Representativeness in Software Engineering Research

September 5, 2012
Technical Report
MSR-TR-2012-93

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Representativeness in Software Engineering Research

Meiyappan Nagappan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Kingston, Canada
mei@cs.queensu.ca

Thomas Zimmermann
Microsoft Research
Redmond, WA, USA
tzimmer@microsoft.com

Christian Bird
Microsoft Research
Redmond, WA, USA
Christian.Bird@microsoft.com

***Abstract*—One of the goals of software engineering research is to achieve generality: Are the phenomena found in a few projects reflective of what goes on in others? Will a technique benefit more than just the projects it is evaluated on? The discipline of our community has gained rigor over the past twenty years and is now attempting to achieve generality through evaluation and study of an increasing number of software projects (sometime *hundreds!)*. However, quantity is not the only important component. Selecting projects that are representative of a larger body of software of interest is just as critical. Little attention has been paid to selecting projects in such a way that generality and representativeness is maximized or even quantitatively characterized and reported. In this paper, we present a general technique for quantifying how representative a sample of software projects is of a population across many dimensions. We also present a greedy algorithm for choosing a maximally representative sample. We demonstrate our technique on research presented over the past two years at ICSE and FSE with respect to a population of 20,000 active open source projects. Finally, we propose methods of reporting objective measures of representativeness in research.**

I. Introduction

Over the past twenty years, the discipline of software engineering research has grown in maturity and rigor. Researchers and our community in general have worked towards maximizing the impact that software engineering research has on practice. One such way is by providing techniques and results that are as general (and thus useful) as possible. However, achieving generality is challenging. Basili et al. remarked that "general conclusions from empirical studies in software engineering are difficult because any process depends on a potentially large number of relevant context variables" [1].

With the availability of OSS projects, the software engineering research community has called for more extensive validation and researchers have answered the call. As an extreme example, the study of Smalltalk feature usage by Robbes et al. [2] examined 1,000 projects! Another example is the study by Gabel and Su that examined 6,000 projects [3]. But if care isn't taken when selecting which projects to analyze, then increasing the sample size doesn't actually contribute to the goal of increased generality. *More is not necessarily better*.

As an illustration, consider a researcher that has a technique to evaluate or a hypothesis to investigate and wants to include a large number of projects in an effort to demonstrate generality and therefore impact. To choose the subject programs, the researcher goes to the json.org website, which lists over twenty JSON parsers that are all readily available for download and analysis. While the researcher may select and evaluate twenty distinct projects, they all fall within a very narrow range of functionality (and are likely similar in terms of structure, size, running time, etc.). Although such an evaluation would include many projects, generality would not be achieved because the projects studied are not representative of a larger and wider population of software projects. We would learn about JSON parsers, but little about other types of software. This extreme illustration is contrived and no serious researcher would select projects in such a way. Nonetheless, the majority of studies that select projects for evaluation or investigation do so in a fairly subjective or non-obvious objective way. One likely reason for this is that there is currently no standard method of selecting projects for study.

Other fields such as medicine and sociology have overcome similar issues through published and accepted methodological guidelines for subject selection. For example, guidelines have been provided for the design of clinical trials which include how subjects should be selected [2] (see Section 6.3 "Patient Selection" pp. 171-174 and Section 6.6 "Design Considerations" pp. 179-182). Additionally the National Institutes of Health (NIH) in the United States have developed requirements [4] to make sure that minimum generality bars are met by requiring that certain subpopulations are included in such trials. The aim of these guidelines is to ensure that the subjects are representative of a larger population.

Software engineering research lacks such guidelines today, but we can fortunately leverage the methodological advances of these older fields. In this paper, we present techniques for measuring the representativeness of a sample, relative to a larger population along various dimensions, and for selecting a maximally representative sample from a population. Our technique is based on the theories underlying subject selection methodologies in other fields. Whereas today, researchers employ what we term *opportunistic sampling*, selecting projects that are easiest to mine or display some attribute that makes them attractive to researchers, we recommend *representative sampling*, in which a subset of a statistical population is selected that accurately reflects the members of the entire population.

In our work, we present a framework to quantify the representativeness of the sample in the population so that (a) the researcher can arrive at appropriate conclusions about the results of the experiments, (b) the research community can understand the context under which the results are applicable, and (c) the practitioners can easily identify if a particular research contribution is applicable to their software. In a way similar to the adoption of structured abstracts [5] in research papers, we

hope that researchers will use the techniques and recommendations in this paper to achieve consistent methods of reporting representativeness in their research. We make the following contributions:

1. We present a technique for objectively measuring how representative a sample set of projects are of a population of projects.
2. We present a technique for selecting a sample of projects to maximize the representativeness of a study.
3. We assess the representativeness of studies in top tier software engineering research venues and provide guidance for reporting representativeness.

In the rest of this paper, we first present a general framework for evaluating representativeness of a sample from a population of software projects and selecting a maximally representative sample (Section II). We then demonstrate this technique by calculating the representativeness of research over the past two years at ICSE and FSE (Section III). Finally, we provide appropriate methods of reporting representativeness and discuss implications (Section IV) and related work (Section V).

II. Framework

In this section, we present our framework for assessing representativeness: we first introduce our terminology (Section II.A and II.B) followed by algorithms to score the representativeness of a set of projects (Section II.C) and select the projects that increase the score the most (Section II.D).

We implemented both algorithms in the R programming language and they will be made available as an R package. The appendix contains a walkthrough on how to use our implementation.

II.A. Terminology: Universe, Space, Configuration

The ***universe*** is a large set of projects; it is often also called population. The universe can vary for different research areas. For example, research on mobile phone applications will have a different universe than web applications.

> Possible universes: all open-source projects, all closed-source projects, all web applications, all mobile phone applications, all open-source projects on Ohloh, and many others.

Within the universe, each project is characterized with one or more ***dimensions***.

> Possible dimensions: total lines of code, number of developers, main programming language, project domain, recent activity, project age, and many others.

The set of dimensions that are relevant for the generality of a research topic define the ***space*** of the research topic. Similar to universes, the space can vary between different research topics. For example, we expect program analysis research to have a different space than empirical research on productivity:

> Possible space for program analysis research: total lines of code, main programming language.
>
> Possible space for empirical research on productivity: total lines of code, number of developers, main programming language, project domain, recent activity, project age, and likely others.

