The Babel of Software Development: Linguistic Diversity in Open Source

Bogdan Vasilescu*, Alexander Serebrenik, and Mark G. J. van den Brand

Eindhoven University of Technology, The Netherlands
{b.n.vasilescu, a.serebrenik, m.g.j.v.d.brand}@tue.nl

Abstract. Open source software (OSS) development communities are typically very specialised, on the one hand, and experience high turnover, on the other. Combination of specialization and turnover can cause parts of the system implemented in a certain programming language to become unmaintainable, if knowledge of that language has disappeared together with the retiring developers.

Inspired by measures of linguistic diversity from the study of natural languages, we propose a method to quantify the risk of not having maintainers for code implemented in a certain programming language. To illustrate our approach, we studied risks associated with different languages in Emacs, and found examples of low risk due to high popularity (e.g., C, Emacs Lisp); low risk due to similarity with popular languages (e.g., C++, Java, Python); or high risk due to both low popularity and low similarity with popular languages (e.g., Lex). Our results show that methods from the social sciences can be successfully applied in the study of information systems, and open numerous avenues for future research.

1 Introduction

Open source software (OSS) development is typically characterised as a decentralised, self-directed, highly interactive, and knowledge-intensive process [14]. In OSS, programmers with different skill sets and skill levels, supporters, and users organise themselves in virtual (online) communities, and voluntarily contribute to a collaborative software project [22].

OSS communities are typically very specialised [26, 29, 37]: contributors focus on few activity types and are very territorial, touching only few parts of the system. OSS communities also co-evolve together with the associated OSS systems [22]: faced with turnover [28], these communities are sustained and reproduced over time through the progressive integration of new members [6]. However, with the abandonment of existing developers, OSS communities lose human resources with knowledge of the system or of some of its components, or, stated differently, with mastery of certain programming languages. Ensuring the heterogeneity of an OSS community in terms of the skills of its members is important for a project's survival and performance [9]. To further put this issue

*The author is supported by the Dutch Science Foundation through the project "Multi-Language Systems: Analysis and Visualization of Evolution—Analysis" (612.001.020).

into context, software systems are increasingly developed using multiple programming languages, as illustrated by the growing proportion of multi-language software developed in the United States from 1998 (30%) [16] to 2008 (50%) [17]. In addition, as languages become obsolete and development teams are faced with the problem of maintaining legacy code, or migrating it in order to survive, finding developers with knowledge of obsolescent technologies becomes more challenging. As the case may be, OSS communities are exposed to the *risk of not finding suitable contributors with knowledge of certain programming languages.*

Although new to software maintenance research, quantifying the risks associated with knowledge of programming languages in OSS communities around multi-language systems is related to the well-known concept of linguistic diversity from the study of natural languages [11]. Drawing inspiration from measures of linguistic diversity (Section 2), in this paper we attempt to quantify the aforementioned risk, associated with a given programming language in an OSS community (Section 3). Our model assumes that contributors are polyglot, i.e., they can "speak" more than one programming language. Moreover, analogously to dialects of a natural language being regarded as similar (mutually intelligible), our model also considers certain programming languages to be related. To quantify the strength of this relation, we mine patterns of shared knowledge of programming languages from developers participating in StackOverflow, a popular Q&A website (Section 4). Such relations need not be symmetrical: just like "Swedish is more easily understandable for a Dane, than Danish for a Swede" [20], our StackOverflow-based measure considers, e.g., that a C++ developer would be able to take over code written in C with less difficulty than the other way around.

By design, we can distinguish between two types of programming languages: those causing *high risk* within an OSS community (due to limited spread and low "similarity" with other more popular languages), and those causing *low risk* (either due to their popularity, or to their closeness to other more popular languages known to members of the community). Finally, to illustrate our risk measure, we track its evolution throughout the evolution of Emacs (Section 5).

2 Linguistic diversity for natural languages

Measuring linguistic diversity for natural languages is an old research topic, dating back to Greenberg in 1956 [11]. For a given geographical area, Greenberg considers the probability that two randomly-chosen individuals *do not* speak the same language as a measure of the region's linguistic diversity. In this model (the first in a series of eight such measures proposed by Greenberg), if everyone speaks the same language, the probability that two randomly-chosen individuals speak the same language is, naturally, 1. Similarly, if everyone speaks a different language, this probability is 0. In general, for a language ℓ, the probability p_ℓ that a randomly-chosen individual speaks ℓ is the proportion of ℓ-speakers to the total population, i.e., $p_\ell = \frac{|S_\ell|}{|P|}$, where S_ℓ is the set of ℓ-speakers, P is the entire population, and $|\cdot|$ denotes cardinality. Consequently, the probability that two randomly-chosen individuals speak ℓ is p_ℓ^2, hence the probability that two randomly-chosen individuals speak the same language is $\sum_{\ell \in L} p_\ell^2$, where L is the set of all languages spoken in that region. The Greenberg linguistic diversity index A [11], corresponding to this simple model, is defined as[1]

