Recovering Inter-Project Dependencies in Software Ecosystems

Mircea Lungu
Faculty of Informatics
University of Lugano, Switzerland
mircea.lungu@usi.ch

Romain Robbes
Computer Science Department (DCC)
University of Chile, Santiago
rrobbes@dcc.uchile.cl

Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland
michele.lanza@usi.ch

ABSTRACT

In large software systems, knowing the dependencies between modules or components is critical to assess the impact of changes. To recover the dependencies, fact extractors analyze the system as a whole and build the dependency graph, parsing the system down to the statement level. At the level of software ecosystems, which are collections of software projects, the dependencies that need to be recovered reside not only within the individual systems, but also between the libraries, frameworks, and entire software systems that make up the complete ecosystem; scaling issues arise.

In this paper we present and evaluate several variants of a lightweight and scalable approach to recover dependencies between the software projects of an ecosystem. We evaluate our recovery algorithms on the Squeak 3.10 Universe, an ecosystem containing more than 200 software projects.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Measurements

1. INTRODUCTION

Static and evolutionary analysis of software systems has traditionally focused on single systems. However, software systems do not exist by themselves, but instead they exist in larger contexts called "software ecosystems". In previous work we have studied software ecosystems and argued for the importance of holistic ecosystems analysis to better understand both the ecosystems and the individual systems that compose them. Our approach was driven by the goal of automating the generation of ecosystem viewpoints that capture both the social and the structural aspects of software ecosystems [3]. In this paper we focus on structural aspects of an ecosystem.

A preeminent sub-problem of static analysis is the analysis of dependencies between the elements of a system. Knowing dependencies among components in a system is critical for assessing the usage of a given component and the impact of changes performed on it.

Recovering ecosystem-level dependencies is important for: (1) understanding ecosystem's structure – a developer knows the structure of the project that he is working on, but not always how it fits in the context of the ecosystem; (2) monitoring framework usage – if developers of a framework are aware of how their project is used, they can make informed decisions about API evolution; (3) understanding how projects use each other – new developers of a project that depends on an existing framework, would find it useful to extract usage patterns that other projects have of that framework.

Extracting dependencies between projects is harder than extracting dependencies between the components of a project, for a number of reasons:

1. *Scale.* Ecosystems contain information orders of magnitude larger than individual systems. Scaling up approaches tailored for single systems is doubtful at best.
2. *Accuracy.* The inter-project dependencies, i.e., dependencies between artifacts residing in two different software systems are simply not considered by existing static analysis approaches.
3. *Errors.* Parsing a project in order to extract a model of its structure usually requires that the project compiles. However, at the ecosystem level, there are times when multiple deprecated projects do not actually compile.

Considering all these issues, we analyze several lightweight techniques for recovering the dependencies in entire ecosystems. Once the dependencies are known, the road is open for more accurate analyses of the software ecosystem. The contributions of this paper are: (1) The definition of *Ecco*, a lightweight software ecosystem representation that keeps track of enough data to recover dependencies between projects; (2) Several dependency-recovery algorithms which reflect characteristics of source code at the ecosystem level; and (3) The evaluation of these dependency-recovery algorithms on the *Squeak 3.10 Universe*, an ecosystem of 211 projects.

Structure of the paper. In Section 2 we discuss related work. Section 3 gives an overview of ecosystems. Section 4 presents our lightweight model *Ecco*. Section 5 presents our evaluation methodology, while Section 6 presents dependency recovery algorithms and their performance, which we discuss in Section 7. Section 8 concludes.

2. RELATED WORK

Ossher et al. resolve dependencies between projects in order to obtain a successful build of a target project [6]. Their approach parses a project and looks for the type definitions that are missing in the projects referenced in Sourcerer's project repository. Their approach requires finer grained information than ours. They propose and evaluate one dependency resolution algorithm, while we evaluate several variants of lightweight algorithms. They target primarily the Java programming language, in which this type information is available, while our current case study is a Smalltalk ecosystem, which lacks static type information and hence it is a harder problem.

Mockus indexes a large amount of open source versioning control systems, in order to run analyses on the entire public source code base [5]. In an exploratory study, Gonzales-Barahona et al. have analyzed the Debian Linux distribution. They visualize the dependencies between projects in Debian, but those are declared dependencies [2].

In our previous work [4] we visualized dependencies between projects at the ecosystem level, but we did not extract those dependencies ourselves, but rather, we extracted them from the super-repository meta-data.