The goal of representative research is typically to provide a high coverage of the space in a universe. The underlying assumption of this paper is that *projects with similar values in the dimensions*—that is they are close to each other in the space—*are representative of each other*. This assumption is commonly made in the software engineering field, especially in effort estimation research [6,7]. For each dimension *d*, we define a ***similarity function*** which decides whether two projects p_1 and p_2 are similar with respect to that dimension:

$$similar_d(p_1, p_2) \rightarrow \{\text{true; false}\}$$

The list of the similarity functions for a given space is called the ***configuration***.

$$\text{configuration } C = (similar_1, \dots, similar_n)$$

Similar to universe and space, similarity functions (and the configuration) can vary across projects. For some research topics, projects written in C might be considered similar to projects written in C++, while for other research they might be considered different.

To identify similar projects within the universe, we require the projects to be similar to each other in *all* dimensions.

$$similar(p_1, p_2) = \bigcap_d similar_d(p_1, p_2)$$

If no similarity function is defined for a dimension, we assume the following default functions, with *p[d]* the score of project *p* in dimension *d* and |*e*| the absolute (positive) value of the specified expression *e*:

- For *numeric dimensions* (e.g., number of developers): We consider two projects to be similar in a dimension if their values are in the same order of magnitude (as computed by $\log_{10}$).

  $$similar_d(p_1, p_2) \rightarrow |\log_{10} p_1[d] - \log_{10} p_2[d]| \leq 0.5$$

- For *categorical dimensions* (e.g., main programming language: We consider two projects to be similar in a dimension if the values are identical.

  $$similar_d(p_1, p_2) \rightarrow p_1[d] = p_2[d]$$

As mentioned above the similarity functions can be overridden in a configuration. Different configurations may exist for different research topics and areas.

II.B. Example: Scoring and Project Selection

Figure 1(a) shows a sample universe and a sample space: the universe contains 50 projects, each represented by a point. The space is defined by two dimensions: the number of developers (horizontal) and the number of lines of code (vertical). In practice, the universe can be thousands of projects and the space can be defined by numerous dimensions, not just two. We will present a more complex instantiation of our framework in Section III.

Consider project A in Figure 1(a) which is represented by an enlarged point. The light gray areas indicate the projects that are similar to project A in one dimension (based on the similarity functions that are defined in the configuration). The intersection of the light gray areas (the dark gray area) indicates the projects that are similar to A with respect to the entire

space. In total seven projects are similar, thus project A covers (7+1)/50=16% of the universe. We can also compute coverage for individual dimensions: project A covers 13/50=26% for number of developers and 11/50=22% for lines of code.

Figure 1(b) illustrates how a second project increases the covered space:

- If we add project B, ten additional projects are covered, the universe coverage increase to 18/50=36%. The coverage of the developer and lines of code dimensions increases to 60% and 56% respectively.
- However if we add project C instead of project B, there is only little impact on coverage. All similar projects have been already covered because project C is close to project A. Thus the coverage increases only to 18%.

This illustrates an important point: to provide a good coverage of the universe, one should *select projects that are diverse* rather than similar to each other. We now introduce algorithms to score representativeness (*score_projects*) and to select additional projects such that the score is maximized (*next_projects*).

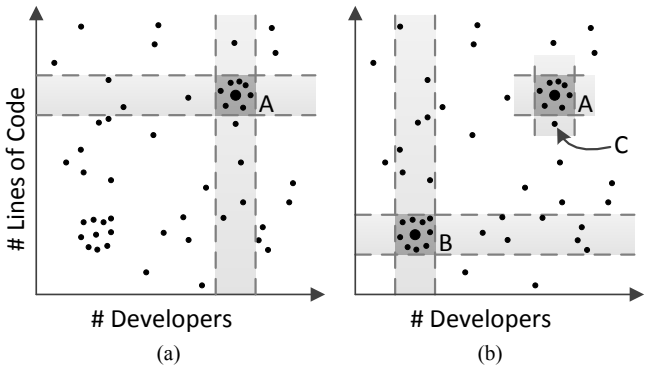

Fig. 1. Sample universe of 50 projects defined by a two-dimensional space. (a) The light gray areas indicate projects similar to project A in one dimension. The dark gray areas indicate projects similar to project A in both dimensions. (b) Project B increases the coverage of the space more than project C does, because C is too similar to projects already covered by project A.

II.C. Scoring Representativeness

We score the representativeness of a set of projects P for a given universe U, an n-dimensional space D, and a configuration $(similar_1, \dots, similar_n)$ as follows. (Recall the definition of similar which is $similar = similar_1 \wedge \dots \wedge similar_n$)

$$score = \frac{\left|\bigcup_{p \in P} \{q | similar(p,q)\}\right|}{|U|}$$

As discussed before, research topics can have different parameters for universe, space, and configuration. Therefore it is important to not just report the score but also the context in which it was computed: What projects is the research intending to be representative of (universe)? Based on what criteria (space, configuration)?

To compute the score for a set of projects, we implemented the algorithm shown in Algorithm I in R. For each project $p \in$ ject $p \in P$, the algorithm computes the set of projects *c_project* that are covered by p (Lines 3-10). As a naming convention we use the prefix *c_* in variable names for sets of covered projects.