$$A = 1 - \sum_{\ell \in L} p_\ell^2 \tag{1}$$

A reaches its minimum of 0 when everyone speaks the same language (linguistic uniformity). Similarly, A reaches its maximum of 1 when everyone speaks a different language (maximal linguistic diversity). However, this model is overly simplistic, since (i) it does not consider *mutual intelligibility between different languages* (linguistic diversity should be lower in areas where related languages or dialects are spoken), and (ii) it does not consider *polyglotism* (a member of the population can speak more than one language). Concerns (i) and (ii) above are orthogonal. To account for (i), Greenberg proposed B [11], defined as

$$B = 1 - \sum_{\ell, m \in L} p_\ell p_m \cdot \mathit{sim}(\ell, m), \tag{2}$$

where $\mathit{sim}(\ell, m)$ is a measure of mutual intelligibility interpreted as the similarity between languages ℓ and m (ranging between 0 when ℓ and m are completely independent, and 1 when $\ell = m$). Clearly, B reduces to A if $\mathit{sim}(\ell, m) = 1$ when $\ell = m$, and 0 otherwise.

To account for (ii), if polyglot, an individual is considered equally probable to speak any of the languages she commands, hence the expressions for A and B above are adjusted accordingly. Let $\mathcal{L}$ be the power set (set of all subsets) of L, excluding the empty set; $|\mathcal{L}| = 2^n - 1$, where $n = |L|$. For example, if $L = \{A, B, C\}$, then $\mathcal{L} = \{A, B, C, AB, AC, BC, ABC\}$. Let X_ℓ be the set of *exclusive* ℓ speakers, i.e., individuals that speak ℓ but do not speak any other language besides ℓ. For a subset $s \in \mathcal{L}$, by abuse of notation, we write X_s to denote the set of individuals that speak the combination of languages in s exclusively (i.e., they speak all languages in s, but no other languages besides those). By definition, $\sum_{s \in \mathcal{L}} |X_s| = |P|$. Index B becomes [11]

$$F = 1 - \sum_{s, t \in \mathcal{L}} p_s p_t \cdot \frac{\sum_{\ell \in s, m \in t} \mathit{sim}(\ell, m)}{|s| \cdot |t|}, \tag{3}$$

where $p_s = \frac{|X_s|}{|P|}$, for all $s \in \mathcal{L}$. Clearly, F reduces to B in the monolingual case, since $|s| = 1$ for all $s \in \mathcal{L}$. B further reduces to A as discussed above.

Indices B and F as defined above interpret mutual intelligibility as similarity and hence assume it to be symmetric. However, it is well-known that mutual intelligibility is not necessarily symmetric: e.g., Swedes have more difculties understanding Danish as opposed to Danes attempting to understand Swedish [20].

[1]In the original paper [11] Greenberg only describes the linguistic diversity indices, but does not formalise them. The current formalisation is ours.

Therefore, one can define $sim(\ell, m) = \max(mi_\ell(m), mi_m(\ell))$ where $mi_\ell(m)$ is the measure of intelligibility of the language m for speakers of language ℓ. Similarly to $sim(\ell, m)$ we require $mi_\ell(m)$ to range between 0 and 1, such that $mi_\ell(m) = 0$ if m is unintelligible for the speakers of ℓ and $mi_\ell(m) = 1$ if m is intelligible for all speakers of ℓ. In particular, if $\ell = m$ then $mi_\ell(m) = 1$.

3 Risk of using a programming language

There are many risks impacting software development, and many methods to estimate them [23]. In this paper we do not aim to cover all possible facets of risk, but rather focus on a particular scenario. For a (open source) software project using multiple programming languages, we study the readiness of the developer community to take over code implemented in a certain language, and evaluate the risk of not finding contributors that can "speak" that language.

Based on the discussion of linguistic diversity above, we require that a measure of this risk be *domain-specific*, i.e., aware of relations between programming languages. To simplify our model, we assume *perfect fluency* of developers in all the features of the languages they speak, even as the languages evolve. In addition, we assume *constant knowledge in time* (i.e., once a developer speaks a certain programming language, she never "forgets" how to speak it). For the purpose of empirically illustrating the risk measure (Section 5), we need to approximate at each point in time the developers with knowledge of a certain programming language. To this end, we furthermore assume *instant fluency*: a developer is said to "speak" a certain programming language at time τ if she has performed at least one change to a source code file in that language, prior to τ. Relaxing these assumptions is considered as future work.

3.1 Risk measure

Let S be a multi-lingual software system, and let D be its developer community at time τ. Let L be the set of programming languages in use in S at time τ, i.e., those for which there exist source code files at time τ that need to be maintained. Similarly to the formalisation of the Greenberg indices from Section 2, let $\mathcal{L}$ be the power set of L, excluding the empty set. Let X_ℓ be the set of developers at time τ that speak ℓ *exclusively*, i.e., they speak ℓ but do not speak any other language besides ℓ. For a subset $s \in \mathcal{L}$, let X_s be the set of developers at time τ that speak the combination of languages in s exclusively (i.e., they speak all languages in s, but no other languages besides those). By definition,

$$\sum_{s \in \mathcal{L}} |X_s| = |D|. \tag{4}$$

For a programming language $\ell \in L$, we define the risk of S at time τ of not finding developers that *can* speak ℓ as