3. SOFTWARE ECOSYSTEMS

In our previous work we have defined a software ecosystem as *a collection of software projects which are developed and co-evolve in the same environment* [3].

The environment can have a geo-spatial identity as in the case of a small company or a research group, but can also be free of geographical limitations, as in the case of an open-source community or a multi-national enterprise.

We consider software ecosystems to be another level of abstraction at software analysis needs to be performed, the next after code, design, and architecture. One such type of analysis is reverse engineering an ecosystem, a process which analyzes low-level information about the projects in the ecosystem and generates high-level views that characterize the ecosystem as a whole. These high-level views can be either focused on the projects, or on the developers in the ecosystem [3].

One of the critical ecosystem views related to projects is the project dependency map [3] which presents the details of the dependencies that exist between the projects in an ecosystem.

The main source of information for extracting information about inter-project dependencies are the *super-repositories* associated with a given ecosystem. Customarily, the history of every project in the ecosystem is recorded in the version control repository of that project. At the ecosystem level we say that the history of the entire ecosystem is recorded in a super-repository. We define a super-repository as *a collection of all the version control repositories for multiple software projects* [4].

Some super-repositories contain meta-information about project configurations and project dependencies (e.g., Store for Smalltalk, the Debian Linux distribution), while others do not (e.g., CVS, SVN, Git). Our dependency extraction techniques are useful for both types of super-repositories, either to verify existing dependencies or to recover implicit ones.

4. ECCO

Treating an ecosystem with the same level of detail as a single system introduces scalability issues. We need to abstract away most of the details of individual systems if we want the construction of the model and the subsequent analyses to scale to an entire ecosystem.

Ecco models an ecosystem as a set of projects written in an object-oriented language (see Figure 1), for which we need to find dependencies.

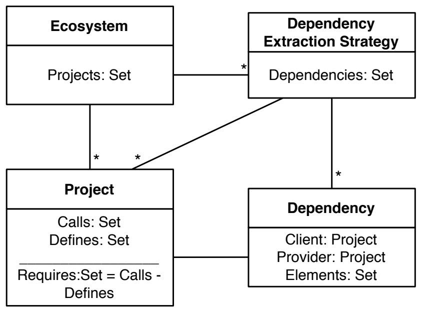

Figure 1: *Ecco*'s metamodel

In this model, a project contains three sets of entities: (1) entities **called** by the project, are entities that the project references; (2) entities **provided** by the project are entities that the project defines; and (3) entities **required** are the ones that are called but are not provided by the project.

Entities can be classes or methods. To keep the model lightweight, these entities are modelled with their identifiers only. To populate the model, one needs parsing mechanisms which recognize the definition and usage of classes and methods. To model dependencies, we use a simple relationship between a client project and a provider project. The relationship contains a set of entities that the client may use from the provider, and an optional weight or confidence in the dependency (when recovered). There are multiple possible dependency sets based on the extraction strategy.

5. EVALUATION

To evaluate the accuracy of our dependency recovery techniques we selected the *Squeak 3.10 Universe* [7], a package distribution based on the source code for Squeak Smalltalk. It contains 211 Projects that define 8,692 classes and 68,808 unique method names. We extracted the dependencies between the packages as they are specified in the universe, to create our oracle of dependencies to recover.

The *Squeak 3.10 Universe* contains 173 documented dependencies between the 211 projects. Each package further depends on the base system, but these dependencies are implicit.

There are other sources of implicit dependencies, as shown in Figure 2. On the left we see the dependencies declared in the package distribution. On the right, we see the dependecies extracted by a hypothetical algorithm. The extracted dependencies can be of three types: (1) Original—the ones that were present in the reference dependency graph, true positives; (2) New—the ones that were not present in the original graph, false positives; (3) Inferable—the ones that were not present in the original graph, but are actually indirect dependencies.

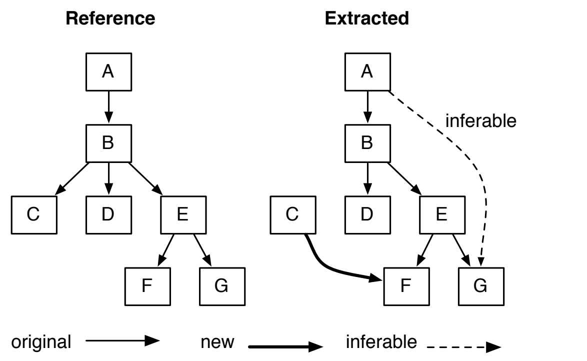

Figure 2: The retrieved dependencies can be of three possible types: original, new, and inferable

In Figure 2 the dependency from A to G can actually be correct. A might depend on functionality defined in G, but the maintainer, knowing that G is in the dependency chain of A, did not specify this dependency in an explicit manner. Since G needs to be present for B to be loaded, and B has to be present before A is loaded, specifying the dependency is optional. We classify potentially indirect dependencies as neither true nor false positives.

