly invokes the `getOSGiModelFromWorkingCopy` API to retrieve the OSGi M^3 model of the current project. Then, it computes a given metric based on the information stored in the M^3 model. In particular,

[2] https://update.rascal-mpl.org/unstable

```
1  @metric{numberOSGiBundleDependencies}
2  @doc{Retrieves the number of OSGi bunlde dependencies (i.e. Require-Bundle dependencies).}
3  @friendlyName{Number all OSGi bundle dependencies}
4  @appliesTo{java()}
5  int numberOSGiBundleDependencies(
6    map[loc, loc] workingCopies = ()) {
7    if(repo ← workingCopies) {
8      m = getOSGiModelFromWorkingCopy(workingCopies[repo]);
9      return size(m.requiredBundles=={}?{}:m.requiredBundles.reqBundle);
10   }
11   return 0;
12 }
13
14 @metric{numberUsedOSGiImportedPackagesInSourceCode}
15 @doc{Retrieves the number of OSGi imported packages USED in the project source code.}
16 @friendlyName{Number OSGi imported packages in source code}
17 @appliesTo{java()}
18 int numberUsedOSGiImportedPackagesInSourceCode(
19   ProjectDelta delta = ProjectDelta::\empty(),
20   map[loc, loc] workingCopies = (),
21   rel[Language, loc, M3] m3s = {}) {
22   M3 m3 = systemM3(m3s, delta = delta);
23   if(repo ← workingCopies) {
24     m = getOSGiModelFromWorkingCopy(workingCopies[repo]);
25     return size(
26       (ternaryReltoSet(m3.importedPackages)
27        + ternaryReltoSet(m3.dynamicImportedPackages))
28        - (getImportedPackagesBC(m3)));
29   }
30   return 0;
31 }
```

Listing 6: Examples of predefined OSGi metrics.

if we look at the `numberUsedOSGiImportedPackagesInSourceCode` metric, we first define its identifier with the `@metric` annotation (Line 14). Besides, its corresponding description, its name in natural language, and the programming language of the projects that it can process are specified in the `@doc`, `@friendlyName`, and `@appliesTo` annotations, respectively (Lines 15-17). Then, we declare the Rascal function and, as previously defined in the OSSMETER deliverable **D3.2: Report on Source Code Activity Metrics**, we specify the data structures that are required by the metric (i.e., `delta`, `workingCopies`, and `m3s`) (Lines 19-22). These parameters are retrieved from memory once the metric provider is executed. Afterwards, we compute the system M^3 model (Line 22); and, for a given working copy, we compute the union of the imported and dynamic imported packages of the project, and we remove the packages that are actually being used in the source code. The cardinality of the resulting set is then returned (Lines 25-30).

3.3 Apache Maven Dependency Miner

The plug-in `org.eclipse.scava.dependency.model.maven` implements the meta-data and dependency miner for Apache Maven. Maven Project Object Model (POM) files (`pom.xml`) are parsed using the built-in XML

```
data MavenModel = mavenModel (
  loc id,
  rel[loc logical, loc physical, map[str,str] params] locations = {},
  rel[loc project, loc dependency, map[str,str] params] dependencies = {}
);
```

Listing 7: The Maven M^3 model in Rascal.

parser of Rascal. From a set of `pom.xml` files, the parser builds a Maven M^3 model conforming to the ADT depicted in Listing 7.