$$risk_\ell = 1 - \sum_{s \in \mathcal{L}} p_s \cdot \max_{k \in s} mi_\ell(k), \tag{5}$$

where $p_s = \frac{|X_s|}{|D|}$ is the probability at time τ that a developer speaks the combination of languages in s exclusively, and $mi_\ell(k)$ is an asymmetric mutual intelligibility measure as above. To illustrate the need for an asymmetric measure recall, for example, that C was originally a subset of C++ (the version of C defined by C89 is commonly referred to as the "C subset of C++" [30]), hence we perceive C to be more similar to C++ than C++ is to C. Therefore, assuming fluency of developers in all language features, and comparable complexity of the different components, we expect a C++ developer to be able to take over C code with less difficulty than the other way around.

As opposed to Greenberg [11] who is interested in an "average" case (i.e., as discussed in Section 2, if polyglot, an individual is considered equally probable to speak any of the languages she commands) when computing the linguistic diversity index F (3), we opt for the $\max(\cdot)$ function in (5) to denote that if polyglot, it is the language most intelligible to the language in question that will influence how difficult it is for a developer to take over that code.

To obtain a better understanding of how (5) can provide insights in the risk of not finding developers that can speak ℓ, we distinguish between developers D_ℓ that speak ℓ, and developers $D_{\neg\ell} = D \setminus D_\ell$ that do not speak ℓ. Similarly, let $\mathcal{L}_\ell$ be a subset of $\mathcal{L}$ such that $\forall s \in \mathcal{L}_\ell, \ell \in s$, and let $\mathcal{L}_{\neg\ell} = \mathcal{L} \setminus \mathcal{L}_\ell$. Then, we can rewrite (5) as $risk_\ell = 1 - \sum_{s\in\mathcal{L}_\ell} p_s \cdot \max_{k\in s} mi_\ell(k) - \sum_{s\in\mathcal{L}_{\neg\ell}} p_s \cdot \max_{k\in s} mi_\ell(k)$ which, given that $mi_\ell(k) = 1$ if $k = \ell$, and $\max_{k\in s} mi_\ell(k) = 1$ for all $s \in \mathcal{L}_\ell$, further simplifies to:

$$risk_\ell = \frac{|D_{\neg\ell}|}{|D|} - \sum_{s\in\mathcal{L}_{\neg\ell}} p_s \cdot \max_{k\in s} mi_\ell(k) \tag{6}$$

Closer inspection of (6) reveals that the risk of not finding developers that can speak ℓ is *high* if very few developers speak ℓ (i.e., $\frac{|D_{\neg\ell}|}{|D|} \simeq 1$) and other languages are very distinct from ℓ rendering ℓ barely intelligible for "speakers" of those languages (i.e., $\sum_{s\in\mathcal{L}_{\neg\ell}} p_s \cdot \max_{k\in s} mi_\ell(k) \simeq 0$ because the languages in the collection are very different from ℓ, $\max_{k\in s} mi_\ell(k) \simeq 0$). By a complementary argument, two typical *low*-risk scenarios are when almost everybody can speak ℓ (i.e., $\frac{|D_{\neg\ell}|}{|D|} \simeq 0$, hence $p_s \simeq 0$ for $s \in \mathcal{L}_{\neg\ell}$), or when almost nobody can speak ℓ (i.e., $\frac{|D_{\neg\ell}|}{|D|} \simeq 1$) but popular languages make ℓ easily understandable (i.e., $\max_{k\in s} mi_\ell(k) \simeq 1$ for $s \in \mathcal{L}_{\neg\ell}$). To distinguish between these two scenarios in the empirical evaluation (Section 5), we also consider the percentage of developers that do not speak ℓ, $\frac{|D_{\neg\ell}|}{|D|}$.

4 Similarity and mutual intelligibility between programming languages

Mutual intelligibility, while being distinct from traditional notion of similarity, is still close to it. Therefore, in this section we mostly focus on measures of similarity of natural languages [7, 11] and their counterparts in programming linguistics [8] (the study of programming languages), and introduce our mutual intelligibility measure based on analysing StackOverflow[2].

4.1 Approaches to similarity between programming languages

In linguistics, two complementary approaches to compute similarity between languages are commonly pursued. First, a similarity measure can be obtained "using an arbitrary but fixed basic vocabulary, e.g., the most recent version of the glottochronology list", by computing "the proportion of resemblances between each pair of languages to the total list" [11] (a similar approach has been recently pursued to study asymmetric mutual intelligibility [20]). Second, a similarity measure can be obtained using the distance between the branches languages fall into in a classification tree [7]. Using this approach, the more features two languages have in common, the more similar they are.