ALGORITHM I. SCORING PROJECTS

score_projects(projects P, universe U, space D, config C):
1: $c\_space \leftarrow \emptyset$
2: $c\_dim \leftarrow [\emptyset, \dots, \emptyset]$
3: for each project $p \in P$:
4: $\quad c\_project \leftarrow U$
5: $\quad$ for each dimension $d \in D$:
6: $\quad\quad are\_similar(p,q) \leftarrow C[d](p,q)$
7: $\quad\quad sim\_projects \leftarrow \{q | are\_similar(p,q)\}$
8: $\quad\quad c\_project \leftarrow c\_project \cap sim\_projects$
9: $\quad\quad c\_dim[d] \leftarrow c\_dim[d] \cup sim\_projects$
10: $\quad c\_space \leftarrow c\_space \cup c\_project$
11: $score \leftarrow |c\_space|/|U|$
12: $dim\_score \leftarrow$ apply$(c_dim, X \rightarrow |X|/|U|)$
13: return $(score, dim\_score)$

ALGORITHM II. SELECTING THE NEXT PROJECTS

next_projects(K, projects P, universe U, space D, config C):
1: $result \leftarrow [\,]$
2: $similar(p,q) = C[1](p,q) \wedge \dots \wedge C[d](p,q)$
3: $c\_space \leftarrow \bigcup_{p \in P}\{q | similar(p,q)\}$
4: $candidates \leftarrow U - P$
5: for $i \in \{1, \dots, K\}$:
6: $\quad c\_best \leftarrow \emptyset$
7: $\quad p\_best \leftarrow$ NA
8: $\quad$ for each candidate $p \in candidates$:
9: $\quad\quad c\_candidate \leftarrow \{q | similar(p,q)\}$
10: $\quad\quad c\_new \leftarrow (c\_space \cup c\_candidate) - c\_space$
11: $\quad\quad$ if $|c\_new| > |c\_best|$:
12: $\quad\quad\quad c\_best \leftarrow c\_new$
13: $\quad\quad\quad p\_best \leftarrow p$
14: $\quad$ if $p\_best =$ NA:
15: $\quad\quad$ break
16: $\quad result \leftarrow$ append$(result, p_best)$
17: $\quad candidates \leftarrow candidates - \{p\_best\}$
18: $\quad c\_space \leftarrow c\_space \cup c\_best$
19: return $(result)$

In addition, the algorithm computes the projects *c_dim[d]* covered by each dimension *d* (Line 9). After iterating through the set P, the algorithm computes the representativeness score within the entire space (Line 11) and for each dimension (Line 12). The *apply* function maps the function $X \rightarrow |X|/|U|$ to the vector *c_dim* and returns a vector with the result.

II.D. Project Selection

In order to guide project selection in such a way that the representativeness of a sample is maximized, we implemented the greedy algorithm that is shown in Algorithm II. The input to the algorithm is the number K of projects to be selected, a set of already selected projects P, a universe U, an n-dimensional space D, and a configuration $C = (similar_1, \dots, similar_n)$.

The algorithm returns a list of up to K projects; the list is ordered decreasingly based on how much the projects increase the coverage of the space. The set of preselected projects P can

be empty. By calling the algorithm with $P = \emptyset$ and $K = |U|$ one can order the entire universe of projects based on their coverage increase and returns the subset of projects that is needed to cover the entire universe (for a score of 100%).

The main part of the algorithm is the loop in Lines 5-18 that is repeated at most *K* times. The loop is exited early (Lines 14-15) when no project is found that increases the coverage; in this case the entire universe has been covered (score of 100%). The algorithm maintains a candidate set of projects (*candidates*), which is initialized to the projects in universe *U* but not in *P* (Line 4, we use $-$ to denote set difference). The body of the main loop computes for each candidate $p \in candidates$ (Lines 8-13) how much its coverage (Line 9) would increase the current coverage *c_space* (Line 10) and memorizes the maximum increase (Lines 11-13). At the end of an iteration *i*, the project *p_best* with the highest coverage increase is appended to the result list and then removed from the candidates list (Lines 16-17); the current coverage *c_space* is updated to include the projects in *c_best* (Line 18).

Our R implementation includes several optimizations that are not included in Algorithm I for the sake of comprehension. To reduce the cost of set operations we use index vectors in R (similar to bit vectors). Computing the projects similar to a candidate in Line 9 is an expensive operation and we therefore cache the results across loop iterations. Lastly, starting from the second iteration, we do process candidates in Line 10 in *decreasing order* of their $|c\_new|$ values from the previous iteration. The $|c\_new|$ values from iteration $i-1$ are an upper bound of how much a candidate can contribute to the coverage in iteration i. If the current best increase $|c\_best|$ in iteration i is greater or equal than the previous increase $|c\_new|$ of the current candidate in iteration $i-1$, we can exit the inner loop (Lines 8-13) and skip the remaining candidates. This optimization significantly reduces the search space for projects.

II.E. Implementation in R

Note to reviewers: We are in the process of releasing the R implementation as open source. All code from Microsoft requires review before release and we expect the R package to be available in September/October 2012. We will send the URL of the code to the PC chairs once it is available.

III. INSTANTIATION

In this section we provide an example of an instantiation of our framework and illustrate how it can be used to quantify the representativeness of software engineering research.

III.A. The Ohloh Universe

We chose as *universe* the active projects that are mapped by the Ohloh platform [8]. Ohloh is a social coding platform that collects data such as main programming language, number of developers, licenses, as well as software metrics (lines of code, activity statistics, etc.). Note that the Ohloh data is just one possible universe and there are many other universes that could be used for similar purposes.

To collect data to describe the projects in the universe, we used the following steps:

1. We extracted the identifiers of *active* projects using the *Project* API of Ohloh. We decided to include only the active projects in the universe because we wanted to measure representativeness for ongoing development. We followed Richard Sands' definition [9] of an active project, that is, a project that had at least one commit and at least 2 committers in the last 12 months.
2. For each project identifier, we extracted three different categories of data (each with one call to the API). The first is the *Analysis* category which has data about main programming language, source code size and contributors. The second is the *Activity* category which summarizes how much developers have changed each month (commits, churn). We accumulated the activity data for the period of June 2011 to May 2012. Finally, we collected what is called the *Factoid* category. This category contains basic observations about projects such as team size, project age, comment ratio, and license conflicts.
3. We aggregated the XML files returned by the Ohloh APIs and converted them into tab-separated text files using a custom script. We removed projects from the universe that had missing data (156 projects had no main language or an incomplete code analysis) or invalid data (40 projects had a negative number for total lines of code).

After selecting only active projects and removing projects with missing and invalid data, the universe consists of a total of 20,028 projects. This number is comparable to the number of active projects reported by Richard Sands [9].

III.B. The Ohloh Space

We use the following dimensions for the *space*. The list of dimensions is inspired by the comparison feature in Ohloh. The data for the dimensions is provided by Ohloh.

