Reducing Redundancies in Multi-Revision Code Analysis

Alexandru, Carol V ; Panichella, Sebastiano ; Gall, Harald C

Abstract: Software engineering research often requires analyzing multiple revisions of several software projects, be it to make and test predictions or to observe and identify patterns in how software evolves. However, code analysis tools are almost exclusively designed for the analysis of one specific version of the code, and the time and resources requirements grow linearly with each additional revision to be analyzed. Thus, code studies often observe a relatively small number of revisions and projects. Furthermore, each programming ecosystem provides dedicated tools, hence researchers typically only analyze code of one language, even when researching topics that should generalize to other ecosystems. To alleviate these issues, frameworks and models have been developed to combine analysis tools or automate the analysis of multiple revisions, but little research has gone into actually removing redundancies in multi-revision, multi-language code analysis. We present a novel end-to-end approach that systematically avoids redundancies every step of the way: when reading sources from version control, during parsing, in the internal code representation, and during the actual analysis. We evaluate our open-source implementation, LISA, on the full history of 300 projects, written in 3 different programming languages, computing basic code metrics for over 1.1 million program revisions. When analyzing many revisions, LISA requires less than a second on average to compute basic code metrics for all files in a single revision, even for projects consisting of millions of lines of code.

DOI: https://doi.org/10.1109/SANER.2017.7884617

Originally published at:
Alexandru, Carol V; Panichella, Sebastiano; Gall, Harald C (2017). Reducing Redundancies in Multi-Revision Code Analysis. In: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20 February 2017 - 24 February 2017, IEEE.
DOI: https://doi.org/10.1109/SANER.2017.7884617

Reducing Redundancies in Multi-Revision Code Analysis

Carol V. Alexandru, Sebastiano Panichella and Harald C. Gall
Software Evolution and Architecture Lab
University of Zurich, Switzerland
{alexandru,panichella,gall}@ifi.uzh.ch

***Abstract*—Software engineering research often requires analyzing multiple revisions of several software projects, be it to make and test predictions or to observe and identify patterns in how software evolves. However, code analysis tools are almost exclusively designed for the analysis of one specific version of the code, and the time and resources requirements grow linearly with each additional revision to be analyzed. Thus, code studies often observe a relatively small number of revisions and projects. Furthermore, each programming ecosystem provides dedicated tools, hence researchers typically only analyze code of one language, even when researching topics that should generalize to other ecosystems. To alleviate these issues, frameworks and models have been developed to combine analysis tools or automate the analysis of multiple revisions, but little research has gone into actually *removing* redundancies in multi-revision, multi-language code analysis. We present a novel end-to-end approach that systematically avoids redundancies every step of the way: when reading sources from version control, during parsing, in the internal code representation, and during the actual analysis. We evaluate our open-source implementation, 𝕃ISA, on the full history of 300 projects, written in 3 different programming languages, computing basic code metrics for over 1.1 million program revisions. When analyzing many revisions, 𝕃ISA requires less than a second on average to compute basic code metrics for all files in a single revision, even for projects consisting of millions of lines of code.**

I. Introduction

Static software analysis has a broad range of applications and is omnipresent both in software engineering research and practice. Professionals use software analysis towards goals such as enforcing coding guidelines, allocating resources and estimating future effort [22], [39], [49], [67]. It also aids in development, for example by means of refactoring suggestions [14], [22], [53], [64] or bug detection [35], [56]. Practitioners most often apply their tools to the current version of their software, be it during development in an IDE or during continuous integration. Their tools often grow naturally out of needs within a software ecosystem and are tailored to the peculiarities of a specific language [11], [12], [15], [51]. With regard to multi-language analyses, practitioners also show growing interest in [61], given that an increasing amount of software is written in multiple languages (like web applications and multi-platform mobile apps) [9], [52], [54].

Researchers on the other hand use code analysis to discover new patterns and anti-patterns in existing code [45], [55], [66], to learn more about software evolution [18], [24], [26], [50], [57] and to develop new methods for change and bug prediction [22], [35], [38], [56]. In this context, tools developed for practical application are not always ideal, because researchers expect high generalizability [6], [8], [13], [20], [59], [60], [70] and replicability [28] from their results, yet the analysis of large samples using existing methods is hard to automate and replicate for a number of reasons, which we discuss in the following section.

A. Challenges

In the context of software engineering and evolution research, the selection of software projects suited for a particular investigation and the collection of code metrics are important steps toward large and replicable case studies. However, researchers need to overcome several technical and practical obstacles. First and foremost, software analysis for large projects is *costly in terms of time and resources* [17], [20], [21]. Moreover, nowadays software applications are written in multiple programming languages [9], [52], [54] and several issues with studying multi-language software have been identified by researchers and practitioners alike [9], [40], [54]. It is extraordinarily *hard to conduct generalizable code studies involving multiple languages* and attempts at solving this problem so far incur even higher performance penalties.

Manual effort. When the goal is to gain a broad overview on the evolution of many projects, the effort required to set up and run analyses for each individual project and revision presents a major problem. Tools which run on the compiled application are largely unsuitable for this kind of investigation, because the compiled version is rarely available for each revision and would need to be prepared manually. Automated build approaches so far exhibit limited effectiveness because every project tends to be unique in terms of the build environment and dependencies needed for compilation [17], [21], [31], [47], [48]. *Ideally, software evolution analysis tools should operate automatically for the most part.*

Redundant analysis of unchanged source code. The majority of existing code analysis approaches are not designed with a multi-revision use-case in mind [9], [40], [54]. They can only analyze one revision at a time and require one execution per revision, even if each commit makes only minute changes to the program. Researchers may be able to take advantage of tools that operate on the file level to only analyze files that change, but the ratio of changed to unchanged code within a single file is still very small and the majority of the analysis will still be redundant. *Ideally, a software evolution analysis tool should operate on the smallest possible delta between code revisions.*

Source data duplication. Most code analysis tools operate on files and folders [17], [20], [21]. While single revisions of even a large project remain manageable, the need to checkout thousands of revisions can hinder effective parallelization. *Ideally, the analysis of code contained in repositories should occur in-place.*

Isolation of programming ecosystems. Many code studies involve only one programming language (e.g., [9], [13], [40], [50], [54], [59], [60]). Undoubtedly, this is, in part, due to the fact that every programming ecosystem comes with its own set of tools and any data gathering effort is duplicated with each additional language. Several successful approaches for performing studies on the evolution of software projects [24], [25], [57] and bug prediction [37], [38], [44] limit their applicability to a single programming language. *Ideally, software analysis approaches should be easily transferable to multiple programming languages.*

Mismatch of concepts in different languages. To enable the analysis of multiple languages using common tooling, researchers have devised meta-models as a common representation for source code of different languages [9], [40], [54]. These attempt to tie language-specific entities and structures to a common representation.

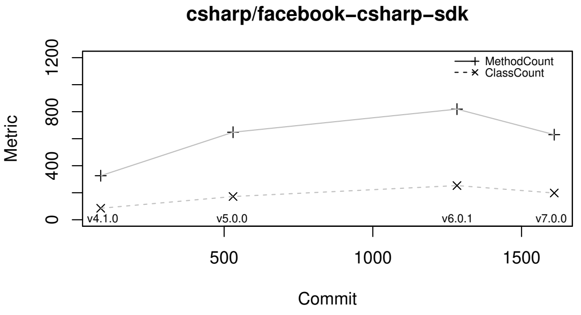

Fig. 1. Method Count and Class Count for 4 releases of the facebook-csharp-sdk project.

However, each language exhibits unique features, hence accurate mappings are not always obvious and depend on the goals of each individual study or approach. *Ideally, code analysis tools should genericise over arbitrary structures and cross- or multi-language analyses should be enabled without having to adhere to a specific, pre-defined meta-model.*

Low-resolution historical data. Given all previous obstacles, the majority of code evolution studies not only use a small sample of projects of a single language, but also choose to analyze only a limited number of (often major) releases [19], [52], [54]. Considering that modern software development typically sees several commits per day for a given project [7], [30], analyzing only a few releases can give an incomplete picture. To give an example, fig. 1 shows the Method Count and Class Count computed for 4 major releases of the facebook-csharp-sdk open source project. A preliminary interpretation of the graph suggests a rather steady rise and fall in the metrics of the source code as a whole. However, we cannot make any claims on how the software project evolved on a shorter time scale. We will revisit this example in our evaluation.