In programming linguistics, the approaches above are to a large extent unfeasible. First, application of approaches based on a common vocabulary would require establishing an agreed list of universal concepts present in all programming languages, akin to the Swadesh list for the natural languages [33]. The "word list" approach is being criticized in linguistics [13]; moreover, it introduces the need for identifying so-called cognates, or etymologically related words. The process of identifying cognates is complicated, since cognates do not necessarily look similar and words that look similar are not necessarily cognates [12]. Choosing the word list approach for similarity of programming languages assumes presence of universal concepts common to all (or at least most) programming languages. Even if compilation of such a list is possible at any given moment, it would rapidly become obsolete, since programming languages emerge much faster than natural languages. Moreover, the word list approach can be expected to trigger similar discussions about possible cognates, e.g., whether notions of a function in Lisp and C should be considered cognates or not.

One could also base a similarity measure on the shared concepts that underlie the design of both languages (e.g., data and types, variables and storage) and the paradigms to which they adhere (e.g., imperative or object-oriented) (cf. [7]). "We can master a new programming language most effectively if we understand the underlying concepts that it shares with other programming languages" [38, p4]. Again, the more attributes two languages would share in common, the more similar they would be considered. However, selecting the right attributes is challenging, to say the least. Most reliably, one could make use of taxonomies of programming languages [27]. However, as languages evolve, such taxonomies become inherently out of date and their categories change [15].

As an alternative to word-list and classification-tree approaches, one may consider recent studies [4, 18] that targeted the joint usage of programming languages. Karus and Gall [18] studied 22 open-source systems and observed two groups of languages for which the source code files frequently co-change, namely XML, XML Schema, WSDL (Web Service Definition Language) and Java on the

[2] http://stackoverflow.com

one hand, and JavaScript and XSL (e.g., XSLT, XPath) on the other hand. In a larger-scale study of 9,997 projects, Delorey et al. [4] observed that JavaScript and PHP, Java and JavaScript, C and C++, and C and Perl are commonly used both by the same authors as well as in the same projects.

As opposed to the actual usage, reflected in implementation of software systems, Doyle and Stretch [5] studied services offered by British software companies. The authors considered two programming languages to be similar if multiple companies offered these languages as part of their services. While Doyle and Stretch [5] employ the term "related by usage" to describe this relation, we prefer to call it "related by knowledge" and to reserve the term "related by usage" to such approaches as [4, 18]. Indeed, companies offering multiple programming languages as part of their services do not necessarily use these languages in the same project. Instead, these companies have employees, potentially different, knowledgeable about each of these languages.

Both the "related-by-usage" and "related-by-knowledge" approaches can be seen as pertaining to pragmatics of programming languages which, together with semantics, are considered the most decisive for quantifying the similarity between programming languages [38, p5]. Therefore, we also expect that as opposed to both the word-list and classification-tree approaches, *pragmatic similarity* more accurately reflects developer expertise and ease of switching from one programming language to another. In Section 4.3 we also propose a pragmatics-pertaining mutual intelligibility measure, refining the "related-by-knowledge" insights of Doyle and Stretch [5]. Specifically, we base the mutual intelligibility measure on shared knowledge of the programming languages, as reflected in StackOverflow tags representing programming languages.

4.2 StackOverflow

StackOverflow (SO) is a free programming questions and answers (Q&A) site known to foster knowledge sharing among the developers [34,36]. When posting a question, SO users associate at least one and at most five different tags to it, and, in turn, become associated with these tags. When answering a question, SO users inherit all the tags associated with this question. Therefore, while each question can have at most five tags, a user can inherit an arbitrarily large collection of tags from all the questions she asked and answered. Tags can be related to programming languages (e.g., `c#`, `java`, `php`), operating systems (e.g., `windows`, `linux`), specific frameworks or technologies (e.g., `hibernate`, `grails`), specific versions of either of the above (e.g., `c#-4.0`, `windows-7`, `hibernate-4.x`), cross-cutting concerns (e.g., `logging`, `algorithm`), or other topics. SO tags can be collaboratively edited: while anyone can suggest an edit to question tags, only higher ranked users can review and edit tags, ensuring quality and reliability of the tags. Here we explore the public SO data from September 2011 (2,010,348 questions and 756,694 users).[3] The data is organised such that one can distinguish between *question tags* and *user tags*; one can also distinguish between user tags collected from asking questions, and user tags inherited by answering questions.

[3] http://www.clearbits.net/torrents/1836-sept-2011

Question tags. Frequent pairs of tags (e.g., `javascript–jquery`, `asp.net–c#`) indicate that these languages are commonly used together. However, this approach has several drawbacks. First, the number and skills of the users answering these questions is not considered (potentially leading to false positives). For example, there may be many questions tagged τ_1 and τ_2, suggesting that these languages are related, but only few people answering them. The relation between τ_1 and τ_2 might therefore not be representative of the entire (large) developer community (e.g., although few gurus with knowledge of both τ_1 and τ_2 exist, average developers may not possess the skills to easily switch between them).