The plug-in `org.eclipse.scava.metricprovider.trans.rascal.dependency.maven` contains a number of predefined metrics for Maven, built atop the facilities provided by the `org.eclipse.scava.dependency.model.maven` plug-in. These metrics are implemented with the same principles described in Section 3.2. Listing 8 gives an excerpt of some of them. These metrics follow the general pattern promoted in the predecessor project OSSMETER: they declare a number of parameters that, according to their name, are automatically passed by the metric execution engine. For instance, the `numberUniqueMavenDependencies` metric considers the `workingCopies` parameter, which is automatically computed and passed to the metric by the platform (Line 6). To retrieve the Maven M^3 model, the metric reuses the API defined in the `org.eclipse.scava.dependency.model.maven` plug-in. The `getMavenModelFromWorkingCopy` function retrieves a Maven M^3 model that gathers information on all the `pom.xml` files found in the project that is currently analyzed (Lines 7-8). Then, it checks the number of unique dependencies in the model by considering the field selection operator provided by Rascal. The cardinality of the set is finally retrieved (Line 9). Naturally, this set of metrics can easily be extended with new bespoke analysis. These bespoke metrics may even cross-fertilize different metric (e.g., dependencies and NPL, or source code and dependencies).

```
@metric{numberUniqueMavenDependencies}
@doc{Retrieves the number of unique Maven dependencies.}
@friendlyName{Number unique Maven dependencies}
@appliesTo{java()}
int numberUniqueMavenDependencies(
  map[loc, loc] workingCopies = ()) {
  if(repo ← workingCopies) {
    m = getMavenModelFromWorkingCopy(workingCopies[repo]);
    return size(m.dependencies=={}?{}:m.dependencies.dependency);
  }
  return 0;
}

@metric{numberOptionalMavenDependencies}
@doc{Retrieves the number of optional Maven dependencies.}
@friendlyName{Number optional Maven dependencies}
@appliesTo{java()}
int numberOptionalMavenDependencies(
  map[loc, loc] workingCopies = ()) {
  if(repo ← workingCopies) {
    m = getMavenModelFromWorkingCopy(workingCopies[repo]);
    return (0 | it + 1 | <p,d,params> ← m.dependencies, params["optional"]=="true");
  }
  return 0;
}
```

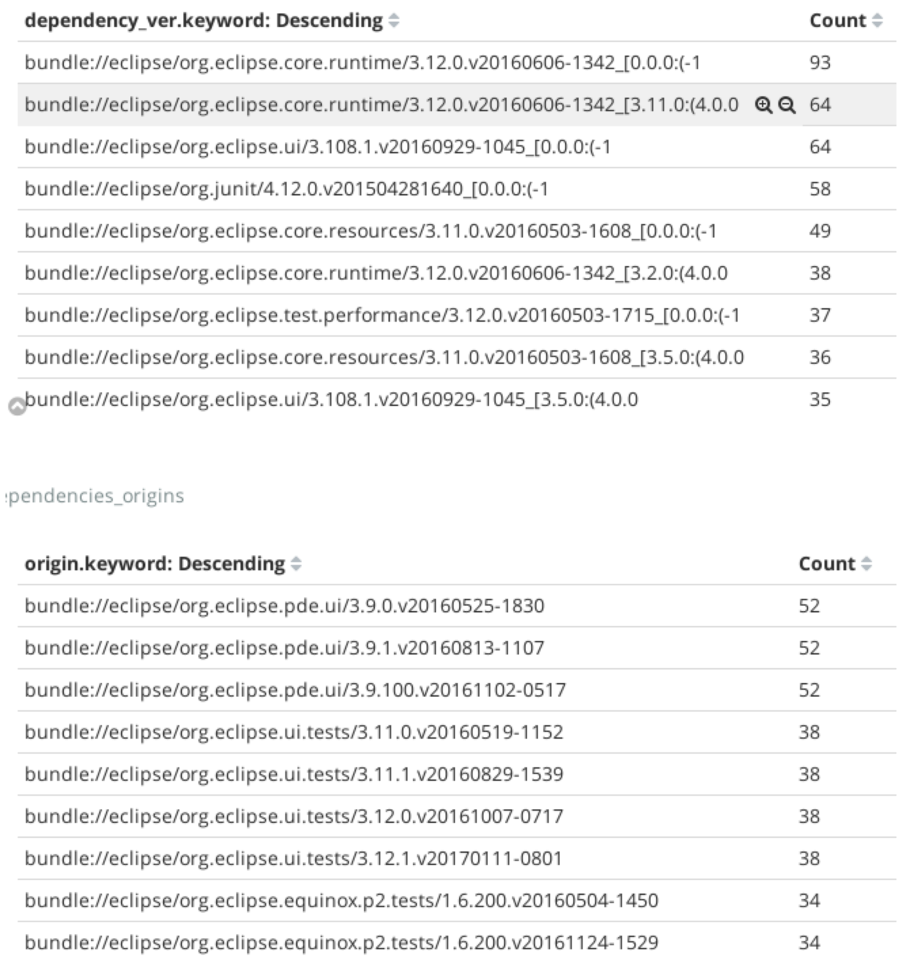
Listing 8: The `numberUniqueMavenDependencies` and `numberOptionalMavenDependencies` metrics defined atop the Maven M^3 model in Rascal.