B. Goal and Contributions

To presently conduct large scale code studies over the full history of many projects and concerning millions of commits, researchers need to either invest a lot of resources for parallelization (e.g., [20]) or spend an excessively long period of time on the analyses (e.g., [66]). With regard to previous work outlined in section V, software engineering research is yet lacking an approach that can capture the entire evolution of many software projects in different languages quickly and without excessive manual effort. The goal of this paper is to provide a framework allowing researchers to overcome the limitations of code analysis that currently hinder the degree of generalizability and replicability of their studies. Specifically, we condense the challenges outlined in section I-A into two main research questions:

- **RQ**$_1$: *How can redundancies in multi-revision program analysis be minimized?*
- **RQ**$_2$: *How can we abstract over different programming languages in the context of high speed code analysis?*

Even though these are two distinct problems, imposing the restrictions of both problems onto a solution requires us to consider overlaps and mutual exclusions between the two. For example, performing multi-revision analyses for only one specific language is a very different goal from constructing a framework that accelerates code analysis for arbitrary languages, as it may allow the re-use of language-specific tools and data structures, such as a compilers and language-specific code representations. On the other hand, meta-models tend to be heavy-weight under most circumstances, whereas the high-speed requirement forces us to find a more light-weight solution. Ultimately, we believe that a generic solution is of greater benefit for the scientific community. That said, many of the techniques we present in this paper could be separated and used independently.

Our contributions can be summarized as follows:

1) We develop a novel algorithm for loading and analyzing a large number of different revisions of a project asynchronously and with minimum redundancy, enabling the concurrent analysis of thousands of revisions at minimal marginal cost for each additional revision.
2) We introduce light-weight entity mappings, a novel and flexible method for easily formulating ad-hoc mappings between language-specific entities and cross-language semantic concepts, without the requirement for an actual translation to a pre-defined meta-model.
3) We provide the research community with an open-source implementation of our approach, called $\mathbb{L}$ISA[1] (Lean Language-Independent Software Analyzer), as well as a replication package including 8.7 GB worth of fine-grained code measurements for the projects analyzed in our study[2].

As a secondary contribution derived from our evaluation, we determine the necessary sampling interval to accurately represent the overall evolution of projects written in different languages in the context of sparse evolution analyses.

C. Paper Structure

In sections II-A to II-C we present a multi-revision data structure and an algorithm for loading source code of thousands of revision into a common representation. In section II-D, we introduce light-weight entity mappings, an easy way to generalize code analysis to multiple languages in the context of high performance software analysis. We evaluate our implementation in section III. Section IV discusses limitations of our approach and section V contains related work. We conclude the paper in section VI.

II. Approach

In the following sections, we describe several novel techniques which vastly reduce the redundancies inherent to analyzing multiple revisions of a program. This includes a multi-version graph representation of source code, an asynchronous parsing and deduplication algorithm and a highly flexible abstraction that enables the analysis of multiple programming languages using the same analysis instructions. We combine these techniques in $\mathbb{L}$ISA, although they could be applied independently to improve different aspects of existing applications.

A. Multi-Revision Graph Representation of Source Code

In this section, we describe a condensed graph representation which allows us to analyze the source code of thousands of revisions simultaneously.

Graph representation. We define a directed graph as an ordered pair $G = (V, E)$, consisting of a vertex set V and an edge set E. Each vertex $v \in V$ is a tuple (i, M_v) consisting of a unique identifier i and a map M_v containing metadata on the vertex in the form of (k, m) key-value pairs. Every edge $e \in E$ is a tuple (s, d, M_e) identified by the source and destination vertices s and d and can also contain a map M_e with metadata describing the edge. Such a graph is commonly used to model various representations of a program. For example,

[1] https://bitbucket.org/sealuzh/lisa

[2] http://tiny.uzh.ch/C0

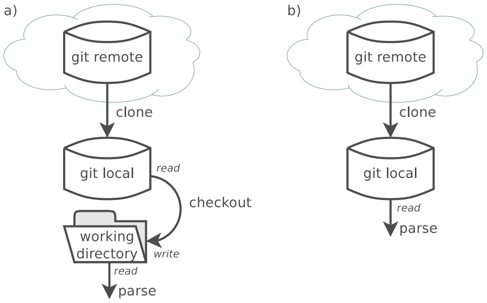

Fig. 2. Contrary to the traditional method of checking out (i.e., reading and writing) *all* files in each revision and then parsing the source code from the filesystem (a), parsing the source code directly from a bare Git repository requires only a single read operation per *relevant* file (b).

using a language grammar, one can create a concrete syntax tree (CST) for each file in a project, such that every vertex v represents a symbolic token in the original source code, connected to zero or more child vertices, populating the graph with disconnected trees (one for each file). In this case, vertices identify by their file name and syntax tree path, e.g., `/src/Main.java/CompilationUnit0` and store their literal token as metadata, while edges store no metadata. Similarly, one can build an abstract syntax tree (AST), where multiple tokens are interpreted and combined into higher-level entities. In that case, vertices also store additional metadata depending on the vertex type. For example, an AST vertex may represent a Java class and store metadata such as its name and visibility. A graph is also suitable for modeling a compiled program, in which case relationships between different parts of the program can be represented by additional edges containing metadata about the relationship, e.g., whether it describes attribute access or a method call. The metadata also serves as a container for additional data created during analysis. For example, a vertex representing a class may contain a method count as part of the metadata.

Both the graph model and its implementation are agnostic regarding the type of representation. $\mathbb{L}$ISA uses Signal/Collect [62], a low-level graph framework, to store the graph and exposes an interface with just two members for clients to implement: a definition of suffixes of file names that the client supports and a *parse* routine, which is provided with the content of a file and an agent for adding vertices and edges to the graph. This means that any pre-existing parsers can be used in conjunction with $\mathbb{L}$ISA and any kind of revisional data can be represented. Hence, the data contained in M_v and M_e is determined by the kind of graph that is loaded and by the analyses performed. If parse errors occur, it is up to the parser to handle them, e.g., by ignoring the file for that particular commit. Our implementation ships with adaptors for the JDK *javac* parser and for ANTLRv4. ANTLR is a parser generator for which existing grammars can be found for many languages[3]. Given an ANTLR grammar, it is possible to create a new $\mathbb{L}$ISA-compatible parser with only a few lines of boilerplate code.

Asynchronous code-loading. Differently from traditional tools, which check out individual revisions from the version control system to act on source code contained in files and folders, our implementation encourages direct, asynchronous access to the sources of multiple revisions. Our `SourceAgent` interface, which we implement for Git, mandates the preliminary creation of key-value pairs (p, B) for each file in any revision of the project, where p identifies a file by its path and B denotes a chronologically ordered sequence of sources for the file in different revisions. This way, each pair can be assigned to one of many parallel workers, which read the sources for a single file sequentially, while sources for different files can be read independently from each other. In Git, this extraction step can be performed in a single traversal of the Git graph database, without the need to ever check out a working copy of the code, as illustrated in fig. 2. That said, this technique will work with any version control system that allows direct access to individual revisions of files, which is a fairly standard feature.

This approach has four distinct performance benefits over the traditional approach: Avoiding the checkout to a working directory first reduces the number of reads for each relevant file from 2 to 1, because each blob is only read once during parsing and never for a checkout, and it reduces the number of writes from 1 to 0, since blobs are never re-written to the file system. Second, it avoids the unnecessary checkout of any other assets which are not even relevant for the analysis. Third, it allows for the asynchronous, parallelized parsing of many revisions simultaneously, which is otherwise only possible by checking out multiple copies of the entire source code to different locations. Finally, files are only read in revisions where they underwent actual change.

