Synthesizing Benchmarks for Architecture Recovery Algorithms

Rodrigo Souza
rodrigorgs@gmail.com

Dalton Guerrero
dalton@dsc.ufcg.edu.br

Jorge Figueiredo
abrantes@dsc.ufcg.edu.br

Christina Chavez
flach@ufba.br

November 2, 2011

Abstract

As a software system evolves, its actual architecture may deviate from its original reference architecture. A better understanding of the actual architecture can be gained by applying architecture recovery algorithms to the source code of the system. Unfortunately, there is little knowledge regarding the quality of such algorithms, mostly because, in order to assess their accuracy, one needs updated, detailed architectures for a variety of software systems. To overcome such problem, we propose a simulation model that synthesizes implementation-level dependency graphs that conform to the module view of any given reference architecture. We show, using observations from network theory, that the graphs are very similar to dependency graphs extracted from the source code of real software systems. Then, we synthesize thousands of graphs, in a controlled way, and use them as benchmarks for six well-known architecture recovery algorithms: ACDC, Bunch, SL75, SL90, CL75, and CL90. By comparing the given reference architectures with those recovered by the algorithms, we conclude that ACDC and Bunch outperform the alternatives, specially if the architecture contains more than a couple of modules.

1 Introduction

Architectural drift

Reverse engineering / Architecture recovery algorithms

2 Background

Architecture recovery algorithms

Evaluation of architecture recovery algorithms.

3 The BCR+ model

TODO: Talk about network theory / complex networks / scale-free networks

When designing the architecture of a software system, the module viewpoint plays a major role, by specifying the modules of the system and their interrelationships. Constraining such dependencies benefits the maintenatibility, portability, and reusability of the software system.

In this section, we describe the BCR+ model for synthesizing graphs from a description of the module viewpoint of an arbitrary architecture — that is, the modules and dependencies between modules. The BCR+ model produces—by means of the mechanisms of growth and preferential attachment—directed graphs that are both scale-free and segmented in modules.

TODO: there are other models, but they don't take architecture as input

3.1 Overview

The BCR+ model is a generalization of a graph model proposed by Bollobás et al [3]. The original model generates directed graphs that are scale-free. Our model extends the original by generating graphs in which the nodes are organized in modules, according to an modular architecture given as input.

The BCR+ model takes the following parameters as input:

- number of vertices, n;
- three probability values, p_1, p_2 e p_3, com $p_1 + p_2 + p_3 = 1$;
- base in-degree, δ_{in};
- base out-degree, δ_{out}.
- an directed graph representing dependencies between modules, G;
- a constant, μ, with $0,0 \leq \mu \leq 1,0$;

The last two parameters are new in BCR+. The other ones are from the original model by Bollobás et al [3].

In the graph G, each vertex represents one module of the architecture. Each edge defines a relationship of dependency between modules. We say that a module M_1 depends on another module, M_2, if G contains an edge from the vertex that represents M_1 to the vertex that represents M_2. In the graph that is created, an edge from a vertex $v_1 \in M_1$ to another vertex, $v_2 \in M_2$, can be created only if M_1 depends on M_2 in the graph G or if M_1 and M_2 are the same module.

TODO: Say that these graphs contain both internal and external edges. Define internal/external.

The parameter μ controls the proportion of external edges in the graph—that is, edges connecting vertices in distinct modules. Lower values lead to graphs with fewer external edges.

The original model by Bollobás et al [3] is a particular case of BCR+ when $\mu = 0$ and G contains one single vertex, representing one single module.

As a computer model, BCR+ takes the parameters as inputs and, by running an algorithm, outputs a graph. The algorithm builds the output graph incrementally. It starts by creating a module for each vertex in the input graph G, and then adding one vertex to each module. After that, it creates all external edges that are allowed by G (see Figure 1). Then, the graph is modified according to three formation rules that are applied successively, in random order, until the graph grows to n vertices. At each algorithm step, the probability of the i-th being applied is given by the parameter p_i.

3.2 Formation Rules

There are three formation rules in BCR+. Each one modifies the output graph by adding or removing vertices or edges in the graph. Also, each formation rule has a probability of being applied, given by the parameters p_1, p_2 and p_3.