User tags - answering questions. Since users inherit tags from questions they answer, frequent pairs of tags indicate that developers who possess knowledge of one language commonly also possess knowledge of the other. A frequent pair of tags (τ_1, τ_2) can emerge from multiple situations:

– many users inheriting τ_1 and τ_2 by answering questions tagged (τ_1, τ_2): either there are few questions tagged (τ_1, τ_2), but many users answering them (i.e., although τ_1 and τ_2 do not seem to be commonly associated in practice – e.g., they used to be but are by now obsolete, there is still a large pool of developers mastering both), or there are many questions tagged (τ_1, τ_2), and many users answering them (i.e., τ_1 and τ_2 are both commonly associated in practice, and supported by a large user base);
– many users inheriting τ_1 from questions tagged τ_1, and τ_2 from different questions tagged τ_2 (hence the pair τ_1–τ_2). In addition to an argument similar to the previous one, this also indicates languages that although rarely related in practice, are commonly mastered by developers. Hence, although seemingly unrelated, it seems easy for developers to switch from one language to another since they frequently master both.

User tags - asking questions. Users also inherit tags from all the questions they ask. A frequently occurring pair (τ_1, τ_2) indicates that developers are frequently faced with joint usage of τ_1 and τ_2, irrespective of the expertise available. In turn, this can suggest an emerging trend in relating τ_1 and τ_2 by usage. Note that as opposed to the variation across questions (many questions [of the same person]), frequent pairs (τ_1, τ_2) are now supported by large user pools (many questions of many persons). However, this does not indicate developer expertise.

Questions vs. users. In conclusion, both the questions-based and the users-based approaches are subject to potential false positives resulting from competing rather than interacting languages. However, we opt for user tags rather than of question tags, since the former can suggest relations between languages representative of the skills of the developer community (i.e., there are many users that share knowledge of both—fewer false positives), as well as relations between independent languages (i.e., languages which are seemingly unrelated, but knowledge of both is frequently shared by users—fewer false negatives).

4.3 StackOverflow-based mutual intelligibility measure

To quantify shared knowledge of programming languages by developers participating in SO discussions, we perform association rule mining [1] on SO tags representing programming languages. We say that a language k (with tag τ_k) is mutually intelligible or "related by knowledge" to a language ℓ (with τ_ℓ) if many of the SO users having inherited τ_k are also associated with τ_ℓ. As mutual intelligibility measure of language k with respect to language ℓ we choose *confidence*, one of the measures typically used to quantify the strength of association rules.

$$mi_\ell(k) = conf(\tau_k \Rightarrow \tau_\ell) = \frac{nBoth}{nLeft}, \qquad (7)$$

where *nLeft* is the number of users associated with τ_k, and *nBoth* is the number of users associated with both τ_k and τ_ℓ.

To ensure quality of the association rules, we perform a number of pre- and postprocessing steps. Preprocessing consists of filtering out potentially unreliable posts (either questions or answers with negative or zero score, as reflected by the number of votes), and infrequent pairs of tags (encountered for a single user). This limits the number of eligible SO contributors to slightly over 400,000 (out of 756,694 initially). Postprocessing is based on *lift*, another popular quality measure for association rules. If $lift > 1$, the tags appear more frequently together in the data than expected under the assumption of conditional independence [2]. Moreover, we require it to be unlikely that $lift > 1$ is observed only by chance and perform Fisher's exact test to determine statistical significance. Hence, we say that k is unintelligible to the speakers of ℓ (we redefine when $mi_\ell(k) = 0$) if $lift \leq 1$, or $lift > 1$ is not statistically significant at 5% significance level. Approximately 7% of the pairs were removed by this filtering step (e.g., Python $\Rightarrow$ Visual FoxPro has lift 0.83; Curry $\Rightarrow$ C# has lift 1.78, $p = 0.31$). Finally, to reduce the amount of data processing required, we limit our scope to a subset of 160 programming languages, hence 160 corresponding SO tags. Our subset includes the most popular programming languages mentioned by TIOBE[4], Wikipedia[5], and the Transparent Language Popularity Index[6], as well as exotic languages such as M4 and RelaxNG, in use in Emacs. The complete list of languages included in our selection is part of the online appendix.[7]

4.4 Empirical results

Table 1 displays values of the mutual intelligibility measure for a subset of the programming languages considered (also studied in [4, 5, 18]). An entry (*row*, *column*) represents the similarity of the language in *column* with respect to the one in *row*. For complete results we refer to the online appendix[7]. By definition each language is perfectly mutually intelligible with itself (100% on the main

[4] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[5] http://en.wikipedia.org/wiki/List_of_programming_languages

[6] http://lang-index.sourceforge.net

[7] http://www.win.tue.nl/~bvasiles/languages/list.html

	Asm	C	C++	Cobol	CSS	Groovy	HTML	Java	JavaScript	Perl	PHP	Shell	XML
Asm	100%	55%	54%	1%	15%	1%	23%	39%	28%	12%	28%	1%	18%
C	8%	100%	48%	0%	12%	1%	17%	31%	21%	8%	21%	0%	13%
C++	5%	32%	100%	0%	10%	1%	15%	26%	18%	6%	18%	0%	11%
COBOL	12%	35%	40%	100%	24%	3%	29%	48%	38%	17%	37%	1%	28%
CSS	2%	10%	13%	0%	100%	1%	61%	21%	54%	5%	39%	0%	16%
Groovy	3%	15%	18%	1%	17%	100%	26%	63%	32%	7%	23%	0%	26%
HTML	2%	11%	14%	0%	46%	1%	100%	25%	56%	5%	40%	0%	18%
Java	2%	12%	15%	0%	10%	2%	15%	100%	19%	4%	16%	0%	12%
JavaScript	2%	9%	11%	0%	25%	1%	35%	20%	100%	4%	31%	0%	13%
Perl	5%	25%	27%	1%	18%	2%	26%	31%	30%	100%	31%	1%	19%
PHP	2%	9%	11%	0%	19%	1%	26%	17%	33%	4%	100%	0%	12%
Shell	12%	34%	38%	1%	19%	3%	32%	43%	33%	24%	35%	100%	24%
XML	3%	14%	19%	0%	20%	2%	29%	34%	35%	7%	31%	0%	100%