Multi-revision graph representation. Typically, a single commit only changes a tiny part of an application. This means that the graph representations of two adjacent revisions overlap to a large degree and that most vertices are unchanged for many consecutive revisions (thousands, in practice). To avoid creating separate graphs for each revision, we extend our graph representation such that each vertex tuple (i, R, M_{vR}) carries an additional set R denoting one or more *revision ranges*. A revision range $r(s, e)$ consists of start and end revisions s and e and indicates in which revisions the vertex exists. Likewise, the metadata M_{vR} of a vertex now stores metadata for each revision range that the vertex represents. Edges on the other hand do not require any special treatment: whether or not an edge exists in a revision is determined by whether or not the two connecting vertices exist. In this paper, we will generally denote a range as two dash-separated numbers in brackets, e.g., a revision range with start 10 and end 15 is denoted as [10-15]. No information is lost using this graph representation, as the structure of a single revision can still be identified simply by selecting only vertices whose revision ranges contain the desired revision. Even so, it saves a tremendous amount of space, because the majority of the graph remains unchanged for much of the entire evolution of a project.

B. Low Redundancy Graph and Metadata

The interface provided by $\mathbb{L}$ISA for clients to populate the graph is transparent with regard to this multi-revision representation. When a client reads a file for a particular revision and adds a vertex with an identifier i to the graph, $\mathbb{L}$ISA transparently handles the case where the vertex already exists and adds or extends a revision range inside the existing vertex to accommodate the new revision. This has several important consequences:

1) An entity represented by the same vertex may contain differing metadata in different revisions. A simple example for this are literal tokens: There may be an "Integer" syntax vertex, whose metadata contains the number 5 as a literal in one revision, but 7 in another. In this case, two separate, adjacent ranges with different metadata are necessary to accurately represent both revisions using the same graph vertex.

[3] For example, https://github.com/antlr/grammars-v4 contains grammar files for over 60 structured file formats.

2) If the same file was parsed in two adjacent revisions (e.g., 5 and 6), then it makes sense to merge the ranges [5-5] and [6-6] into a common range [5-6] for all vertices that underwent no change.
3) If a file has not undergone any changes in a revision, then, as per the previous section, it will not be parsed, leading to erroneous gaps in the revision ranges of a vertex. For example, if a file underwent changes in revisions 5 and 10, it will not be parsed in revisions 6 to 9, resulting in two ranges [5-5] and [10-10]. But the syntax nodes within are actually present in the source in-between these revisions, necessitating a range extension for a final configuration of [5-9][10-10], or even [5-10] if the two parsed revisions contain the same metadata in the given vertex.
4) Finally, the metadata of a vertex for two adjacent revisions may also exhibit some overlap. Imagine a vertex representing a method during a computation which calculates the cyclomatic complexity and statement count of individual entities. Between two adjacent revisions, the method may have the same name and complexity, but a different statement count. In this case, two separate ranges are necessary to describe the vertex, yet most of the metadata is shared.

In the following section, we describe the data structures and algorithm used to efficiently build and store the revision-specific data in such a fashion that *neither vertices nor metadata are unnecessarily duplicated.*

Sparse graph loading algorithm. We explain the algorithm, which is shown in algorithm 1, by following the evolution of a single syntax tree vertex v of an exemplary project with just 10 revisions, as illustrated in table I. v is parsed from a file f which Git stores as a blob for each revision where it underwent any change. The first row in table I indicates the different revisions of the project, enumerated sequentially starting at 0. The letters in the second row denote unique metadata configurations that exist in v. In this example, the node has the same metadata in revisions 0 to 3, then the node is deleted in revision 4, then it reappears with different metadata in revisions 6 and then the metadata is again changed in revisions 8. The third row indicates whether the file f has been affected by a revision. In this example, f is parsed in revisions 0, 1, 3, 4, 5, 6 and 8. This implies that in revisions 2, 7 and 9 there were no changes in f, meaning that all vertices that existed in previous revisions (1, 6 and 8) are unchanged and still exist, even if the file wasn't re-parsed. In revisions 4 and 5, f was parsed, but v did not appear in the source code of those revisions.

Algorithm 1 𝕃ISA range compression algorithm. The $ranges$ variable is a map from revision ranges to metadata and contains all metadata for a single vertex.

```
procedure UPDATEVERTEX($v, rev, meta$)
    CREATERANGE($v.ranges, rev, meta$)
    $queue$(COMPRESSRANGES($v.ranges, v.mark, rev - 1$))
procedure CREATERANGE($ranges, rev, meta$)
    $new \leftarrow ([rev, rev] \rightarrow meta)$
    $ranges \leftarrow ranges + new$
procedure COMPRESSRANGES($ranges, mark, end$)
    if $end = null$ then                                        ▷ Case 1
        $mark \leftarrow ranges.head$
    else
        $next \leftarrow ranges.get(end + 1)$
        if $mark = null$ & $next = null$ then                    ▷ Case 2
            $doNothing$
        else if $mark = null$ & $next \neq null$ then            ▷ Case 3
            $mark \leftarrow next$
        else if $mark \neq null$ & $next = null$ then            ▷ Case 4
            $extended \leftarrow ([mark.start, end] \rightarrow mark.meta)$
            $ranges \leftarrow ranges - mark + extended$
            $mark \leftarrow null$
        else if $mark \neq null$ & $next \neq null$ then
            if $mark.meta = next.meta$ then                      ▷ Case 5
                $merged \leftarrow ([mark.start, next.end] \rightarrow mark.meta)$
                $ranges \leftarrow ranges - mark - next + merged$
                $mark \leftarrow merged$
            else if $mark.meta \neq next.meta$ then              ▷ Case 6
                $extended \leftarrow ([mark.start, end] \rightarrow mark.meta)$
                $ranges \leftarrow ranges - mark + extended$
                $mark \leftarrow next$
```

When a client adds a vertex to the graph and this vertex already exists, then 𝕃ISA updates the vertex (UPDATEVERTEX in algorithm 1) by simply adding another single-revision range and queuing a range compression task leading up to the preceding revision. Hence, the range compression itself also runs asynchronously and "cleans up" behind the parser, compressing ranges added by the parser. For example, the parser may already have parsed the first 4 revisions (0 to 3), meaning that it will have created 3 single-revision ranges [0-0], [1-1] and [3-3] in vertex v. Since the metadata of v is the same in all these revisions, including revision 2, where no changes were made to the file at all, the goal of the range compression algorithm is to combine all these ranges into a single range [0-3].

We now step through the execution of the compression algorithm for each revision of the example vertex. Note that for each possible *case* in the algorithm, table I contains the case number in the last column. Also note that "$marking$" a range simply means saving a reference to that range as a temporary property of the vertex. This $mark$ indicates whether a revision range is "open-ended" in the sense that it may be extended in the next revision. After parsing revision 0, the compression algorithm simply needs to $mark$ the first existing range in v (case 1). After parsing revision 1, it finds a $marked$ range and an existing range at revision $end + 1 = 1$. It compares the two ranges, finds that they are equal (case 5), merges them and $marks$ the resulting range. Revision 2 saw no changes in f, so v remains unchanged by the parser. Note however that in reality, the code represented by v continued to exist, hence the revision range needs to be extended. This happens in revision 3, where f was

TABLE I
METADATA OF AN EXEMPLARY GRAPH VERTEX.

Revision	0	1	2	3	4	5	6	7	8	9			
Content	A	A	A	A			B	B	C	C			
Touched	X	X		X	X	X	X		X				
Revision	0	1	2	3	4	5	6	7	8	9	*End*	*Mark*	*Case*
0	A										-	-	1
	A											[0-0]	
1	A	A									0	[0-0]	5
	A											[0-1]	
2	A											[0-1]	
3	A			A							2	[0-1]	5
	A											[0-3]	
4	A										3	[0-3]	4
	A											-	
5	A										4	-	2
	A											-	
6	A						B				5	-	3
	A						B					[6-6]	
7	A						B					[6-6]	
8	A						B		C		7	[6-6]	6
	A						B		C			[8-8]	
9	A						B		C			[8-8]	
final	A						B		C		9	[8-8]	4
	A						B		C			-	

modified again. The algorithm finds the $marked$ range and a range at $end+1 = 3$ containing the same metadata, prompting a merge (case 5). In revision 4, there exists a $marked$ range, but there is no range at $end + 1 = 4$. A merge is not necessary, so it just $unmarks$ the existing $marked$ range (case 4). In revision 5, there is no $marked$ range and no $next$ range, so nothing happens (case 2). This simply means that even though f had undergone changes, they did not affect v, which does not exist in end or in $next$. In revision 6, there is no $marked$ range, but there is a $next$ range, so it is $marked$ (case 3). Revision 7 saw no changes. In revision 8, there is a $marked$ range and a $next$ range, but they are not equal, so the $marked$ range is extended until end and then the $next$ range is $marked$ instead (case 6). Revision 9 saw no changes. Once all revisions have been parsed, a final pass is made through *all* vertices in the graph, extending any $marked$ ranges to include the last revision. This is necessary because these $marked$ ranges previously ended with the revision where the file was last *parsed*, while these vertices actually exist until the end of the project history. In this example, the $marked$ range is extended to include the last revision (case 4).