The rules are illustrated in Figure 2. Simplified rules:

- Rule 1: one vertex is added to some module, together with an outgoing edge to another vertex in the same module.
- Rule 2: one vertex is added to some module, together with an ingoing edge coming from another vertex in the same module.
- Rule 3: one edge is added between two preexisting vertices. There are two variations of this rule:
  - Rule 3a: choose vertices from distinct modules.
  - Rule 3b: choose vertices that are in the same module.

The rules 1, 2, and 3b come directly from the original model by Bollobás et al [3]. The rule 3a was created in BCR+ to account for inter-module dependencies.

The choice of vertices to which add edges to is done according to preferential attachment. When we say that a vertex "chosen according to $\mathrm{f}(x)$", we mean that the probability of choosing the vertex x is proportional to $\mathrm{f}(x)$:

$$\mathrm{P}(x) \;=\; \frac{\mathrm{f}(x)}{\sum_i \mathrm{f}(i)}$$

The denominator is a normalization factor, such as the sum of probabilities $\mathrm{P}(x)$ is 1.

With this definition in mind, the rules can be fully specified:

- Rule 1: *Add a vertex with an outgoing edge.* An existing vertex, w, is chosen according to $\mathrm{f}(x) = \delta_{in} + \mathrm{g}_{in}(x)$ (that is, parameter δ_{in} added to the vertex in-degree). A new vertex, v, is added to the module that contains w, together with an edge from v to w.
- Rule 2: *Add a vertex with an ingoing edge.* An existing vertex, w, is chosen according to $\mathrm{f}(x) = \delta_{out} + \mathrm{g}_{out}(x)$. A new vertex, v, is added to the module that contains w, together with an edge from w to v.

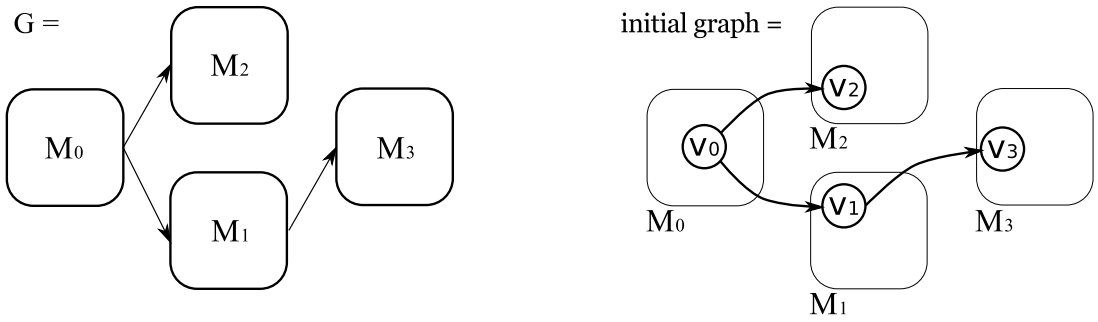

Figure 1: Initial graph synthesized by BCR+, given an input graph parameter G.

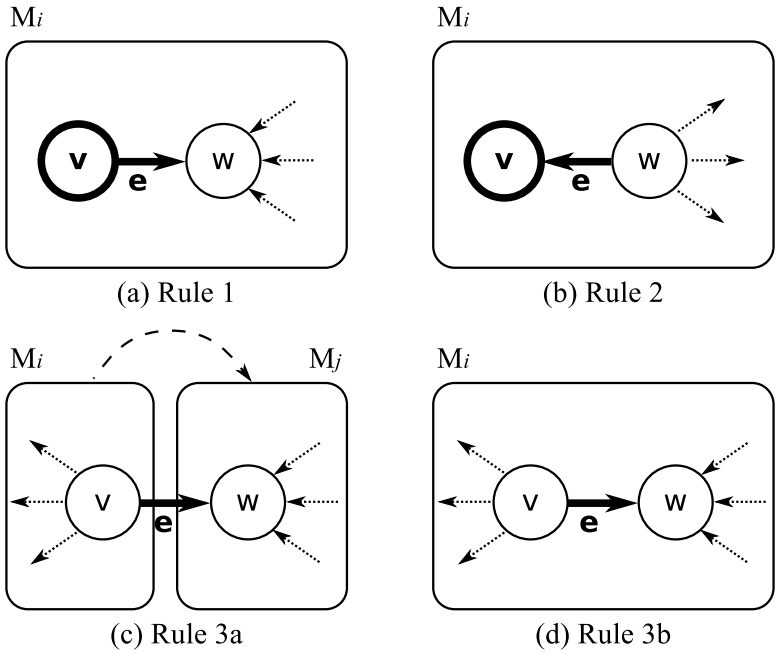

Figure 2: Formation rules for BCR+. M_i and M_j are distinct modules, such as M_i depends on M_j. In the diagram for each rule, thicker lines represent vertices and edges created when the rule is applied.