Once we have extracted the dependencies we build the *Ecco* model of our ecosystem. We also create a project representing the base system, in order to have a complete picture of the ecosystem, and of dependencies to the base system. Building the model of an ecosystem of the scale of *Squeak 3.10 Universe* becomes a matter of minutes.

To run the algorithms, we first build an index of each entity defined in the ecosystem. The index contains the set of projects that defines the entity. The index is the primary data structure used by the algorithms alongside the *Ecco* model of the ecosystem. Once the index is built, we run each algorithm and store the list of candidate dependencies they output.

6. EXTRACTION APPROACHES

We present several strategies for extracting dependencies from an ecosystem and analyze various parameters that affect their performance. For each technique we provide a *rationale*, detail the *algorithm*, and discuss its *results*. To compare the approaches we use the well-known IR metrics *precision*, *recall*, and *F-measure*.

6.1 Unique Method Invocations

Rationale. If a project calls a method defined only by another project, the first depends on the second.
Algorithm. For each project we build an index of the methods that it defines and the methods that it calls. If package A contains at least one method which is defined only in package B, this strategy considers that there is a dependency relationship between package A and B.
Results. $\mathbf{P} = 0.19$ $\mathbf{R} = 0.59$ $\mathbf{F} = 0.29$

None of the measures is exceptionally good, and precision is poor. Since precision and recall are complementary, we can increase one at the expense of the other. We look in turn at techniques for improving the precision or the recall.

In order to improve the precision we would need to be more conservative and request that at least n unique methods be provided by a given project before we declare the existence of a dependency. For $n = 3$, we obtain a F-measure of 0.55.

To obtain a better recall we relax the condition that a method be unique, and consider that a method can be a dependency if it is declared in n or less projects. However, this degrades the precision, for $n = 2$, F falls to 0.06.

6.2 Unique Class References

Rationale. The class names are more unique than the method names, so using this strategy we expect to increase the accuracy of the extraction algorithm. If a class subclasses or references a class defined in a separate project, we assume that the first project depends on the second.
Algorithm. The strategy is the same as the previous one, but considers classes instead of methods. For each project we build an index of the classes that are declared in that package. If any class defined in package A subclasses or references a class defined in package B (package B being the only provider) we consider that A depends on B.
Results. $\mathbf{P} = 0.80$ $\mathbf{R} = 0.71$ $\mathbf{F} = 0.75$

This strategy outperforms the best strategy based on methods. We expected the precision to be much higher, but did not expect the recall to rise as well. This result is partially due to classes with unique names which are being used with common methods.

We experimented with requiring more than one unique class before we declare a dependency, but this decreased the performance; so did relaxing the uniqueness requirement.

6.3 Weighted Dependencies

Rationale. A dependency between two projects provides to the client a number of entities that it requires. Dependencies providing more (or more unique) entities are more likely to be accurate.
Algorithm. We generate all the dependencies that satisfy at least one required entity of each project, and assign a weight to them. We return the entities above a threshold. The filtering can also be a second step on any dependency list produced by any other algorithm. We defined several weighting schemes:

- *Number of dependencies* counts the number of entities the client requires that the provider satisfies.
- *Number of non-trivial dependencies.* As in [1], we distinguish between "common" names for entities (e.g., Table) and uncommon names (e.g., DependencyTable). A common name is an identifier made from a single word; we filter it out.
- *Proportional dependencies* defines the weight of each satisfied entity as the inverse of the number of projects defining it.

Results. $\mathbf{P} = 0.85$ $\mathbf{R} = 0.70$ $\mathbf{F} = 0.77$

We explored the space of possible thresholds, but we could not find a weighting scheme or a threshold that exceeded the performance of strategies that rely on the presence of uniquely defined classes, at best equalling it, which is disappointing since these algorithms are ressource intensive.

The only improvement we report is when we filter the dependencies returned by the "uniquely provided classes" algorithm with non-trivial class dependencies. This increases the precision and slightly lowers the recall, leading to an f-measure of 0.77, instead of 0.75. Some variables in the code are assumed to be class names; these tend to have more generic names than classes, so filtering on the generality of the names filters them out.