- *Main language.* The most common programming language in the project. Ohloh ignores XML and HTML when making this determination.
- *Total lines of code.* Blank lines and comment lines are excluded by Ohloh when counting lines of code.
- *Number of contributors (12 months).* Contributors with at least one commit in the last 12 months.
- *Number of churn (12 months).* Number of added and deleted lines of code, excluding comment lines and blank lines, in the last 12 months.
- *Number of commits (12 months).* Commits made in the last 12 months.
- *Project age.* The Ohloh factoid for project age: projects less than 1 year old are Young, between 1 year and 3 years they are Normal, between 3 and 5 years they are Old, and above 5 years they are Very Old.
- *Project activity.* The Ohloh factoid for project activity: if during the last 12 calendar months, there were at least 25% fewer commits than in the prior 12 months, the activity is Decreasing; if there were 25% more commits, the activity is Increasing; otherwise the activity is Stable.

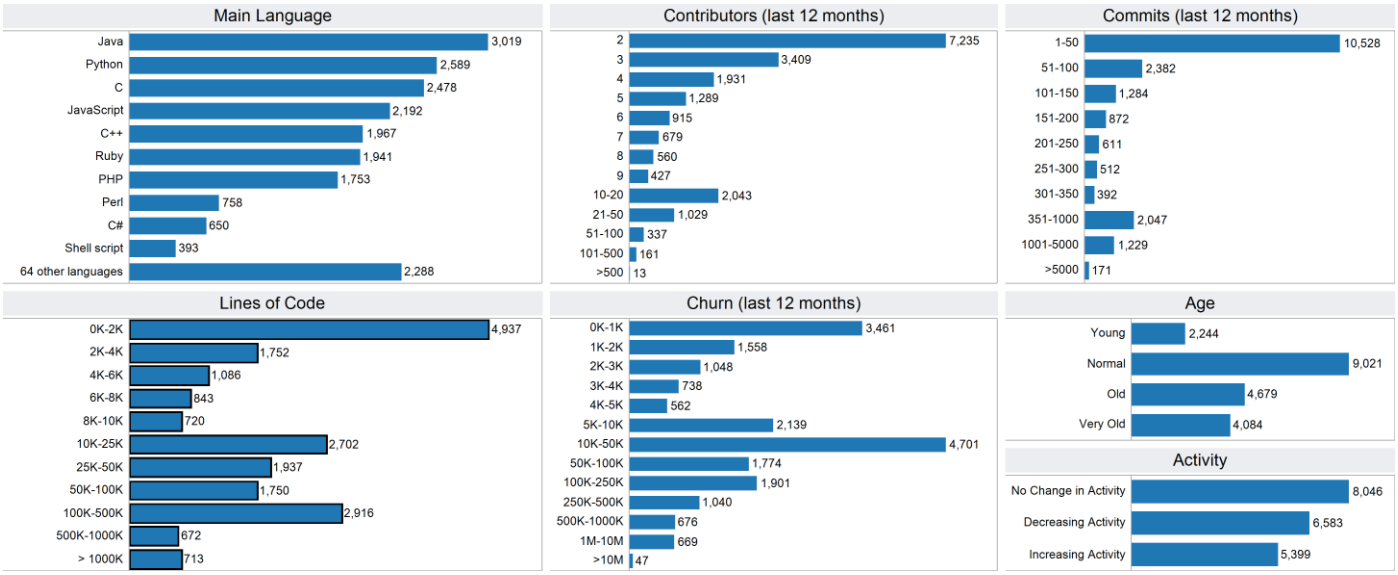

Fig. 2. Histograms of the dimensions in the Ohloh universe.

In our case, metrics for the last 12 months are for the period of June 2011 to May 2012. Again this is just one possible space and there will be other dimensions that can be relevant for the generality of research.

Figure 2 shows the distributions of the dimensions in our dataset. There are over 70 programming languages captured in the Ohloh dataset; the most frequently used languages are Java, Python, C, and JavaScript. A large number of projects are very small in terms of size, people, and activity: 4,937 projects are less than 2,000 lines of code; yet 713 projects exceed a million lines of code. Many projects have only 2 contributors (7,235 projects) and not more than 50 commits (10,528 projects) in the last 12 months. Again there are extreme cases with hundreds of contributors and thousands of commits.

III.C. Covering the Ohloh Universe

As a first experiment, we computed the set of projects required to cover the entire population of 20,028 Ohloh projects. For this we called the *next_projects* algorithm with *N=20,028* and an empty initial project list *P*.

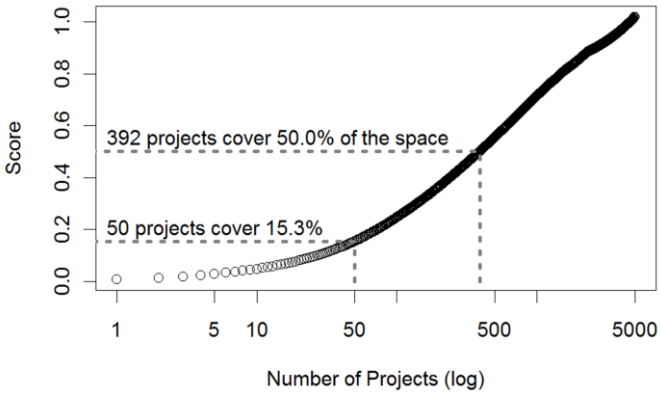

Fig. 3. Number of projects that are needed to cover the Ohloh universe. Each point in the graph means that *x* projects can cover *y* percent of the universe.

$$\textbf{next\_projects}(N = 20028, \text{projects } P = \emptyset, \text{universe } U = \text{ohloh}, \text{space } D, \text{config } C)$$

Figure 3 shows the results with a cumulative sum plot. Each point *(x,y)* in the graph indicates that the first *x* projects returned by *next_projects* covered *y* percent of the Ohloh universe. The first 50 projects (or 2.5%) covered 15.3% of the universe, 392 projects covered 50%, and 5030 projects covered the entire universe.

In Table I we show the first 15 projects returned by the algorithm *next_projects*. These are the projects that increase the coverage of the space the most. We draw the following conclusions. First, small software projects written in dynamic languages dominate the list (seven of the first nine are in Ruby or Python and under 2000 LOC). Are researchers exploring the problems faced by these projects? Even when considering all 15 projects, these projects together comprise less than 200,000 LOC and just over 1,000 commits, an order of magnitude lower than for Apache HTTP, Mozilla Firefox, or Eclipse JDT. The time and space required to analyze or evaluate on these projects are fairly low, providing a ripe opportunity for researchers to achieve impact without large resource demands. This result also counters a common criticism of software engineering: research: many people expect that any research has to scale to large-scale software. However, as Table I and Figure 1 show, the space of smaller projects is non-negligible.