- Rule 3: *Add an edge between pre-existing vertices.* A vertex, v, is chosen according to $\mathrm{f}(x) = \delta_{out} + \mathrm{g}_{out}(x)$. Then, an edge is added from v to another vertex, w, chosen according to $\mathrm{f}(x) = \delta_{in} + \mathrm{g}_{in}(x)$. The vertex w is not chosen among the set of all vertices. In fact, there are two rules, and the choice of the rule to apply is probabilistic and depends on the parameter μ:
  - Rule 3a: with proability μ, w is chosen among the vertices that are in modules on which the module of v depends, according to the parameter G.
  - Rule 3b: with probability $1 - \mu$, w is chosen among the vertices that are in the same module as v.

TODO: BCR+ is a growth model, can simulate the evolution of a software system subject to constraints in module interaction. See CSMR paper (extended version), pg 2, col 2, just before section B.

3.3 Example

TODO: Do we need an Example section?

4 Software-Realism

If synthetic graphs are to be used as surrogates to dependency graphs extracted from the source code of software systems, it is expected that they resemble such graphs. To capture this software quality of a graph, we define the concept of software-realism, that indicates to what extent a graph resembles software dependency graphs.

A metric for software-realism should be consistent, that is, it should present high values when measuring dependency graphs, and low values when measuring graphs from other domains—e.g., protein interaction networks, social networks etc.

First, we define a metric for software-realism, based on structural properties of a graph. Then, we show that the metric is consistent, by applying both to software dependency graphs and to graphs from non-software domains. Finally, we show that the BCR+ model is capable of synthesizing graphs with a high degree of software-realism.

4.1 Background

Given three vertices, one can conceive 13 distinct connected directed graphs—the so-called triads—, as shown in Figure 3. By counting how often each triad appears in a graph, one can build a vector, called triad concentration profile, that summarizes any graph in a compact, fixed-size way.

Previous work has shown that graphs from the same domain tend to be caracterized by similar triad concentration profiles [8]. For example, Figure 3(a) presents triad concentration profiles for two software dependency graphs, whereas Figure 3(b) presents triad concentration profiles for graphs in distinct domains: a software dependency graph and a linguistic graph. An informal analysis reveals that the similarity between the profiles is greater in the first case. It should suffice to notice that, in the second case, the concentration of the first two triads is somewhat reversed (in the linguistic graph, the second triad is the most frequent).

Previous works [7, 6, 5] have explored the use of Pearson's correlation coefficient on triad concentration profiles in order to find clusters of structurally similar graphs. The results suggest that the technique is effective in identifying groups of graphs from the same domain (e.g., social, linguistic, biologic etc.).

4.2 Software-Realism Metric

TODO: Similarity between graphs:

$$\mathrm{sim}(a, b) \;=\; \mathrm{cor}(\mathrm{PCT}(a), \mathrm{PCT}(b)).$$

TODO: Software-realism metric:

$$\mathrm{S}(x, R) \;=\; \frac{\sum_{s \in R} \mathrm{sim}(x, s)}{|R|}.$$

4.3 Metric Evaluation

TODO: Classification model:

$$\mathrm{m}(x, R, S_0) \;=\; \begin{cases} \text{sw-realistic}, & \text{se } \mathrm{S}(x, R) \geq S_0; \\ \text{non sw-realistic}, & \text{caso contrário.} \end{cases}$$

TODO: data collection; data processing (extracting dependency graphs); analysis (accuracy, precision, recall)

TODO: choice of S_0.

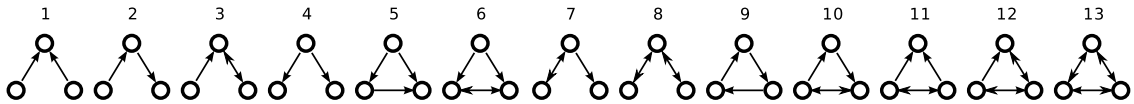

Figure 3: All triads—connected directed graphs with three vertices—, numbered from 1 to 13.