As a result of the compression, instead of storing metadata for each revision (eight in case of v) only three ranges with different metadata remain (visible in the final row of table I). In practice, revision ranges tend to comprise a much larger number of revisions. If a file is added to a project in an early revision, it can be that the majority of vertices within exhibit the same metadata for thousands of subsequent revisions.

Range splitting and shared metadata. After loading the graph, the initial metadata for a vertex will likely be contained in a small number of revision ranges. However, once the graph is being analyzed (as detailed in section II-C), the metadata in different ranges may start to diverge to a certain degree. For example, a “Class” node might exist in revisions 5 to 280 resulting in a single all-encompassing revision range [5-280], but its attribute count could be computed as 4 in the first 100 revisions and as 7 in the remaining revisions. This means that the revision range [5-280] needs to be split into [5-105][106-280], and that separate metadata needs to be stored for those two revision ranges. 𝕃ISA performs these splits dynamically during the computation whenever necessary.

However, storing completely separate metadata for each revision range would again introduce significant data duplication: in our current example, only the attribute count will be different for the two ranges. Hence, to avoid data duplication, we use referentially transparent, immutable data structures to store the metadata. Figure 3 shows how metadata for the vertex is stored pre- and post split. No data is copied upon the split, since both new ranges share references to those metadata which have not changed. In fact, the default state for any vertex (which is defined by the user depending on the analysis) exists only once in the entire computation and all ranges only store a reference to it. The concrete implementation for this uses Scala *case classes* for storing data, an example for which can be seen in line 4 of listing 1.

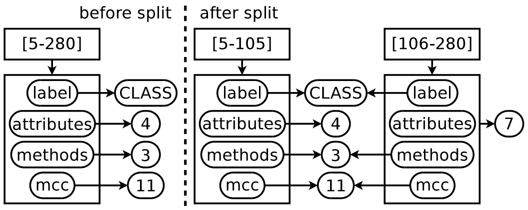

Fig. 3. Range splitting during the computation does not duplicate any data thanks to the use of immutable data structures for storing analysis data.

C. Multi-Version Code Analysis

Signal/Collect, the graph framework used in 𝕃ISA, operates under a specific computational paradigm, where vertices in the graph communicate with each other to form an emergent computation. More specifically, each vertex can *signal* other vertices on outgoing edges to transmit arbitrary messages, *collect* incoming signals and *modify* its own state. We ask the reader to refer to [62] for an in-depth description of the paradigm.

To formulate analyses in 𝕃ISA, the user defines a *data structure* for analysis data (which 𝕃ISA integrates into the metadata of a vertex), a *start* function governing how and where in the graph the first signals are emitted for this particular analysis, and an *onCollect* function, which determines how the analysis processes incoming signals. Both these functions are *side effect free* and return ‘new’ metadata to reflect changes (however, referential transparency of case classes in Scala prevents unnecessary data duplication). As an example, listing 1 contains an implementation of cyclomatic complexity [46]. The data structure used by this analysis (defined on line 4; initialized on line 5) contains a boolean to determine whether or not to persist the computed value for a given vertex and a single integer value to hold the actual complexity value. The start function causes an `MccPacket` to be sent out from each *leaf* vertex in the graph (lines 9-11). The value contained in the packet is 1 by default, or 2 if the leaf vertex itself is already a node that increases the complexity of the program. In regular programs, having a leaf node contribute to the complexity is impossible (e.g., there cannot be an ‘If’ statement with no truth check beneath it in the tree), but as we will see in the next section, 𝕃ISA filters irrelevant nodes during parsing, such that the entire tree below a branching statement may be omitted from the graph. The `onCollect` function accumulates incoming values according to the definition of cyclomatic complexity (line 20) and once all child values have been received, it sends the complexity of the local vertex to its parent (closure starting at line 24). During the

Listing 1
Implementation for McCabe’s Complexity in 𝕃ISA

```
1  object MccAnalysis
2  extends Analysis
3  with ChildCountHelper {
4    case class Mcc(persist: Boolean, n: Int)
5    extends Data { def this() = this(false, 1) }
6
7    override def start = {
8    implicit domain => state =>
9      if (leaf(state))
10       state ! new MccPacket(
11         if (state is 'branch) 2 else 1)
12     else state
13   }
14
15   class MccPacket(val count: Int)
16   extends AnalysisPacket {
17     override def onCollect = {
18     implicit domain => state =>
19
20       val mcc = state[Mcc].n + count - 1
21
22       allChildren[Mcc](state)(
23         incomplete = state + Mcc(false, mcc),
24         complete = {
25           val newCount =
26             if (state is 'branch) mcc + 1 else mcc
27           val persist =
28             state is ('class, 'method, 'unit)
29           state + Mcc(persist, mcc)
30                 ! new MccPacket(newCount) })}}}
```

Listing 2
DEMO.JAVA

```java
package org.example;

public class Demo {
  public void run() {
    for (int i = 1; i < 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
        System.out.println(i);
      }
    }
  }
}
```

analysis, a transient value is computed and stored in the metadata of each vertex in the graph as a consequence of this algorithm, but the analyst would likely want to know the complexity only for specific kinds of nodes. The determination, whether the `mcc` value for a given vertex should be persisted is done on line 28, where the `persist` property is set `true` if the vertex is a class, method or file. The user could easily add other vertex types (e.g., blocks or closures) if needed.

Once an analysis is completed, the analysis results are persisted using a user-provided persistence strategy, for example to store values in a database for further processing. 𝕃ISA ships with a persistor that will dump all data, for which persistence has been enabled, into a sparse CSV file, where each line corresponds to a vertex in a given range and each column corresponds to one kind of data. Where data has not been computed for a specific vertex, the cell remains empty.

Since the computations are executed directly on the range-compressed graph, calculations are executed only once for a revision range of any particular vertex and outgoing signals are also attached to a revision range. The revision range of a vertex may be split, as described in section II-A, if an incoming signal concerns a partial range of the receiving vertex. In this fashion, the number of computations necessary to compute metrics for individual commits is vastly reduced.

D. Multi-Language Analysis Genericism

Programming languages share certain concepts and many code-related analyses can be expressed for different languages. And while the concrete syntax tree representation can vary greatly for different languages and parsers, the *relative* structure can be fairly similar. For example, when comparing the ASTs parsed from a Java and a Python program, the exact sequence, labeling and nesting of vertices leading from a root node to the leaf nodes of a method may differ greatly. However, structural features (where the vertices are located relative to each other), in the context of code analysis (and not program compilation) are very similar: A method/function vertex that has a class vertex as a parent is contained in that class; or: two `if`s on the same level represent 4 possible paths through the local scope, while one `if` nested in another creates only 3 possible paths. This means that we primarily need to explicitly capture *entities*, while retaining the *structure* as offered by the parser. Furthermore, and especially in the context of large scale code analysis, the representation of the source code should capture only the minimum necessary for a particular analysis. Consider we wanted to compute the cyclomatic complexity for methods in a program: the example in listing 2 contains 140 AST vertices when parsed using ANTLR, yet most of them are entirely irrelevant towards the complexity metric.