4 Dependencies Metrics in the CROSSMINER Dashboard

While the dependency analysis components are already readily available in the CROSSMINER platform, we are currently working on displaying the dependency metrics in the CROSSMINER dashboards in an effective way.

As shown in the different listings of this document, most metrics compute simple numbers specifying, for instance, what is the number of optional dependencies in a Maven project, or how many of the declared OSGi dependencies are not actually used in the source code.

Just like most source code metrics, these metrics are currently displayed as historical charts in the dashboards, highlighting how they are evolving over the development time of the analyzed project, and allowing developers and managers to react when, for instance, the number of unused dependencies grows. The raw data produced by the metrics can also be accessed from the dashboards, as depicted in Figure 1.



dependency_ver.keyword: Descending	Count
bundle://eclipse/org.eclipse.core.runtime/3.12.0.v20160606-1342_[0.0.0:(-1	93
bundle://eclipse/org.eclipse.core.runtime/3.12.0.v20160606-1342_[3.11.0:(4.0.0	64
bundle://eclipse/org.eclipse.ui/3.108.1.v20160929-1045_[0.0.0:(-1	64
bundle://eclipse/org.junit/4.12.0.v201504281640_[0.0.0:(-1	58
bundle://eclipse/org.eclipse.core.resources/3.11.0.v20160503-1608_[0.0.0:(-1	49
bundle://eclipse/org.eclipse.core.runtime/3.12.0.v20160606-1342_[3.2.0:(4.0.0	38
bundle://eclipse/org.eclipse.test.performance/3.12.0.v20160503-1715_[0.0.0:(-1	37
bundle://eclipse/org.eclipse.core.resources/3.11.0.v20160503-1608_[3.5.0:(4.0.0	36
bundle://eclipse/org.eclipse.ui/3.108.1.v20160929-1045_[3.5.0:(4.0.0	35

pendencies_origins

origin.keyword: Descending	Count
bundle://eclipse/org.eclipse.pde.ui/3.9.0.v20160525-1830	52
bundle://eclipse/org.eclipse.pde.ui/3.9.1.v20160813-1107	52
bundle://eclipse/org.eclipse.pde.ui/3.9.100.v20161102-0517	52
bundle://eclipse/org.eclipse.ui.tests/3.11.0.v20160519-1152	38
bundle://eclipse/org.eclipse.ui.tests/3.11.1.v20160829-1539	38
bundle://eclipse/org.eclipse.ui.tests/3.12.0.v20161007-0717	38
bundle://eclipse/org.eclipse.ui.tests/3.12.1.v20170111-0801	38
bundle://eclipse/org.eclipse.equinox.p2.tests/1.6.200.v20160504-1450	34
bundle://eclipse/org.eclipse.equinox.p2.tests/1.6.200.v20161124-1529	34

Figure 1: Raw data related to dependency metrics in the dashboards.

A proof-of-concept was developed independently to showcase the representation of dependencies as a dependency graph, as required in the project requirements (Figure 2). While this representation is not a "metric" in itself, it helps developers and decision makers to analyze which other components an OSS project is relying on. This dependency graph is not yet integrated in the dashboards, but this will be done with the help of Bitergia in later stages of the project. The current version of the proof-of-concept is available online: https://crossminer.biterg.io/app/kibana#/dashboard/AV7m5g5kfuk55ZFAddq4.