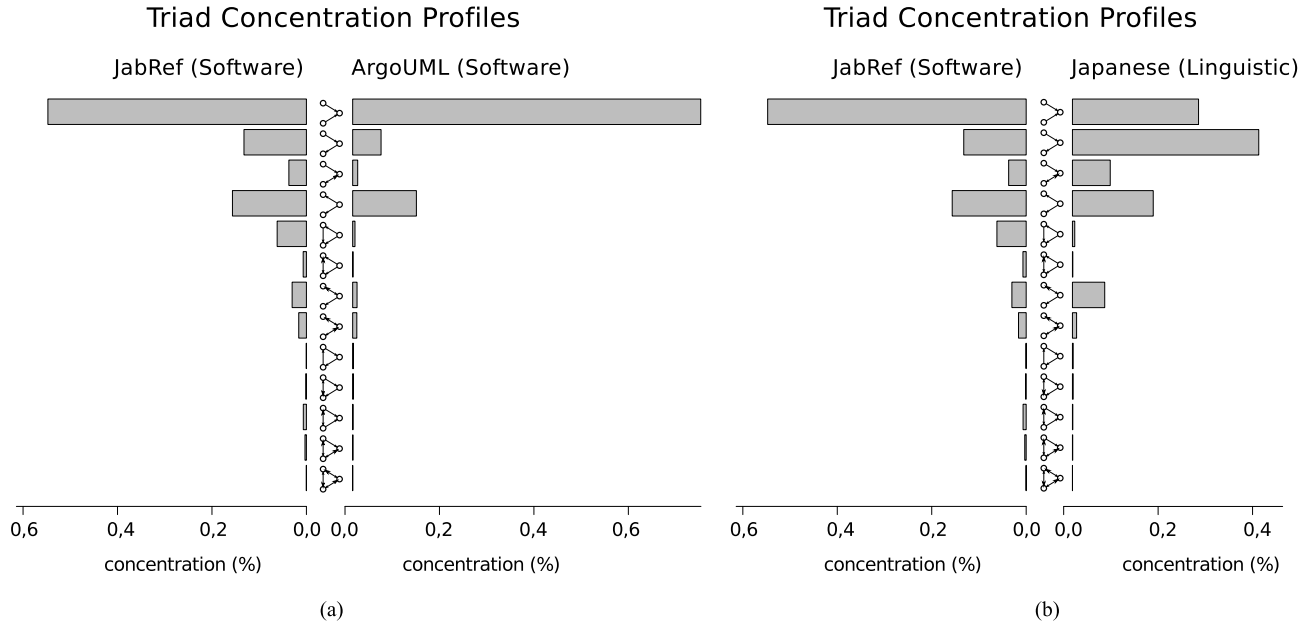

Figure 4: Comparison between triad concentration profiles for three distinct graphs (computed with the igraph tool [4]). (a) Class dependency graphs for two softwares: JabRef, version 2.5b2 (left) and ArgoUML version 0.28 (right). (b) Graph for JabRef, version 2.5b2 (left), and graph of word succession in a sample of Japanese texts (right) [7].

4.4 Software-Realism of BCR+

TODO: choice of parameters; graph synthesizing; graph classification (according to sw-realism)

5 Evaluation of Architecture Recovery Algorithms

TODO: Clustering

TODO: Brief description about Bunch, ACDC, SL75/SL90, CL75, CL90

TODO: Traditional experimental setup for evaluation of architecture recovery algorithms: given a dependency graph and a reference architecture, run the algorithm on the graph and compare the recovered architecture with the reference architecture.

TODO: MoJo metric.

5.1 Experimental Setup

TODO: Section 6.3 from dissertation

5.2 Comparison of Architecture Recovery Algorithms

The first experiment was aimed at comparing the performance of the algorithms when applied software-realistic graphs, measured by the similarity between the clusters found by the algorithms and the modules of the reference architecture.

We have selected for this experiment only the graphs that were classified as software-realistic, totalizing about 6,000 graphs. Then, each algorithm was applied to each one of the synthetic graphs, resulting in a set of clusters together with a mapping from the set of vertices to the set of clusters. The performance of the clusterings were measured with the metric MoJoSim.