The solution we propose to solve this problem is notably simple: It requires a one-way many-to-many relation $T \rightarrow L$ from the domain of entity types T required for a particular analysis, to the co-domain of parser-specific labels L used by a particular language. In 𝕃ISA, this relation is simply expressed as a map from symbols to sets of labels. Listing 3 shows an example which is sufficient for the complexity analysis in listing 1, which notably doesn not contain any language-specific labels. It specifies how the entities mentioned by the analysis (`'method` and `'branch`) can be identified in Java. Note that the analysis also checks for other potential symbols (`'class` and `'unit`), but specifying these is not mandatory. Applied to the `Demo` program, this mapping populates the graph with a mere 5 vertices connected in a straight line: one vertex each for the file `Demo.java` itself, the method `run`, the `for` loop, the `if` statement and the `||` operator. All other vertices (such as numbers and other operators) are automatically ignored at the parsing step. When applying `MccAnalysis` to this graph, the result (a complexity of 4, which is persisted only for the vertex matching the `'method` symbol) is still correct.

Listing 3
A VERTEX LABEL MAPPING FOR JAVA.

```scala
object AntlrJavaParseTree extends Domain {
  override val mapping = Map(
    'method -> Set("MethodDeclaration"),
    'branch -> Set("IfStatement", "ForStatement",
          "WhileStatement", "CatchClause",
          "SwitchLabel", "TryStatement",
          "DoStatement", "ConditionalExpression")
  )
}
```

Note that it is still possible to write language-specific analyses by specifying the explicit label used by the parser instead of a mapped symbol, in case a language-specific structure needs to be analyzed.

E. Approach Summary

Compared to traditional approaches, 𝕃ISA avoids redundancies at every juncture of multi-version analysis: It analyzes code taken directly from the Git database and only re-reads files containing changes, while user-defined mappings vastly reduce the number of vertices required to perform analyses on the loaded source code. Both code and metadata present in multiple revisions is stored only once and the computations themselves are also executed only once for each revision range at the vertex/subtree level, avoiding the expensive re-computation of data at the file level. Finally, the computed data is selectively persisted to keep the results manageable for further analysis. Furthermore, the user-defined mappings not only reduce the size of the graph, but also make analyses transferable to other languages, which 𝕃ISA can easily support via existing ANTLR grammars or through dedicated parsers.

III. EVALUATION

In RQ1, we ask how redundancy can be avoided when analyzing multiple revisions of a project. Our proposed approach, as summarized in section II-E, combines multiple novel techniques towards that goal and we want to know how effective they are. For RQ2, we want to know if we can analyze code from multiple programming languages using the same underlying framework and analysis formulations. We want to gauge the practicality of lightweight mappings, our proposed solution. To frame our evaluation in a practical research context, we attempt to answer a fundamental question with regard to software evolution analysis: *How many releases of a project do I actually need to analyze to gain an accurate picture of its evolution?*

Our evaluation is structured as follows. In section III-A, we conduct a large-scale study over the full history of 300 projects written in 3 different languages. Based on this, we discuss the performance of our approach and the effectiveness of redundancy removal techniques implemented in 𝕃ISA in section III-B and also discuss the practicality of light-weight mappings. In section III-C, we present the results of our artifact study. Finally, we discuss threats to our study in section III-D.

We provide a comprehensive replication package, including all scripts used in the evaluation as well as all the resulting data (8.7 GB worth of analysis results) online[4].

A. Code Study

Using research reports and statistics on popular Java and JavaScript projects [1], [5] as well as GitHub's "most starred" query option, we selected 100 Java, C# and JavaScript projects, ignoring projects that do not contain an application per se (such as tutorials and demo projects) and preferring projects with a larger number of commits (for a median of 2956 commits for Java, 2051 for C# and 1097 for JavaScript). We proceeded to formulate analyses to compute the number of classes, methods, method parameters, variables and statements, the cyclomatic complexity, control flow nesting depth and number of distinct control flow paths, and to detect the BrainMethod [42] code smell. We also compute the number of direct children and total number of vertices beneath each vertex as a proxy for the size of the code base. We fed 𝕃ISA with existing ANTLR grammars for all three languages and proceeded to define suitable language mappings to match the entities relevant to our analyses to the ones used in the ANTLR grammars. Note that we enabled only one parser per project; even though 𝕃ISA supports the analysis of multiple languages within the same computation, we wanted to observe its performance characteristics on a language-by-language basis, so that we can assess the impact of the used parser on overall performance.

We ensured the correctness of the formulated analyses (and thus the range compression algorithm) by creating a sample project for each language which contains a large number of code combinations varying across multiple revisions, manually calculating the expected code metrics and confirming that they match the results procured through 𝕃ISA. Then we ran the tool on the Git URLs of the 300 projects to compute the metrics for all revisions connected to the Git `HEAD` and persisted those metrics at the file, class and method level.

𝕃ISA's hardware requirements scale with the size of the project and the number of revisions (or rather the amount of actual change between revisions). Up to thousands of revisions and 100'000s of LOC, commodity hardware with 4-16GB of memory is sufficient. However, analyzing big projects, such as Mono, containing over 100k revisions and over 5M LOC, requires more memory. To accomodate the analysis of such projects, we used a Google Compute Engine instance with 24 cores and 158GB of memory to analyze all projects.

B. Evaluation of the Approach

The figures shown in table II indicates that the redundancy reduction techniques we apply are extremely effective. We briefly revisit the challenges outlined in section I-A:

Source data duplication. 𝕃ISA reads directly from bare Git. Even though this may seem a simple matter, to our knowledge, no existing *code* analysis tool takes advantage of this technique. Not only does it avoid checkouts, which would be costly both in terms of time and storage space, but it also enables 𝕃ISA to parse the source code of many releases in parallel, further speeding up the analysis.

TABLE II
AMOUNT OF CODE ANALYZED AND ANALYSIS DURATIONS IN THE STUDY.

		Java	C#	JavaScript
Projects		100	100	55
Revisions analyzed	total	646 261	489 764	204 301
	smallest	[1]715	[2]394	[3]234
	largest	[4]29 392	[5]106 160	[6]12 038
	median	2 956	2 051	1 097
Files parsed	total	3 235 852	3 234 178	507 612
	smallest	[7]946	[2]964	[8]10
	largest	[9]300 185	[10]312 229	[11]61 843
	median	11 171	10 685	2 198
Lines read	total	1 370 998 072	961 974 773	194 758 719
	smallest	[12]328 118	[13]55 971	[8]212
	largest	[9]194 875 095	[5]160 422 521	[6]18 194 261
	median	3 406 857	2 235 404	669 704
Runtime	parsing	13:25h	52:12h	29:09h
	analysis	1:35h	2:20h	1:00h
	total[A]	18:43h	57:20h	30:33h
	shortest[A]	[7]10s	[13]9s	[15]9s
	longest[A]	[9]2:44h	[5]8:43h	[16]5:56h
	average[A]	11:14min	34:24min	18:20min
	median[A]	2:15min	4:54min	3:43min
	total avg./rev.[B]	84ms	401ms	531ms
	median avg./rev.[B]	30ms	116ms	166ms

[A]) Including time spent cloning repositories and persisting results.
[B]) Excluding time spent cloning repositories and persisting results.
[1]javaassist [2]shadowsocks-windows [3]hain [4]cloudstack [5]mono [6]ember.js [7]commons-email [8]awesome-react [9]xtext [10]ravendb [11]babel [12]Android-Universal-Image-Loader [13]Fody [15]debug [16]d3

Redundant analysis of unchanged code. 𝕃ISA only reads the bare minimum necessary to still capture a complete representation for analysis. It's crucial to note that table II shows the number of files and lines *actually parsed* by 𝕃ISA, which are orders of magnitude *smaller* than the volume contained in all analyzed releases added together. To give a concrete example: analyzing **all 106 160 revisions** of the `mono` project, 𝕃ISA only had to read 160 422 521 lines of code. This is merely **40 times** the number of lines of code contained in the most recent revision alone (3 914 887 LOC). Likewise, the 2.5 billion lines of code 𝕃ISA parsed also represent only a fraction of the total code volume actually contained in all revisions of all projects.

Manual effort. Formulating the analyses (198 lines of Scala for the 11 metrics in our study) constitute the lion share in terms of manual effort, while writing the language mappings is a matter of minutes. One simply reads the original grammar and selects the right labels for different entities. To analyze a project, 𝕃ISA only requires the Git URL. Since it acts solely on source code, the analyst need not bother with libraries, build tools and other hindrances. Thus, 𝕃ISA allows for short turn-around times when doing exploratory research but also enables the analysis of large samples without excessive wait times.