Table 1. SO-based mutual intelligibility measure ([column] with respect to [row]).

diagonal). Next we observe that Assembly programmers are usually well-versed in other languages, including HTML (23%), Java (39%) and JavaScript (28%). Since there are only 44 posts (questions+answers) tagged `assembly` and `java`, these languages are unlikely to be related by usage, but are related by knowledge (more than 1000 developers with knowledge of both). Hence, if replacement developers are required for Java, one might consider the Assembly developers as candidates. We further observe that all the languages considered exhibit low intelligibility with such languages as COBOL, Groovy and Shell. This means that when COBOL, Groovy or Shell programmers leave, finding their replacement among programmers versed in other languages in Table 1 might be problematic. For COBOL one could argue that the low values can be explained by under-representation of legacy technologies on SO. This is, however, highly unlikely for Groovy, an object-oriented programming language first released in 2007. Low mutual intelligibility values of other languages with COBOL, Groovy and Shell contrast sharply with more easily replaceable developers in such languages as C, C++, HTML or Java. As expected, the table also shows a high degree of asymmetry. For instance, 63% of Groovy programmers know Java but only 1% of Java programmers know Groovy (not surprising since Groovy has been developed for Java, but constitutes only a minor fraction of the overall Java development).

Although we are measuring different things (similarity by usage in case of Karus and Gall [18] and Delorey et al. [4] versus mutual intelligibility or similarity by knowledge in our case), we expect that similarity by usage implies similarity by knowledge, since languages used together by the same person are likely to be known together by that person. Our results partly support the findings of Karus and Gall [18] (strong relation between XML and Java—34%, and limited evidence for C/C++ and XML—13% and 11%, respectively). Our results also support the findings of Delorey et al. [4], who observed strong relations between JavaScript and PHP ($\Rightarrow$ 31%, $\Leftarrow$ 33%), Java and JavaScript ($\Rightarrow$ 19%, $\Leftarrow$ 20%), C and C++ ($\Rightarrow$ 48%, $\Leftarrow$ 32%), and C and Perl ($\Rightarrow$ 8%, $\Leftarrow$ 25%).

5 Illustration of the approach

To illustrate our approach in a real-world context, we performed a case study on Emacs [32], a popular text editor in development since the mid-1970s.

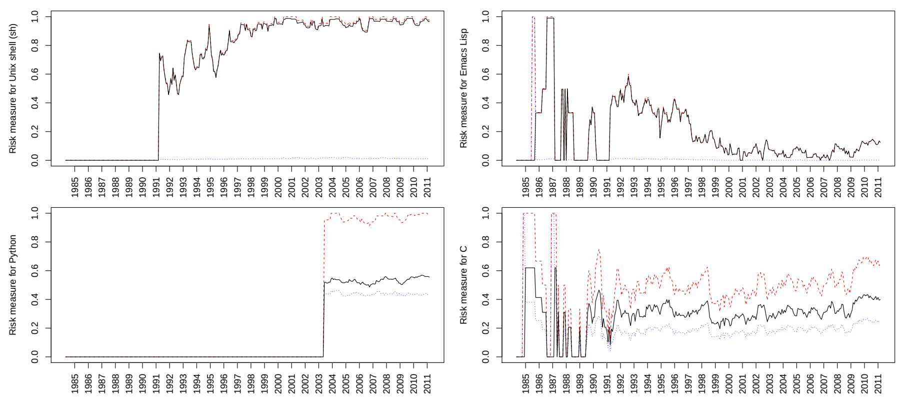

Fig. 1. The risk measure $risk_\ell$ (black solid line), the share of the community that does not speak ℓ (dashed red line), and the difference between the two (dotted blue line).

We identified 446 different (*name*, *email*) pairs in the *author* field for each change recorded in the Git log, corresponding to 27 years of Emacs development (1985-2012). Since there were multiple email addresses associated with the same names, and multiple names associated with the same email addresses, we performed identity merging [19, 37] (369 unique identities remained). To track the evolution of our risk measure throughout the evolution of Emacs, we extracted the programming languages used. We analysed the filename extensions of all the source files mentioned in the Git log. After filtering out files without extensions (mostly related to documentation), configuration files, make files, documentation files, and auxiliary files (e.g., used by the version control system), we uncovered the following 26 different programming languages: Assembly, Awk, Bash, Bison, C, C++, Cocoa, C shell, Emacs Lisp, Grammar, HTML, Java, Lex, Lisp, M4, Objective-C, Pascal, Perl, Prolog, Python, RelaxNG, Unix shell, SRecode, Termcap, Windows Batch, and XML. Next, we estimated the development community from which replacement developers can be sought, at one-month intervals. To filter out inactive contributors, at each point in time (e.g., February 2002), we considered that the community (per programming language) consisted of those developers who performed at least one change to a source code file (implemented in that language) in the past six months (e.g., between September 2001 and February 2002). Finally, we computed $risk_\ell$ at one month intervals.