The Figure 5 shows a boxplot of the MoJoSim values that were measured for each algorithm. The dashed lines indicate the maximum and minimum value from the data. Comparing the median performance of the algorithms, it is clear that ACDC presents the best performance, followed by Bunch, and then by the hierachical algorithms. These differences were verified to be significative by applying a paired Wilcoxon test ($\alpha = 0.05$), with Bonferroni correction to account for multiple hypothesis testing. Among the hierarchical algorithms, there was no evidence that their performances differ.

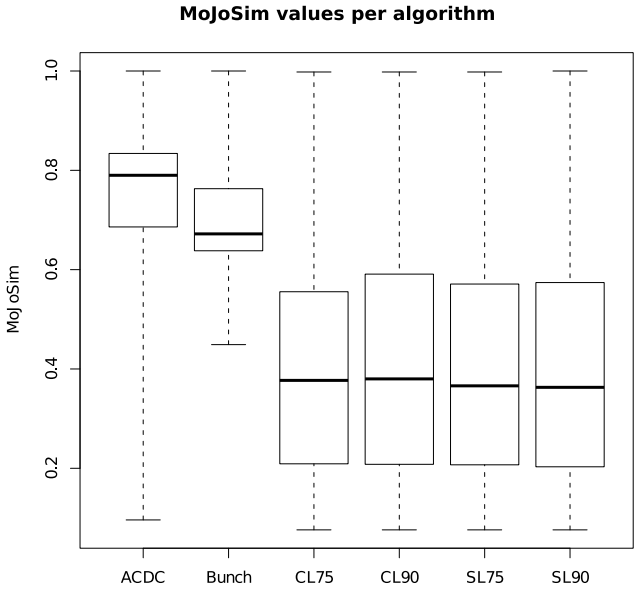

Figure 5: Statistical summary of MoJoSim values for each architecture recovery algorithm.

Another noteworthy aspect of the data is the dispersion of values. The algorithm Bunch presents the smaller dispersion, with more that half of the values between 0.60 and 0.80 (minimum value = 0.45). In the case of ACDC, half of the values are between 0.65 a 0.85, but the minimum value is 0.01. This observation suggests that, although Bunch presents overall inferior performance when compared with ACDC, it can be a resonable choice for yielding more predictable results.

TODO: move to a discussion section. The results disagree with conclusions from Wu, Hassah, and Holt [9]. They have concluded that the SL algorithms outperform ACDC, which outperforms Bunch, which outperforms the CL algorithms. The disagreement probably can be explained by the criteria they used to build the reference architecture for each software system. They use the folder structure from the source of the software system to determine the modules in the architecture reference. In our work, the reference architecture is defined a priori, and the graphs are synthesized ro reduce the dependencies between modules.

We should note that this experiment suffers from sampling bias. Although all analyzeds graphs are software-realistic, we can not guarantee that it is representative of all software systems. We see this

analysis as complementary to similar experiments, that suffer from other types of bias [9, 2, 1].

5.3 Algorithm Parameters Analysis

In the next experiment, we tried to understand how the algorithms behave on a variety of situations—graphs with different number of modules, different proportions of external edges etc. We were able to control a number of graph properties by changing parameter values for the BCR+ model.

In this experiment, all graphs—both software-realistic and non software-realistic—were considered, in order to avoid the software-realism variable to bias the results.

We have analyzed the influence of the number of modules and of the proportion of external edges on the performance of the algorithms. We have chosen these two properties because they map naturally to BCR+ parameters: number of modules is controlled by the input module graph, G, and proportion of external edges is controlled by the parameter μ.

TODO: explain that the sample is paired, that is, there are two graphs that differ only by μ (the other parameters are identical).

TODO: this experiment is not possible with traditional approaches, since it depends on having control on the graphs properties.

As expected, all algorithms performed better when the proportion of external edges was smaller. **TODO:** is it true? Check experimental package

A surprising result was found when analyzing the influence of the number of modules on the performance of the algorithms (see Figure 6). While the performance of ACDC and Bunch are not influenced by the number of modules, the performance of the hierachical algorithms is worse when applied to graphs with more modules. In fact, the performance of such algorithms are comparable to the performance of ACDC and Bunch when there are only two modules, but it gets progressively worse as the number of modules grows.