Low-resolution historical data. Analyzing all 1 341 802 revisions, capturing the detailed history of 300 projects using 𝕃ISA took 4½ days. To put this into perspective, Tufano et al. report that in a recent study, the computation of metrics and code smells in 579 671 revisions of 200 projects required 8 weeks on a 28-core machine [66]. We argue that tools like 𝕃ISA can enable new kinds of studies, which are currently not feasible because of the tremendous effort involved in creating high-resolution historical code measurements, and we demonstrate such a study in the following section.

Isolation of programming ecosystems. We demonstrate that for the computation of basic metrics (which are known to correlate with more advanced metrics [10], [63]), light-weight entity mappings and the principle of avoiding explicit modeling of structure are valid approaches. The formulation of analyses in our study did not require language-specific instructions. However, the mapping of some entities may not be entirely obvious: for example, for JavaScript we mapped

[4] http://tiny.uzh.ch/C0

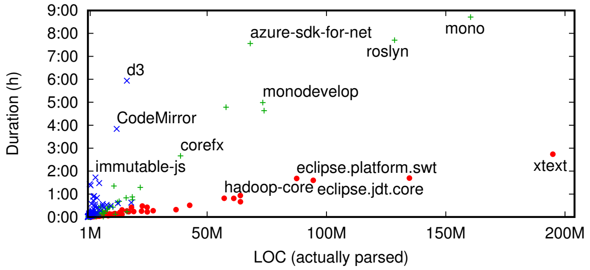

Fig. 4. Analysis duration vs. LOC for Java (●), C# (+) and JavaScript (×) projects.

the class concept to apply to root level AST vertices, because there is no convention on how to represent classes in JavaScript. However, we find light-weight mappings to work well in practice. The open source distribution of 𝕃ISA comes with additional mappings for Lua and Python 3, which were quickly created with little effort.

Performance compared to other tools. A fair one-on-one comparison to other tools is not feasible, as each tool has a different feature sets, restrictions and capabilities. In previous work [8], we compared the performance of a 𝕃ISA prototype, which lacked most of the performance-enhancing techniques discussed in section section II, to two existing analysis tools, namely inFusion [2] and SOFAS [27] for analyzing the AspectJ project. In that comparison, the prototype took 1:31min to analyze a single revision, outperforming SOFAS by a factor of 9.8 and inFusion by a factor of 4.3. The average time needed to analyze one revision fell below 2 seconds when analyzing more than 100 revisions and below 900ms when analyzing more than 1000 revisions, whereas using the other tools, each additional revision to be analyzed incurs the same cost.

We can now compare the performance of 𝕃ISA directly to the original prototype and by extension, to the tools used in the original study [8]. To analyze a single revision of AspectJ, the prototype spent 1:31min while 𝕃ISA spent only 37s. Of the total duration, the prototype spent 31s on parsing and building the code graph and 60s on the analysis, while 𝕃ISA spent 22s and 15s respectively. When analyzing thousands of AspectJ revisions, the prototype spent 650ms on average per revision, while 𝕃ISA spent only 34ms. Of this, the original prototype spent more than 500ms on parsing and graph building, and around 80ms on the analysis. 𝕃ISA on the other hand spent 28ms on parsing and 5.9ms on the analysis. The resulting average of 34ms per revision is just above the median average across all Java projects we analyzed, as shown at the bottom of in table II.

The parsing speed improvement can be attributed both to the filtered parsing, enabled by the light-weight mappings (section II-D), and to the asynchronous multi-revision parsing algorithm (section II-A). The original prototype stored the entire parse trees as provided by the parser and could only parse files for one revision at a time. The analysis speed improvement follows naturally from the filtered parsing, as the signals need to travel much shorter distances within the graph. This demonstrates how the chosen meta-model (or rather, lack thereof) has a significant impact on the overall performance.

Other observations. We observe that the majority of time is spent parsing and that it varies for different languages (72% for Java, 91% for C# and 95% for JavaScript). The same trend is visible in fig. 4. There are two reasons for these differences. First, the speed of an ANTLR-generated parser depends on a variety of factors. For example in JavaScript, terminating statements with a '`;`' is optional and the parser needs to perform *automatic semicolon insertion*, which can significantly slow down the parser. The second reason, which is specific to JavaScript, is the fact that files in JavaScript projects tend to be relatively large. Many libraries are simply developed and distributed in a single file. On top of this, many JavaScript projects include libraries, such as JQuery, AngularJS or d3.js, directly in the repository, because automatic dependency management as part of the build process (as done via Maven for Java or MSBuild for C#) is not widespread in the JavaScript ecosystem. These factors together mean that each time one of these files is updated, the entire file needs to parsed again, even if only a single statement was changed, impeding the effectiveness of 𝕃ISA's redundancy removal techniques.

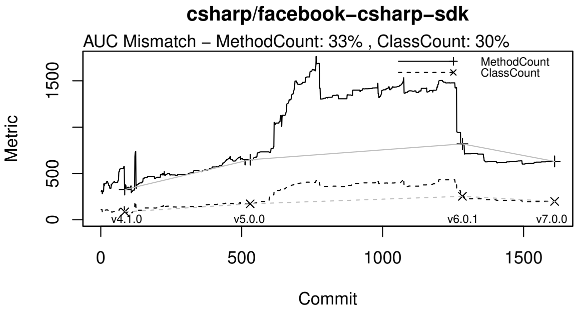

Fig. 5. Method Count and Class Count over the entire history of the facebook-csharp-sdk project.

C. *Sampling Revisions in SE Research*

An often recorded threat in existing code studies is the limited number of releases analyzed. While our tool can perform certain analyses for every commit, we hypothesize that there exists an optimal sampling interval which captures the evolution of a project with sufficient accuracy, such that slower, yet more feature-rich tools can be applied more effectively on a limited sample of revisions. However, to do so, we first need to identify this sampling interval.

Figure 5 shows the major releases of `facebook-csharp-sdk` as well as the actual, fine-grained data obtained upon analyzing all 1611 commits using 𝕃ISA. Even from the naked eye, it's clear that analyzing only major releases does not give a good approximation of the *overall* evolution of the project. Measuring the error between the assumed evolution based on sampling only major releases and the actual evolution in terms of the difference of the Area under the Curve (AUC) reveals a discrepancy upward of 30%.

To generalize this problem and to find an appropriate sampling interval, we apply the following procedure to each project. For 5 metrics from our data set (shown in fig. 6), we create two series: the first contains all, while the second contains s equally spaced commits from the entire history of the project, using an initial value of $s = 4$. We calculate the AUC of the high-resolution series AUC_{real} as well as the sparse series $AUC_{sampled}$ and calculate the absolute difference $AUC_{error} = |AUC_{real} - AUC_{sampled}|$. We quantify the mismatch as $e = AUC_{error}/AUC_{real}$. We then keep incrementing s by 1 until we find the s for which $e <= 0.05$. The sampling interval is therefore given by $i = c/s$ where c is the number of commits in the project. We repeat the experiment for targets $e <= 0.01$ and $e <= 0.1$ to see how sensitive the error is to different sampling intervals. The results for $e <= 0.05$ are visualized in fig. 6. We share the following insights:

- The appropriate sampling interval i varies for different programming languages. At $e <= 0.05$ the median i for Java projects is

near 600, while for C#, it is close to 400. For JavaScript, even lower i are necessary.
- For Java and C#, i is stable across all metrics, while for JavaScript, different metrics fluctuate with different intensity.
- i can vary significantly across projects. Sampling every 5000th commit can still be sufficient in some Java and C# projects, while other projects require an i of less than 50.
- With $e <= 0.01$, the median i drops significantly, to around 80 for Java, 40 for C# and below 5 for JavaScript. With $e <= 0.10$ it reaches ~800 for Java, ~500 for C# and ~200 for JavaScript.

Software evolution studies need to carefully consider the number of revisions analyzed in order to maintain validity. The decision depends on the language, but also on the acceptable error margin for a particular study. A safe recommendation, at least given a large enough number of projects in a study, is to analyze every 250th commit for Java, every 150th commit for C# and every 25th for JavaScript projects. Finally, even when using sampling, projects with thousands of commits still require a large number of commits to be analyzed for an accurate description of their evolution.