III.D. Covering the Ohloh Universe with ICSE and FSE

We now apply the framework instantiated with the Ohloh universe to papers from premiere conferences in the software engineering field: the International Conference on Software Engineering (ICSE) and Foundations of Software Engineering (FSE).

To create the dataset we considered the last two years (ICSE 2011, 2012 and FSE 2010, 2011). The first author read each (full) paper of the main technical research track in each

TABLE I. THE FIRST 15 PROJECTS RETURNED BY **next_projects**($N = 20028$, PROJECTS $P = \emptyset$, UNIVERSE $U = ohloh$, SPACE D, CONFIG C) WITH THE INCREASE IN COVERAGE

Name	Language	Lines	Contributors	Commits	Churn	Age	Activity	Increase
serialize_with_options	Ruby	301	2	10	147	Normal	Increasing	0.574%
Java Chronicle	Java	3892	4	81	8629	Young	Stable	0.569%
hike	Ruby	616	3	11	333	Normal	Stable	0.559%
Talend Service Factory	Java	20295	8	162	27803	Normal	Stable	0.549%
OpenObject Library	Python	1944	5	36	1825	Normal	Stable	0.459%
ruote-amqp-pyclient	Python	315	4	7	139	Normal	Stable	0.454%
sign_server	Python	1791	3	63	3415	Young	Stable	0.414%
redcloth-formatters-plain	Ruby	655	4	5	82	Normal	Decreasing	0.384%
python-yql	Python	1933	2	11	93	Normal	Decreasing	0.369%
mraspaud's mpop	Python	12664	7	160	22124	Normal	Stable	0.369%
appengine-toolkit	JavaScript	18253	5	110	20572	Normal	Stable	0.364%
socket.io-java	Java	23533	4	187	46254	Young	Stable	0.335%
glinux	C	41052	8	55	3114	Very Old	Decreasing	0.335%
Pax URL	Java	31467	7	73	6923	Old	Decreasing	0.330%
honeycrm	Java	14864	2	45	3810	Normal	Decreasing	0.315%

TABLE II. THE REPRESENTATIVENESS OF ALL ICSE AND FSE PAPERS IN THE PAST 2 YEARS AS WELL AS THE FIVE MOST REPRESENTATIVE PAPERS. THE UNIVERSE IS THE ACTIVE OHLOH PROJECTS, THE SPACE IS (MAIN LANGUAGE, TOTAL LINES OF CODE, CONTRIBUTORS, CHURN, COMMITS, PROJECT AGE, PROJECT ACTIVITY) AND THE CONFIGURATION CONSISTS OF THE DEFAULT SIMILARITY FUNCTIONS.

	All papers of ICSE and FSE (past 2 years)	Gabel and Su. Uniqueness of source code	Appel et al. Semistructured merge	Beck and Diehl. Modularity and code coupling	Uddin et al. Analysis of API usage concepts	Jin and Orso. Reproducing field failures
Score	9.15%	1.09%	0.85%	0.81%	0.73%	0.67%
Main language	91.42%	48.57%	43.62%	15.07%	15.07%	12.37%
Total lines of code	99.29%	65.82%	57.06%	55.98%	32.30%	61.93%
Contributors (12 months)	100.00%	99.94%	99.79%	99.13%	99.77%	96.94%
Churn (12 months)	98.08%	70.45%	70.68%	79.32%	62.51%	51.58%
Commits (12 months)	100.00%	97.98%	96.93%	97.68%	56.12%	67.78%
Project age	100.00%	88.80%	43.75%	20.39%	20.39%	20.39%
Project activity	100.00%	100.00%	100.00%	100.00%	100.00%	59.83%

conference, looked for the software projects that were analyzed and recorded the number and—if mentioned—the names of the projects in a spreadsheet. We then queried Ohloh for each of the software projects to find the corresponding identifier, which we used to cross-reference the data with our corpus.

Some projects we could not cross reference with our dataset because of any one of the following reasons: (a) the project was not indexed by Ohloh; (b) the paper used an aggregated set of projects, and we cannot name any one particular project that the authors used; (c) the project does not meet the criteria to be included in the universe, e.g., the project has not been under development in the past year, has only one developers, or has missing or invalid data.

The analysis of the ICSE and FSE conferences revealed several large-scale studies that analyzed hundreds if not thousands of projects. Some of these papers we had to exclude from our analysis as they either analyzed closed-source projects or did not report the names of the individual projects analyzed or analyzed inactive Ohloh projects.

What are the most frequent projects used in ICSE and FSE?

We found 635 unique projects that were analyzed by the ICSE and FSE conferences in the two-year period. Out of these we could map 207 to the universe of active Ohloh projects.

The most frequently studied projects were the Eclipse Java Development Tools (JDT) in 16 papers, Apache HTTP Server in 12 papers, gzip, jEdit, Apache Xalan C++, and Apache Lucene each in 8 papers and Mozilla Firefox in 7 papers. Another frequently studied project is Linux, which was analyzed in 12 papers. While the Linux project is listed on Ohloh, the code analysis has not yet completed and only limited information is available (no activity, no lines of code). Therefore we ignored Linux from our analysis.

How much of the Ohloh universe do ICSE and FSE cover?

The 207 Ohloh projects analyzed in the two years of the ICSE and FSE conferences were representative of 9.15% of the Ohloh population. At a first glance this score seems low, but one has to keep in mind that it is based on strict notion of representativeness: values in *all* dimensions have to be similar for a project to be representative of another. Low scores are not bad as we will discuss in Section IV.A.

Our algorithm also measures the representativeness for each dimension. Here the numbers are very promising (see second column in Table II): for all but one dimension the representativeness scores exceed 98%, which indicates that research published at ICSE and FSE covers a wide spectrum of software in terms of team size, activity, and project size. The lowest score

is for programming language, but still at an impressive 91.42%. The unstudied languages highlight opportunities for future research: Objective-C is used by 245, Vim script by 145, Scala by 119, Erlang by 108, and Haskell by 99 projects.