We discuss four representative examples (Figure 1). Detailed plots for all 26 languages are available in the online appendix[8]. We start with Unix shell (top left). The risk measure and the percentage of non-speakers are very close, i.e., evolution of the risk measure can be explained predominantly by the evolution of the percentage of non-speakers. The increasing trend observed from 1991 to 2000, followed by the stabilisation from 2000 onwards reflects the diminishing proportion of Unix shell developers. Moreover, very low values of the dotted blue line indicate that Unix shell is not commonly known by developers programming

[8] http://www.win.tue.nl/~bvasiles/emacs/risk.html

in popular languages (e.g., Emacs Lisp and C). Emacs Lisp (top right) exhibits similarly close values of the risk measure and the percentage of non-speakers, but both values are low (below 0.2 starting from 1998). The lion's share of the development community is, hence, familiar with Emacs Lisp. In contrast, Python (bottom left), although spoken by a similarly small fraction of the Emacs community as Unix shell (dashed red line), exhibits much lower risk (black line). The high values for the difference between the two time series (dotted blue line), relatively stable in time, indicate that Python is commonly known by developers programming in popular languages. Indeed, $mi_{Python}(EmacsLisp) \simeq 0.46$, $mi_{Python}(Lisp) \simeq 0.44$, and $mi_{Python}(C) \simeq 0.23$. C (bottom right) is spoken by approximately half of the Emacs community and is also commonly known by developers programming in Lisp (0.39) and Emacs Lisp (0.38), resulting in very low risk. Emacs Lisp exhibits a similar pattern, with the difference that its low risk is mostly due to the large share of Emacs Lisp speakers within the community rather than high similarity with the other languages.

6 Conclusions

Inspired by linguistic diversity measures, we proposed a method to quantify the risk of not finding developers who can maintain code implemented in a certain programming language, and empirically illustrated it using a case study. Our method takes into account similarities between programming languages, for which we have proposed a novel measure based on shared knowledge of the developers participating in StackOverflow. By tracking the evolution of such a risk measure as projects evolve (e.g., in a dashboard-like application), risky languages can be discovered on time, and preventive action can be taken to ensure the maintainability of components implemented in those languages.

We believe the results obtained so far to be a promising start. Our new dimension to risk assessment, bordering software maintenance and the social sciences, *does* offer additional insights into the evolution of a software system, and *does* open up many avenues for future research. For example, to offer a more complete understanding of evolution, our risk model should be refined to incorporate the number of artifacts in a certain programming language (the risk seems higher if a large proportion of a system is implemented in a risky language), their role (examples may be less important than the core implementation), their stability as reflected in a version control system (files not changed for a long time seem less risky than recently changed ones, cf. [25]) or their algorithmic or linguistic complexity (files implementing more complex behaviour or using more exotic language features seem to be more risky). We also plan to include a more refined language-tag mapping such that tags corresponding to versions and dialects (e.g., `c#-2.0` and `swi-prolog`) can be accounted for. A further refinement would include distinction between tags representing different technologies (e.g., `ejb`, `hibernate` and `swing`). Apart from being all implemented in Java, such technologies share little in common and likely require different skills to maintain.

We also would like to introduce a project-level risk measure $risk_P$, being the maximum of $risk_\ell$ for all languages ℓ in the project P, indicating the highest risk of not having developers who can take over code implemented in a certain language. Similarly, for a given project P we can identify and rank developers that—should they decide to leave P—would contribute most to increase of $risk_P$. This ranking can be seen as an alternative interpretation of the "bus factor" [10] and a way to quantify the developers' contributions [3].