D. Threats to Validity

Threats to construct validity concern the relationship between theory and observation. We ensure the correctness of analyses performed by 𝕃ISA in a controlled environment by manually calculating metrics on exemplary multi-revision projects. This human judgement may be error-prone, but we tried to alleviate this issue by double-checking the results.

Threats to external validity concern the generalizability of our findings. One threat is the choice of projects used in our experimentation. To mitigate this issue we selected a large sample projects for each programming language and based our choice on their popularity according to multiple sources. Furthermore, all projects in our sample are open-source, and closed-source projects may generally undergo a different evolution. To extend the generality of our findings, we may replicate our study for additional projects and programming languages in future research.

IV. Limitations and Future Work

Attempting to increase the performance of software analysis tools is not trivial and inescapably requires a trade-off between comprehensiveness of analyses and the speed of execution. Any sort of code analysis can be formulated in 𝕃ISA, but considering it acts exclusively on source code (without access to a compiler), analyses which concern a single file (as opposed to cross-file analyses) are easiest to formulate. One could argue, that time used to reformulate analyses in 𝕃ISA could be better spent using language-specific tools. However, other tools cannot easily take advantage of the zero-redundancy approach to multi-revision analysis of 𝕃ISA. That said, the 𝕃ISA graph can hold arbitrary data, as outlined in section II-A, so it would also be possible to, for example, integrate a Java class loader to load multiple pre-compiled versions of a program into 𝕃ISA and perform much more complex analyses, such as coupling or control flow analyses, on the resulting multi-revision graph.

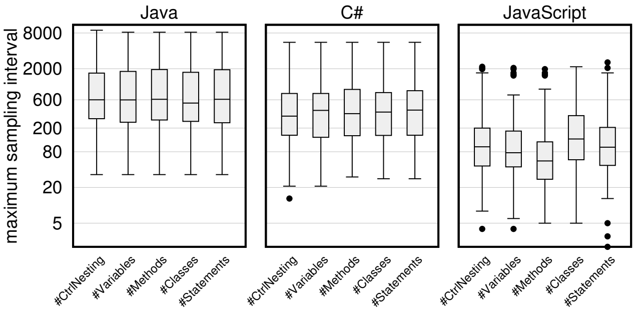

Fig. 6. Maximum sampling interval accurately capturing the evolution within a 5% margin of error based on 100 projects for each of 3 languages.

A technical limitation arises from how 𝕃ISA identifies vertices in the graph. When analyzing sources, the identifier of a vertex always corresponds to the file and syntax tree location of a node. Hence, renaming a file constitutes the creation of a new file, whose code will be represented using new vertices, even if nothing inside the file has changed. Likewise, if the order of methods in a class is changed, the subtrees will be partially re-created. This problem is hard to solve in general [36], [69], even more so in a language-independent manner, because an alternative identification scheme would be required, probably involving the actual names of entities within the file. But using the identifier of a syntax tree node alone would not suffice: e.g., in Java, multiple methods with the same name can exist if they have differing parameters and the same is true for many other entities as well. Note however that this limitation *only* concerns the redundancy reduction in 𝕃ISA. The actual analyses are robust with regard to these kinds of changes and their results are unaffected. If an analyst wishes to keep track of language-specific entities, she can simply formulate an analysis to connect all vertices belonging to the same program entity via additional edges.

It takes much more time to parse the source code than it takes to analyze it. It may be smart to find a way of persisting (and occasionally updating) the parsed graph such that when an analyst wants to run a computation, she does not need to re-parse everything. However, this has the downside that she must either decide on a rigid entity mapping or disable filtered parsing altogether, such that the entire syntax trees are represented in the graph (similarly to how BOA stores ASTs [20]), which in turn inflates memory requirements and computational load.

𝕃ISA only supports Git at the moment. Assuming it is deemed necessary to analyze projects using other version control systems, users have two viable options: they could use conversion tools such as `svn2git` or `git-cvsimport` and analyze the resulting Git repositories, or they could simply extend 𝕃ISA by implementinging the `SourceAgent` trait to extract files from other version control systems. The parallelized source access is implemented separately and can easily be re-used by injecting the `AsyncAgent` trait.

In our artifact study, we only analyzed files for one language per project, because we wanted to compare the impact of different parsers on 𝕃ISA's performance and because we wanted to give language-based recommendations regarding the sampling interval. However, any number of parser can be enabled in 𝕃ISA and it's naturally possible to analyze multiple languages within the same project or formulate analyses that involve multiple languages simultaneously.

V. Related Work

Scaling software analysis and the inclusion of multiple programming languages are ongoing efforts, but rarely are these two topics combined, like in the case of our research. Where the problem of covering multiple languages is explored, scale is not an issue and where large-scale analyses are performed, they are mostly restricted to a single language. In this section, we first related 𝕃ISA to existing code analysis tools and then discuss how 𝕃ISA's light-weight mappings differ from traditional meta-models.

A. Code analysis frameworks and multi-revision analysis.

Fischer et al. developed an approach combining revision- and bug tracking data into a common database, enabling simple queries for obtaining meaningful information on a software project and facilitating the anticipation of its future evolution [23]. Bevan et al. introduced a software analysis framework called Kenyon to facilitate software evolution research providing a flexible infrastructure addressing common logistical issues in software analysis (e.g., configuration retrieval, tool invocation etc.) [13]. Along similar lines, Gall et al. developed Evolizer, a platform for mining software archives, and ChangeDistiller, a change extraction tool that compares ASTs of two different versions, together enabling the retrospective analysis of a software system's evolution [24], [25], improving upon previous algorithms for extracting changes [16]. Ghezzi et al. presented a service-oriented framework called SOFAS, enabling collaborative analyses of software projects and facilitating the replication of mining studies [27]. Successively, Ghezzi et al. analyzed MSR (Mining Software Repositories) studies published between 2004 and 2011 and found that a minority (25 out of 88) are fully replicable, indicating that replicability and generalizability remain important issues in software engineering research [28]. Zimmermann et al., propose a tool called ROSE that, given the full history of a project, predicts the location of most likely further changes [71]. To enable the large-scale analysis of repository metadata, Dyer et al. have developed Boa, an infrastructure, backed by a Hadoop cluster, for performing large-scale studies via a web interface, IDE plugin or web API [20], [21]. It contains the commit data of over 8 million projects which can be analyzed using a domain specific language. More recently it provides access to the ASTs of Java code contained in all releases of numerous projects, which can be analyzed using visitor patterns. Compared to 𝕃ISA, which is a stand-alone library, BOA is also a server infrastructure and uses a concrete, fixed model for Java only.

A technique similar to the merging of ASTs of multiple versions is presented by Le et al., but for patch verification across control flow graphs (CFGs) [43]. Contrary to Le's work, our approach represents ASTs (not CFGs) and it scales to hundreds of thousands of revisions (compared to a few dozen for MVICFGs), as the two approaches have very different goals. Another example for a similar technique can be found in TypeChef [33] which uses a variability-aware lexer to analyze multiple *configurations* of C programs (i.e., `#ifdefs`) in a shared graph. Compared to 𝕃ISA, TypeChef targets type checking and other architecture aspects in a multi-configuration context rather than code metrics in a multi-revision context.

B. Multi-language analysis.

The biggest difference comparing our light-weight mappings to existing meta-models is the fact that the latter always imply a transformation of a source data structure (e.g., an AST) to a concrete instance of the meta-model. In 𝕃ISA however, the mappings themselves only influence the *types of source nodes* which are loaded into the graph. The structure of the graph, however, will be identical to the original structure provided by the parser, minus any nodes which are not mapped. It is only in the context of a particular analysis that the mappings of individual nodes gain meaning. As such, our light-weight mappings can be considered a *view* onto an existing graph structure, rather than a meta-model. Furthermore, code models typically come with predefined entity types and relationships, whereas our light-weight mappings are formulated in the context of a particular analysis.