A possible explanation for this phenomenom can be found in the distribution of module sizes found by hierarchical algorithms. It is common that they found large modules, sometimes covering more than half of the vertices [9]. If the reference architecture contains many smaller modules, large modules found by an algorithm are severely penalyzed by the MoJoSim metric (the large modules need to be split in smaller modules, which is accounted in MoJoSim as a sequence of vertex movimentations).

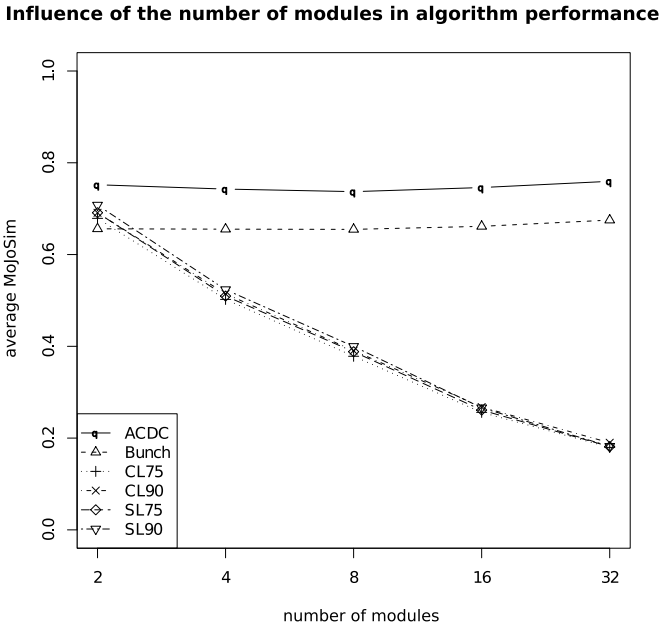

Figure 6: Influence of the number of modules in the reference architecture on the performance of each architecture recovery algorithm.

5.4 Impact of Bidirectional Dependencies

The third experiment was aimed at measuring the influence of the type of dependency (unidirectional of bidirectional) on the performance of each architecture recovery algorithm. The independent variable is the parameter G from BCR+, that is, the definition of the modules and allowed dependencies between modules.

We have analyzed two configurations for G:

- *bidirectional*: G contains two modules, M_1 and M_2; there is a bidirectional edge between M_1 and M_2 (that is, vertices in M_1 may depend on vertices of M_2 and the other way around);
- *unidirectional*: G contains two modules, M_1 and M_2; there is an unidirectional edge from M_1 to M_2 (that is, vertices in M_1 may depend on vertices of M_2, but the opposite is not true).

As far as we know, BCR+ is the only graph model that can be used to conduct this type of experiment.

Other graph models do not offer control over specific dependencies.

For the remaining parameters of BCR+, when considered the values described in Section XXX. For each configuration of parameters, three graphs were synthesized, totalizing 11,400 graphs. Then, each architecure recovery algorithm was applied to each graph. Finally, the performance was computed, using the metric MoJoSim.

The results are shown in a boxplot in Figure 7. It can be observed that all algorithms performed slightly better on the graph with the unidirectional edge. All differences—except that of CL75—are significative at 5% level, as assessed with the paired Wilcoxon test.

The difference of performance for each algorithm is shown in Table 1, as a 95% confidence interval. ACDC and SL75 are more influenced by the choice of dependency type than the other algorithms. Overall, the difference between MoJoSim values does not exceed 0.03 (except in the case of SL90), which is a small value.

We conclude that, in general, the presence of modules with mutual, bidirectional dependencies, influences negatively the performance of the algorithms that were analyzed. The observed difference was small in our study with two modules, but we suspect that this difference can be greater on graphs with more modules and more bidirectional dependencies.

5.5 Discussion

We have described three experiments based on the application of architecture recovery algorithms to synthetic graphs created by the model BCR+. The experiments present evidence that this approach is practicable and can be used to find insights about the algorithms that cannot be found with more traditional approaches.

TODO: more

6 TODO: Other Possible Sections

Discussion
- Limitations
- Conclusion
- Acknowledgments
- References

References

[1] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

[2] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering CSMR '09*, pages 251–254. IEEE Computer Society, 2009.

[3] Béla BollobÁs, Christian Borgs, Jennifer Chayes, and Oliver Riordan. Directed scale-free graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms SODA '03*, pages 132–139. Society for Industrial and Applied Mathematics, 2003.