What are showcases of representative research?

We identified several outstanding papers in terms of high representativeness. In Table II, the Columns 3 to 8 show the total score and the dimension scores for the five most representative papers:

- "A study of the uniqueness of source code" by Gabel and Su [3] analyzed over 6,000 projects of which 30 were named in the paper and analyzed in depth. The score is computed for only the 30 named projects. The bulk of the corpus is from the source distribution of the Fedora Linux distribution (rel. 12). The authors studied multiple programming languages (C, C++, Java).
- "Semistructured merge: rethinking merge in revision control systems" by Apel et al. [10] evaluated a merge algorithm on 24 projects written in the C#, Python, and Java languages.
- "On the congruence of modularity and code coupling" by Beck and Diehl [11] analyzed 16 small to medium sized projects written in Java.
- "Temporal analysis of API usage concepts" by Uddin et al. [12] studied 19 client software projects. They covered a wide spectrum of project size (5.9 to 2991.8 KLOC) but given the nature of their study focused on older projects with larger amounts of history.
- "BugRedux: Reproducing field failures for in-house debugging" by Jin and Orso [13] recreated 17 failures of 15 real world programs. The size of the projects was between 0.5 and 241 KLOC.

Again the total scores seem to be low, which we will discuss in Section IV.A. More importantly however, the numbers in Table II allow assessing which dimensions papers covered well and which dimensions need improvement. For example, Beck and Diehl [11], Uddin et al. [12], and Jin and Orso [13] focused on a single programming language (Java and C respectively). To further increase the generality, additional languages may be studied. Another example is project age: all three papers focused on older projects, possibly because they needed long project histories that are only available for older projects.

Note that this is not a criticism of this research; these are merely ideas on how to become more representative of the Ohloh universe. Also note that the relevant target universe may be different for each paper. For example research on Java projects may limit itself to the Java universe.

It is noteworthy that several of these papers selected their subjects with respect to a dimension that is not included in our space: the functionality of the software. Given the availability of data, the dimension could be easily added to our space and accounted for in our score computation.

III.E. Data Availability

All data that has been used for the experiments in this section is available at the following URL. This includes the Ohloh data for universe and space as well as the spreadsheets with the conference data.

http://sailhome.cs.queensu.ca/replication/representativeness/

IV. Discussion

Having introduced our technique for assessing project sample representativeness and demonstrated it on recent software engineering research, we now discuss issues surrounding the use of such a technique in research. The use is not as straightforward as one might think. Here are some considerations.

IV.A. Low Representativeness is Not Bad

One observation that we have made in the course of using our techniques to measure representativeness is that many studies have low levels of software representativeness. At first glance, one might be tempted to conclude that these studies do not contribute much to the body of knowledge in software engineering or that others with higher representativeness are better. Identifying how representative a study is does not devalue the research, but rather gives further insight into the results.

For example, Zhou et al.'s recent result that bug report attributes can be used to automatically identify the likely location of a fix was evaluated on Eclipse JDT, SWT, AspectJ, and ZXing [14]. The representativeness score for this paper across the Ohloh universe is 0.0028, because these are all Java and C++ codebases aimed at developers (SWT and ZXing are libraries, AspectJ and Eclipse are tools for Java development). The low representativeness does not mean that the results are invalid or not useful. Rather, it yields additional insight into the technique. Apparently, bugs reported against libraries and Java tools contain relevant information to help identify fix locations. Thus, others building on this work might also evaluate on Java tools and libraries. Other avenues of research include investigating whether the approach also works well for codebases where bug reporters are not as likely to be developers.

We stress that the representativeness scores do not increase or decrease the importance of research, but rather enhance our ability to reason about it.

IV.B. Generality is Rare

The discussion from the previous subsection leads to a related point. Few empirical findings in software engineering are completely general [1]. A finding that is true in the context of large scale enterprise server Java development on a ten year old codebase may not hold for a relatively new Android widget. There may be fear when reporting results and trying to achieve representativeness that unless some hypothesis is confirmed in all cases, it is does not contribute to the body of knowledge in software engineering and is not fit for publication. This isn't so.

Kitchenham's work on within- and cross-company effort estimation [15] showed that it *is* indeed possible to estimate effort of one project based on history of others, but that there is no general rule for effort estimation. Rather they used regression analysis to find that similarities in the size of the development team, number of web pages, and high effort functions between projects in different companies are related to similar

effort requirements (i.e., different projects have different effort requirements, but projects that are representative of each other have similar effort needs).

Knowledge can be synthesized when reporting representativeness along different dimensions even when empirical results differ. Systematic reviews rely upon this principle. The recent review of fault prediction performance by Hall et al. [16] essentially constructed a space consisting of modeling techniques, metrics used, and granularity and found that fault prediction approaches performed differently. However, they were also able to conclude that simpler modeling techniques such as Naïve Bayes and Logistic regression tended to perform the best. In the same way, selecting projects that cover a large area in the project universe and examining where results are valid and where they are not give deeper insight into the research results. As Murphy-Hill et al. explain, "Simply explaining the context in which a study occurs goes a long way towards creating impactful research" because this allows a practitioner to "decide whether your research applies to her." [17]

IV.C. Reporting should be Consistent

We have provided a technique for selecting a representative sample of software projects and also for providing representativeness scores for samples. While selecting projects in a more rigorous and objective way is important, reporting in a consistent and consumable manner is just as important.

Most papers include a summary of characteristics of the projects included (e.g., size, age, number of checkins, number of contributors, language). This is an appropriate place to report the representativeness of the selected sample of projects. As illustrated in Section III, the universe and the space that is used should also be *explicitly* described and the rationale provided. How was the universe chosen? Why was each dimension in the space selected? For example, one might select only Java projects as a universe if a technique only makes sense in the context of Java.

If projects from different parts of the space show different results, they should be reported and discussed. Differences by dimension or location in the space provide a unique opportunity to refine theories and investigate further.

Finally, issues in sampling can affect external validity. Any potential problems or gaps in representativeness should be discussed in a section discussing validity, usually entitled "Threats to Validity" or "Limitations".