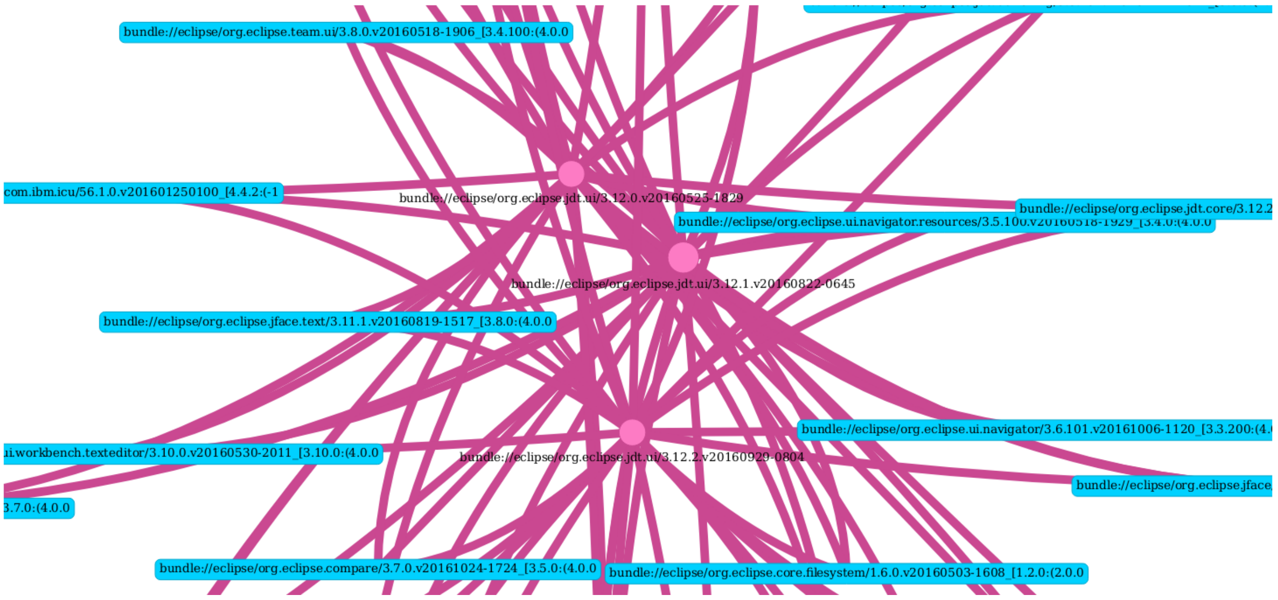

bundle://eclipse/org.eclipse.team.ui/3.8.0.v20160518-1906_[3.4.100:(4.0.0
com.ibm.icu/56.1.0.v201601250100_[4.4.2:(-1
bundle://eclipse/org.eclipse.jdt.ui/3.12.0.v20160525-1829
bundle://eclipse/org.eclipse.jdt.core/3.12.2
bundle://eclipse/org.eclipse.ui.navigator.resources/3.5.100.v20160518-1929_[3.4.0:(4.0.0
bundle://eclipse/org.eclipse.jdt.ui/3.12.1.v20160822-0645
bundle://eclipse/org.eclipse.jface.text/3.11.1.v20160819-1517_[3.8.0:(4.0.0
bundle://eclipse/org.eclipse.ui.navigator/3.6.101.v20161006-1120_[3.3.200:(4.
ui.workbench.texteditor/3.10.0.v20160530-2011_[3.10.0:(4.0.0
bundle://eclipse/org.eclipse.jdt.ui/3.12.2.v20160929-0804
bundle://eclipse/org.eclipse.jface
3.7.0:(4.0.0
bundle://eclipse/org.eclipse.compare/3.7.0.v20161024-1724_[3.5.0:(4.0.0
bundle://eclipse/org.eclipse.core.filesystem/1.6.0.v20160503-1608_[1.2.0:(2.0.0

Figure 2: Excerpt of a dependency graph in the dashboards.

5 Conclusion

This deliverable reports on the final progress made for **Task 2.1**, **Task 2.2** and **Task 2.3**, focusing on the software artifacts we implemented in this context. It accompanies the deliverable **D2.3: Dependency Inference and Analysis – Final Progress Report** which provides complementary information on the research questions we address and the way we infer and analyze OSGi and Apache Maven dependencies in the context of CROSSMINER.

We presented our OSGi analysis tool and described how the metrics we implemented atop the OSGi and Maven M^3 models are integrated with the CROSSMINER platform.

References

[1] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A general model for code analytics in Rascal. In *IEEE 1st International Workshop on Software Analytics*, pages 25–28, 2015.

[2] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, pages 222–289, Berlin, Heidelberg, 2011. Springer.

[3] Lina Ochoa, Thomas Degueule, and Jurgen J. Vinju. An empirical evaluation of OSGi dependencies best practices in the Eclipse IDE. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 170–180, 2018.

[4] The OSGi Alliance. OSGi core release 6 specification, Jun 2014.