[4] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[5] Zhang Lin, Qian GuanQun, and Zhang Li. Network motif & triad significance profile analyses on software system. *W. Trans. on Comp.*, 7:756–765, June 2008.

[6] Yutao Ma, Keqing He, and Jing Liu. Network motifs in object-oriented software systems. *DISCRETE AND IMPULSIVE SYSTEMS*, 14:166, 2007.

[7] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, March 2004.

[8] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.

[9] J. Wu, A. E. Hassan, and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance ICSM '05*, pages 525–535, 2005.

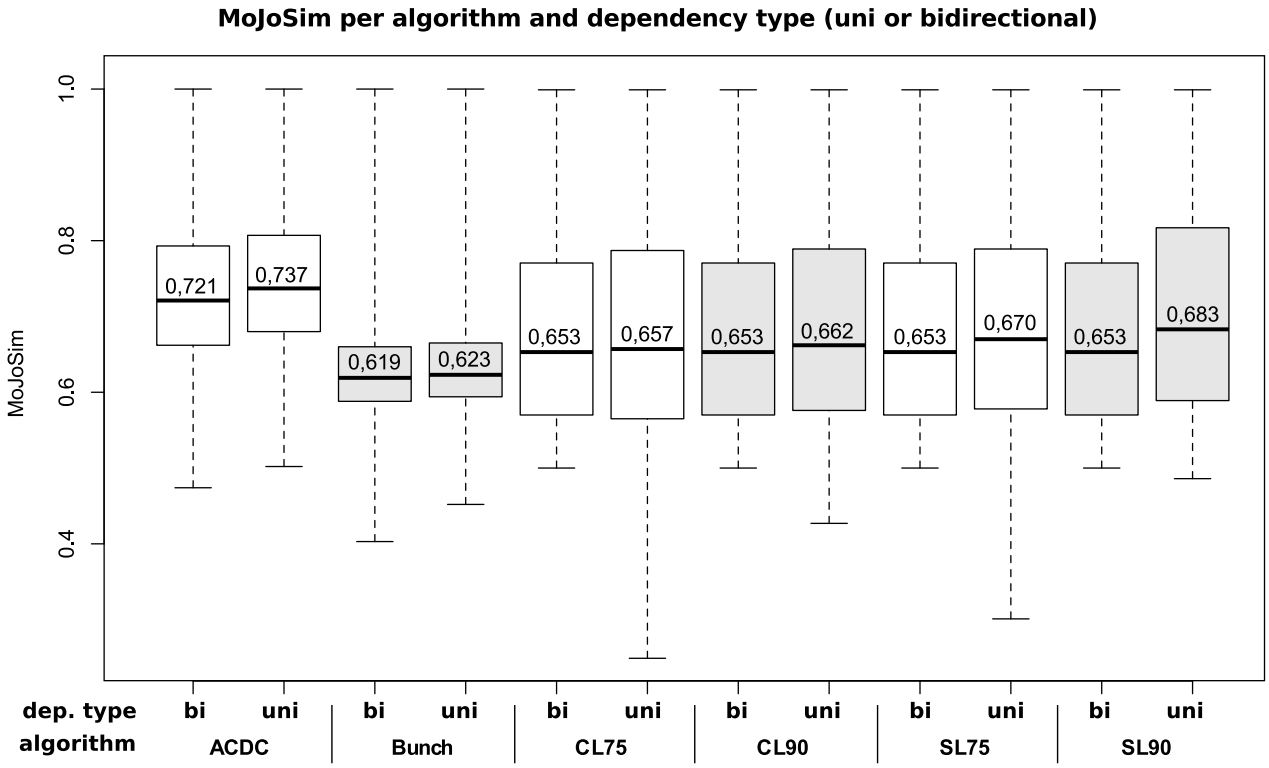

Figure 7: Influence of the type of dependency between modules (unidirectional ou bidirectional) on the performance of each architecture recovery algorithm.

ACDC	Bunch	CL75	CL90	SL75	SL90
[0,016; 0,023]	[0,004; 0,008]	[-0,004; 0,006]	[0,006; 0,015]	[0,007; 0,017]	[0,021; 0,031]

Table 1: Intervalo de confiança de 95% para a diferença de desempenho de cada algoritmo, medido em MoJoSim, entre a configuração *simples* e a configuração *dupla* do parâmetro G.