IV.D. Next Steps

What do we hope will come from this work? Our goal has not been to claim or imply that prior work is flawed, but rather to show that we can improve our practice and provide methods to do so. It is our hope that researchers will begin to select projects in an objective, representative-based way.

We realize that different studies and techniques are aimed at different problems and thus the goal may not always be to achieve maximum representativeness of all software projects. Further, the dimensions that people care about may differ. For instance, when evaluating techniques for mining API rules, the age of each project evaluated on may not be of concern. Our framework is general enough that researchers can define their own universe (the population they want to be representative of) and space (the dimensions they care about). But it does little good if each study reports its representativeness using different and opportunistic spaces and universes. We hope that this work sparks a dialog in our community about representative software engineering research and that some level of consensus on what universes and spaces are appropriate will be achieved. It is likely that different sub-disciplines will arrive at different answers to these questions, which we feel is reasonable.

V. Related Work

Some of the earliest research studies on representativeness were by Kahneman and Tversky [18] [19]. In their study, they stated that the sample size is not related to any property of the population and "will have little to no effect on judgment of likelihood". In their experiments they determined that people's perception of the likelihood of an event depended more on its representativeness to the population than the size of it. Thus they concluded that there is a difference between people's judgment and the normative probabilities. They call this the *representative heuristic*. In a more recent study, Nilsson et al. [20] investigated the cognitive substrate of the representativeness heuristic. In our study we borrow the concept of representativeness from them. However, unlike their studies, we are not evaluating the likelihood of an event or how people's perception differs from the actual probability of an event. We rather propose the means to measure the representatives of the sample (software systems used in the case study) to the population (the relevant universe of software).

Selecting representative samples for case studies has been a problem in fields such as clinical trials, social sciences, and marketing for decades. Hence studies such as the one by Robinson et al. [21] evaluated selection biases and their effects on the ability to make inferences based on results in clinical trials. They found that biases did exist, certain subgroups were underrepresented (e.g., women) while others were overrepresented (e.g., blacks). Their statistical models found that the selection biases may not influence general outcomes of the trials, but would affect generalizability of results for select subgroups.

Another area of research that often encounters the issue of representativeness is the field of systematic literature reviews. If the set of studies selected to be a part of the literature review is not representative of the research field under study, then the conclusions of the reviews can potentially be biased. Hence a variety of guidelines that are written for conducting systematic literature surveys place a large emphasis on the selection of the studies that will be included in the review [22] [23] [24] [25]. All the guidelines suggest that the researchers conducting the review must make the selection and rejection criteria clear for the reader to place the conclusions in context. In literature review studies researcher are not looking for a representative but rather a complete sample. The goal in literature reviews is to obtain every possible sample before including or rejecting them from the study. Hence steps such as searching the gray area of publications and asking experts in the field are suggested to obtain a more inclusive initial sample.

One line of research that attempts to rigorously achieve representativeness is the work on the COCOMO cost estimation model by Boehm et al. [26]. In this model, they collect software development project data and model it in order to help estimate and plan for the cost, effort and schedule of a project. The "Center for Systems and Software Engineering" at the University of Southern California to this day collects data to have a more representative dataset of projects, and to calibrate the model in order to provide better estimates [27]. Kemerer, in his validation of software cost estimation models, found that using an untuned cost estimation model can produce inaccurate estimates (up to 600% in some cases) [28]. In a more recent study, Chen et al. [29] examined how to prepare the available data in order to obtain better estimates. Unlike Chen et al.'s work, we do not provide techniques to pre-process an individual dataset. Our research goals are more similar to the research goals of the COCOMO model. The COCOMO model builds a statistical model with the available datasets. Then it tries to fit the current project that needs estimation, in this model to determine the particular space in the universe that this project belongs to. We use similar concepts, but attempt to determine how representative the current set of projects is in terms of the universe.

There have been several studies in software engineering on guidelines for conducting and reporting empirical software engineering research. [30] [31] [32] [33] [34]. Most of these studies focus on the process to be followed in an empirical study. One of the common themes is that all of the studies are the set of guidelines for reporting the experimental setting. This description will help the reader in understanding the context of the study, and allows future researchers to replicate the study. With respect to the sample of software systems used in the experiments, these studies do not discuss how to select the sample, but rather discuss what to report about the selection.

Unlike these studies, in our work we present a framework for the research community to measure the representativeness of the sample. This will help in quantifying the representativeness of the sample within the population, thereby helping the reader better understand the context under which the results of the study are applicable.

VI. Conclusion

With the availability of open source projects, the software engineering research community is examining an increasing number of software projects to test individual hypothesis or evaluate individual tools. However, more is not necessarily better and *the selection of projects counts* as well. With this paper we provide the researcher community with a technique to assess how well a research study covers a population of software projects. This helps researchers to make informed decisions about which projects to select for a study. Our technique has three parameters (universe, space, and configuration), which all can be customized based on the research topic and should be included with any sample that is scored.

We hope that this work sparks a dialog about representative research in software engineering and that some level of consensus on appropriate universes and spaces will be reached, which likely will differ across different sub-disciplines. We also hope that more datasets will become available that allow to explore alternative universes and spaces.

Our technique also extends to researchers analyzing closed source projects. They can now describe the representativeness of their projects without revealing confidential information about the projects or their metrics and place their results in context. Companies can use our technique to place academic research into the context of their own development by comparing against a company-specific universe and space.

Acknowledgments

We would like to thank our colleagues at the SAIL lab at Queen's University and at the ESE group at Microsoft Research for valuable feedback on this idea. We would also like to thank all the researchers whose work we looked at! Lastly, we would like to thank Black Duck Software and Ohloh (www.ohloh.net) for collecting and making the data available.