6.4 Combined Unique Methods and Classes

Rationale. By considering all the dependencies that are detected with the help of classes and methods together, we expect that the performance will be the highest.

Algorithm. We experimented with two algorithms combining classes and method information. Both involved first computing class dependencies and computing method-level dependencies in cases where class dependencies only could not decide if a dependency were warranted. The cases were:

- When a class-dependency is the sole provider of a single unique class, with a common name (e.g., Timer). In that case, we validate the dependency if the method-level dependency also satisfies at least one method.
- When a class-dependency does not provide any single unique class, but provides at least a class that is provided by 2 projects at most. We validate the dependency if a method-level dependency can be established as well, based on a number of provided methods which is higher than a given threshold.

Results. **P** = 0.85 **R** = 0.70 **F** = 0.77

The first case gave an improvement so slight that it disappeared in the rounding. When looking at the results, we saw that exactly one false negative was transformed in a true positive. The second case gave us an overall f-measure of 0.75 for very high thresholds (i.e., requiring that the dependency satisfies more than 20 methods). That score is equivalent to the score of the unique class algorithm.

7. DISCUSSION OF THE RESULTS

The overall conclusion we extract from our experiments is that simple, class-based dependency recovery techniques work best overall. So far, we were able to reach an equivalent accuracy when considering method information, but were not able to exceed it.

We see this positively, since it means that some of the best lightweight dependency extractors are the ones which are simplest to build: classes are several times less numerous than methods. At the time of writing, parsing the *Squeak 3.10 Universe* takes 4 minutes, indexing the entities takes seconds (a fraction of a second in the case of classes), and running the fastest (and most accurate) algorithm is also a matter of seconds.

In order to get a better impression of the accuracy of our best-performing algoritm, we inspected more closely its false positives and false negatives. Our algorithm returned 105 true positives, 17 false positives, and 42 false negatives.

Of the 17 false positives, 12 were actually true positives that did not prevent the code to load when the dependency was not fulfilled. A portion of them would fail at runtime, but are probably in code that is not used often. The remainder used defensive programming in order to ensure that the entity required was actually present. For instance:

```
(Smalltalk hasClassNamed: #BytecodeGenerator)
    ifFalse: [ MagmaUserError signal:
        'WriteBarrier requires NewCompiler' ]
```

Of the 42 false negatives, 9 were caused by code duplication: 3 projects were present in the universe, but were also included as copies as part of the source code of other projects. This of course impacts algorithms that rely on uniquely provided names.

Finally, 16 of the 42 false negatives and 2 of the 17 false positives were dependencies involving very small projects (10 classes or less), for which not much information is available. We plan to investigate other techniques to better recover these two specific cases.

8. CONCLUSIONS

To increase awareness between clients and providers of source code, recovering dependencies between projects of an ecosystem is critical. We presented a model of software projects and ecosystems suited for dependency recovery. *Ecco* is a very lightweight representation of projects and ecosystems: a project simply contains the list of entity names it depends on, uses and provides.

We presented several techniques for dependency recovery with heuristics matching the nature of the data present in an ecosystem. We validated them on the *Squeak 3.10 Universe*, a Smalltalk ecosystem with 211 projects. Each technique was evaluated in terms of the precision and recall of the set of dependencies it produced, compared with the oracle of manually specified dependencies in the universe.

We found that techniques using class names were both the simplest and the ones with the highest precision and recall. We identified two major factors that hamper the accuracy of our algorithms: the presence of duplicated code, and recovering dependencies involving small code bases. We will investigate how to solve these issues.

Acknowledgements.

We thank Ricardo Baeza-Yates (Yahoo! Research and University of Chile), Mark Carman and Alberto Bacchelli (University of Lugano) for insightful discussions related to this work. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "Di-CoSA" (SNF Project No. 118063).

9. REFERENCES

[1] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *WCRE*, pages 205–214, 2009.

[2] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 2008.

[3] M. Lungu. *Reverse Engineering Software Ecosystems.* PhD thesis, University of Lugano, October 2009.

[4] M. Lungu, M. Lanza, T. Girba, and R. Heeck. Reverse engineering super-repositories. In *WCRE*, pages 120–129, 2007.

[5] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *MSR*, pages 11–20, 2009.

[6] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *MSR*, pages 130–140, 2010.

[7] A. Spoon. Package universes: Which components are real candidates? Technical report, EPFL, 2006.