Beyond the boundaries of *linguistic diversity* is the general concept of diversity, and its measurement in several biological, physical, social, and management sciences [24]. Some of these techniques have recently been applied in context of software engineering as well [21, 31, 35]. A detailed comparison of these techniques with the risk measure proposed in the current paper is considered as future work.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Very Large Data Bases. pp. 487–499. Morgan Kaufmann (1994)
2. Brijs, T., Vanhoof, K., Wets, G.: Defining interestingness for association rules. Information Theories & Applications 10(4), 370–375 (2003)
3. Capiluppi, A., Serebrenik, A., Youssef, A.: Developing an h-index for OSS developers. In: Lanza, M., Di Penta, M., Xi, T. (eds.) MSR. pp. 251–254. IEEE (2012)
4. Delorey, D., Knutson, C., Giraud-Carrier, C.: Programming language trends in open source development: An evaluation using data from all production phase Sourceforge projects. In: WoPDaSD (2007)
5. Doyle, J.R., Stretch, D.D.: The classification of programming languages by usage. Man-Machine Studies 26(3), 343–360 (1987)
6. Ducheneaut, N.: Socialization in an open source software community: A socio-technical analysis. Computer Supported Cooperative Work 14(4), 323–368 (2005)
7. Fearon, J.D.: Ethnic and cultural diversity by country. J. Econ. Growth 8(2), 195–222 (2003)
8. Gelernter, D., Jagannathan, S.: Programming linguistics. MIT Press (1990)
9. Giuri, P., Ploner, M., Rullani, F., Torrisi, S.: Skills, division of labor and performance in collective inventions: Evidence from open source software. International Journal of Industrial Organization 28(1), 54–68 (2010)
10. Goeminne, M., Mens, T.: Evidence for the Pareto principle in Open Source Software Activity. In: SQM. CEUR-WS workshop proceedings (2011)
11. Greenberg, J.: The measurement of linguistic diversity. Language 32(1), 109–115 (1956)
12. Handel, Z.: What is Sino-Tibetan? Snapshot of a field and a language family in flux. Language and Linguistics Compass 2(3), 422–441 (2008)
13. Heggarty, P.: Beyond lexicostatistics: How to get more out of word list comparisons. Diachronica 27(2), 301–324 (2010)
14. Hemetsberger, A., Reinhardt, C.: Learning and knowledge-building in open-source communities a social-experiential approach. Management Learning 37(2), 187–214 (2006)
15. Jepsen, T.C.: Just what is an ontology, anyway? IT Professional 11(5), 22–27 (2009)
16. Jones, C., Jones, T.: Estimating software costs, vol. 3. McGraw-Hill (1998)

17. Jones, C.: Applied Software Measurement: Global Analysis of Productivity and Quality. McGraw-Hill (2008)
18. Karus, S., Gall, H.: A study of language usage evolution in open source software. In: MSR. pp. 13–22. ACM (2011)
19. Kouters, E., Vasilescu, B., Serebrenik, A., van den Brand, M.G.J.: Who's who in Gnome: Using LSA to merge software repository identities. In: ICSM. pp. 592–595. IEEE (2012)
20. Moberg, J., Gooskens, C., Nerbonne, J., Vaillette, N.: Conditional entropy measures intelligibility among related languages. In: Proceedings of Computational Linguistics in the Netherlands. pp. 51–66 (2007)
21. Mordal, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., Ducasse, S.: Software quality metrics aggregation in industry. Software: Evolution and Process (2012)
22. Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., Ye, Y.: Evolution patterns of open-source software systems and communities. In: IWPSE. pp. 76–85. ACM (2002)
23. Neumann, D.E.: An enhanced neural network technique for software risk analysis. IEEE Trans. Softw. Eng 28(9), 904–912 (2002)
24. Patil, G.P., Taillie, C.: Diversity as a concept and its measurement. Journal of the American Statistical Association 77(379), 548–561 (1982)
25. Poncin, W., Serebrenik, A., van den Brand, M.G.J.: Process mining software repositories. In: CSMR. pp. 5–14. IEEE (2011)
26. Posnett, D., D'Souza, R., Devanbu, P., Filkov, V.: Dual ecological measures of focus in software development. In: ICSE. pp. 452–461. IEEE (2013)
27. Rechenberg, P.: Programming languages as thought models. Struct. Program. 11(3), 105–116 (1990)
28. Robles, G., González-Barahona, J.M.: Contributor turnover in libre software projects. In: Open Source Systems. vol. 203, pp. 273–286. Springer (2006)
29. Robles, G., González-Barahona, J.M., Merelo, J.J.: Beyond source code: the importance of other artifacts in software development (a case study). Journal of Systems and Software 79(9), 1233–1248 (2006)
30. Schildt, H.: C/C++ Programmer's Reference. McGraw-Hill, 2nd edn. (2000)
31. Serebrenik, A., van den Brand, M.G.J.: Theil Index for Aggregation of Software Metrics Values. In: ICSM. pp. 1–9. IEEE (2010)
32. Stallman, R.M.: EMACS the extensible, customizable self-documenting display editor. SIGPLAN Not. 16(6), 147–156 (1981)
33. Swadesh, M., Sherzer, J., Hymes, D.: The Origin and Diversification of Language. Adeline Transaction (1971)
34. Vasilescu, B., Filkov, V., Serebrenik, A.: StackOverflow and GitHub: associations between software development and crowdsourced knowledge. In: SocialCom. ASE/IEEE (2013), accepted
35. Vasilescu, B., Serebrenik, A., van den Brand, M.G.J.: You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In: ICSM. pp. 313–322. IEEE (2011)
36. Vasilescu, B., Serebrenik, A., Devanbu, P., Filkov, V.: How social Q&A sites are changing knowledge sharing in Open Source software communities. In: CSCW. ACM (2014), accepted
37. Vasilescu, B., Serebrenik, A., Goeminne, M., Mens, T.: On the variation and specialisation of workload–A case study of the Gnome ecosystem community. Empirical Software Engineering pp. 1–54 (2013)
38. Watt, D.A., Findlay, W.: Programming language design concepts. Wiley (2004)