References

1. Basili, V.R., Shull, F., and Lanubile, F. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25 (1999), 456--473.
2. Robbes, R., Tanter, E., and Rothlisberger, D. How developers use the dynamic features of programming languages: the case of smalltalk. *Proceedings of the International Working Conference on Mining Software Repositories* (2011).
3. Gabel, M. and Su, Z. A study of the uniqueness of source code. In *FSE'10: Proceedings of the International Symposium on Foundations of Software Engineering* (2010), 147-156.
4. NIH. *NIH Guideline on The Inclusion of Women and Minorities*. , 2001. http://grants.nih.gov/grants/funding/women_min/guidelines_amended_10_2001.htm.
5. Mulrow, C.D., Thacker, S.B., and Pugh, J.A. A proposal for more informative abstracts of review articles. *Annals of internal medicine*, 108 (1988), 613--615.
6. Kitchenham, B.A., Mendes, E., and Travassos, G.H. Cross versus Within-Company Cost Estimation Studies: A Systematic Review. *IEEE Trans. Software Eng. (TSE)*, 33, 5 (2007), 316-329.
7. Hill, P.R. *Practical Software Project Estimation*. McGraw-Hill Osborne Media, 2010.
8. BLACK DUCK SOFTWARE. *Ohloh, http://www.ohloh.net/*.
9. Sands, R. *Measuring Project Activity*. http://meta.ohloh.net/2012/04/measuring-project-activity/. 2012.
10. Apel, S., Liebig, J., Brandl, B., Lengauer, C., and Kästner, C. Semistructured merge: rethinking merge in revision control systems. In *ESEC/FSE'11: European Software Engineering Conference and Symposium on Foundations of Software Engineering* (2011), 190-200.
11. Beck, F. and Diehl, S. On the congruence of modularity and code coupling. In *ESEC/FSE'11: European Software Engineering Conference and Symposium on Foundations of Software Engineering* (2011), 354-364.
12. Uddin, G., Dagenais, B., and Robillard, M.P. Temporal analysis of API usage concepts. In *ICSE'12: Proceedings of 34th International Conference on Software Engineering* (2012), 804-814.

13. Jin, W. and Orso, A. BugRedux: Reproducing field failures for in-house debugging. In *ICSE'12: Proceedings of 34th International Conference on Software Engineering* (2012), 474-484.
14. Zhou, J., Zhang, H., and Lo, D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *International Conference on Software Engineering* (2012).
15. Kitchenham, B.A. and Mendes, E. A comparison of cross-company and within-company effort estimation models for web applications. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering* (2004), 47-55.
16. Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. A systematic review of fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 99 (2011).
17. Murphy-Hill, E., Murphy, G.C., and Griswold, W.G. Understanding Context: Creating a Lasting Impact in Experimental Software Engineering Research. In *Proceedings of the Workshop on Future of Software Engineering* (2010), 255-258.
18. Kahneman, D. and Tversky, A. Subjective probability: A judgment of representativeness. *Cognitive Psychology*, 3 (1972), 430 - 454.
19. Tversky, A. and Kahneman, D. Judgment under Uncertainty: Heuristics and Biases. *Science*, 185 (1974), pp. 1124-1131.
20. Nilsson, H., Juslin, P., and Olsson, H. Exemplars in the mist: The cognitive substrate of the representativeness heuristic. *Scandinavian Journal of Psychology*, 49, 201--212.
21. Robinson, D., Woerner, M.G., Pollack, S., and Lerner, G. Subject Selection Biases in Clinical Trials: Data From a Multicenter Schizophrenia Treatment Study. *Journal of Clinical Psychopharmacology*, 16, 2 (April 1996), 170-176.
22. Khan, K.S. et al., eds. NHS Centre for Reviews and Dissemination, University of York, 2001.
23. Kitchenham, B. Procedures for undertaking systematic reviews. *Technical Report TR/SE-0401, Department of Computer Science, Keele University and National ICT, Australia Ltd* (2004).
24. Brereton, P., Kitchenham, B.A., Budgen, D., Turner, M., and Khalil, M. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80 (2007), 571 - 583.
25. *Standards for Systematic Reviews*..
26. Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D.J., and Steece, B.t.=.S.C.E.w.C.I. NHS Centre for Reviews and Dissemination, University of York, 2000.
27. *Center for Systems and Software Engineering*..
28. Kemerer, C.F. An empirical validation of software cost estimation models. *Commun. ACM*, 30 (may 1987), 416--429.
29. Chen, Z., Menzies, T., Port, D., and Boehm, D. Finding the right data for software cost modeling. *Software, IEEE*, 22 (nov.-dec. 2005), 38 - 46.
30. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., and Wesslen, A. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.
31. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., Emam, E.K., and Rosenberg, J. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28 (aug 2002), 721--734.
32. Jedlitschka, A. and Pfahl, D. Reporting guidelines for controlled experiments in software engineering. In *Empirical Software Engineering, 2005. 2005 International Symposium on* (nov. 2005), 10 pp.
33. Kitchenham, B., Al-Khilidar, H., Babar, M.A., Berry, M., Cox, K., Keung, J., Kurniawati, F., Staples, M., Zhang, H., and Zhu, L. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Softw. Engg.*, 13 (feb 2008), 97--121.
34. Runeson, P. and Host, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14 (Apr 2009), 131--164.
35. Sprague, S. and Bhandari, M. Organizationa and Planning. In Gad, S.C., ed., *Clinical Trials Handbook*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2009.

VII. APPENDIX

VII.A. How to compute the score?

This example below uses the Ohloh universe to score the Mozilla Firefox project along the space (Lines of Code, Developers). The text `id ~ total_code_lines + twelve_month_contributor_count` is R syntax and commonly used to define models.

```
url <- "http://sailhome.cs.queensu.ca/replication/representativeness/ masterdata.txt"
ohloh <- read.delim(url, header=T, na.strings=c("", "NA"))
sample <- ohloh[ohloh$name=="Mozilla Firefox",]
score <- score.projects(sample, universe=ohloh, id ~ total_code_lines + twelve_month_contributor_count)
```

The resulting total score is in `score$score` and the dimension scores are in `score$dimension.score`.

VII.B. How to select the next projects?

This example adds 10 more projects to the sample from the previous example. The result is a data frame `np$new.projects` with the projects to be added to the sample and the score object of the combined sample `np$score`.

```
np <- next.projects(10, sample, universe=ohloh, id ~ total_code_lines + twelve_month_contributor_count)
```

VII.C. How to change the configuration?

Provide a list with the similarity functions. Values NA indicates that the default similarity function should be used for a dimension. In the example below the function `custom.similarity` will be used the first dimension.

```
score <- score.projects(sample, universe=ohloh, ..., configuration=c(custom.similarity, NA))
```