A well known source code meta-model is FAMIX, originally proposed by Tichelaar et al. to aid in the refactoring of source code of different languages [65]. FAMIX defines concepts such as classes, methods, attributes, invocations and inheritance. Source code in a given language is transformed into a concrete FAMIX instance, which can be used to perform analyses or give refactoring advice. The authors note that any code meta-model represents a trade-off between being too *coarse-grained* to be useful for a wide range of problems and being to *fine-grained* to remain sufficiently language-independent. FAMIX has since been used as a meta-model by other tools [27], [41], as well. Strein et al. have developed a meta-model for capturing multi-language relationships in source code [61] not dissimilar to FAMIX, but with the added idea of enabling *cross*-language refactorings, for example renaming variables both in the front- and back-end of a multi-language project. Another example is Rakić et al.'s framework for language-independent software analysis [58]. Using an ANTLR parser, it transforms the source code of different languages into a so-called enriched Concrete Syntax Tree (eCST), which is stored as XML, and then re-read to calculate basic code metrics. The eCST is more fine-grained than a FAMIX model. Heavy-weight models similar to FAMIX, such as KDM [4] or ASTM [3] as well as general-purpose models such as RSF [34] or GXL [68] model not only of the kinds of nodes, but also their relationships and structure explicitely. The same is true for M^3 [32], which is specifically designed for use with Rascal Metaprogramming Language [29] and which also includes non-code concepts such as physical and logical source locations.

All meta-models we describe here have only been applied in single-revision, single-project settings and, contrary to our light-weight approach, do not aim to be practical in large-scale analyses, where performance is crucial.

VI. Conclusion

We present several distinct redundancy removal techniques and demonstrate that in combination, they enable the comparatively rapid analysis of code contained 100 000s of commits. After formulating a particular analysis, the selection of projects, the creation of language mappings and the automated execution of analyses in our open-source tool, 𝕃ISA, are straightforward and enable the quick extraction of fine-grained software evolution data from existing source code. We also present the idea of using light-weight mappings instead of traditional meta-models for static, structural analyses of code written in different languages. The light-weight mappings not only represent a simple, analysis-specific bridge between different languages, but they also play an important role in improving 𝕃ISA's performance, as they enable the filtering of unnecessary source data without sacrificing knowledge relevant to analyses. Our evaluation on a large sample of projects allowed us to observe that code in different programming languages evolves differently and that an accurate representation of their evolution is possible only with a large enough number of sampled commits.

𝕃ISA fills a unique niche in the landscape of software analysis tools, occupying the space between language-specific tooling used for the in-depth analysis of individual projects and releases, and traditional software repository mining, where code analysis is typically restricted to merely counting files and lines of code. The techniques discussed in this paper could be adapted for existing solutions individually, but 𝕃ISA also offers clean and easy-to-implement interfaces for additional sources (i.e. version control systems), parsers and storage methods.

References

[1] Gathering statistics on javascript projects since 2015. http://stats.js.org/. Accessed: 2016-03-20.

[2] infusion by intooitus s.r.l. http://www.intooitus.com/products/infusion. Accessed: 2014-03-30.

[3] Omg: Astm. http://www.omg.org/spec/ASTM/1.0/. Accessed: 2016-10-06.

[4] Omg: Kdm. http://www.omg.org/spec/KDM/1.3/. Accessed: 2016-10-06.

[5] We analyzed 30,000 github projects – here are the top 100 libraries in java, js and ruby. http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/. Accessed: 2016-03-20.

[6] *Supporting and Accelerating Reproducible Research in Software Maintenance Using TraceLab Component Library*. IEEE Computer Society, 2013.

[7] K. Agrawal, S. Amreen, and A. Mockus. Commit quality in five high performance computing projects. In *Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science*, SE4HPCS '15, pages 24–29, 2015.

[8] C. V. Alexandru and H. C. Gall. Rapid multi-purpose, multi-commit code analysis. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 635–638, May 2015.

[9] T. Arbuckle. Measuring multi-language software evolution: A case study. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 91–95, 2011.

[10] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, Oct. 1996.

[11] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: an evolutionary study. pages 1–43.

[12] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *2013 IEEE International Conference on Software Maintenance*, pages 280–289, 2013.

[13] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. pages 177–186, 2005.

[14] B. D. Bois, P. V. Gorp, A. Amsel, N. V. Eetvelde, H. Stenten, and S. Demeyer. A discussion of refactoring in research and practice. Technical report, 2004.

[15] J. Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 111–119, 2009.

[16] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 493–504, 1996.

[17] D. Chen, G. Doumeingts, and F. Vernadat. Architectures for enterprise integration and interoperability: Past, present and future. *Comput. Ind.*, 59(7):647–659, Sept. 2008.

[18] M. D'Ambros, H. C. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. 2008.

[19] C. V. C. de Magalhães, F. Q. B. da Silva, and R. E. S. Santos. Investigations about replication of empirical studies in software engineering: Preliminary findings from a mapping study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 37:1–37:10, 2014.

[20] R. Dyer. *Bringing Ultra-large-scale Software Repository Mining to the Masses with Boa*. PhD thesis, Ames, IA, USA, 2013. AAI3610634.

[21] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. pages 23–32, 2013.

[22] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5), 2012.

[23] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, 2003.

[24] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[25] H. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *Software, IEEE*, 26(1):26–33, 2009.

[26] H. C. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *1997 International Conference on Software Maintenance (ICSM '97), Proceedings*, page 160, 1997.

[27] G. Ghezzi and H. Gall. Sofas: A lightweight architecture for software analysis as a service. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 93–102, June 2011.

[28] G. Ghezzi and H. Gall. Replicating mining studies with sofas. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 363–372, 2013.

[29] M. Hills, P. Klint, and J. J. Vinju. *Program Analysis Scenarios in Rascal*, pages 10–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[30] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 99–108, 2008.

[31] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011*, pages 384–387, 2011.

[32] A. Izmaylova, P. Klint, A. Shahi, and J. J. Vinju. M3: an open model for measuring code artifacts. *CoRR*, abs/1312.1188, 2013.

[33] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[34] H. M. Kienle and H. A. Müller. Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247 – 263, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[35] M. Kim, J. Nam, J. Yeon, S. Choi, and S. Kim. Remi: Defect prediction for efficient api testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, page To Apper. ACM, 2010.

[36] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 58–64, New York, NY, USA, 2006. ACM.

[37] S. Kim and M. D. Ernst. Which warnings should i fix first? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 45–54. ACM, 2007.

[38] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45. ACM, 2006.

[39] E. Kocaguneli, T. Menzies, and J. Keung. On the value of ensemble effort estimation. *Software Engineering, IEEE Transactions on*, 38(6):1403–1416, 2012.

[40] K. Kontogiannis, P. K. Linos, and K. Wong. Comprehension and maintenance of large-scale multi-language software applications. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 497–500, 2006.

[41] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler - an information visualization tool for program comprehension. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 672–673, May 2005.

[42] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[43] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1047–1058, New York, NY, USA, 2014. ACM.

[44] E. Ligu, T. Chaikalis, and A. Chatzigeorgiou. Buco reporter: Mining software and bug repositories. In *BCI (Local)*, CEUR Workshop Proceedings, page 121. CEUR-WS.org, 2013.

[45] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359, 2004.

[46] T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[47] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of java build systems. *Empirical Software Engineering*, 17(4):578–608, 2011.

[48] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering*, 20(6):1587–1633, 2014.

[49] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10. ACM, 2009.

[50] T. Mens. Introduction and roadmap: History and challenges of software evolution. In *Software Evolution*, pages 1–11. Springer Berlin Heidelberg, 2008.

[51] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik. Studying evolving software ecosystems based on ecological models. In *Evolving Software Systems*, pages 297–326. Springer Berlin Heidelberg, 2014.

[52] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer-Verlag, 2008.

[53] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.

[54] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22, 2005.

[55] M. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 15–15, 2005.

[56] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461. ACM, 2006.

[57] J. Oosterman, W. Irwin, and N. Churcher. Evojava: A tool for measuring evolving software. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113*, ACSC '11, pages 117–126. Australian Computer Society, Inc., 2011.

[58] G. Rakić, Z. Budimac, and M. Savić. Language independent framework for static code analysis. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 236–243, New York, NY, USA, 2013. ACM.

[59] W. Shang, B. Adams, and A. E. Hassan. An experience report on scaling tools for mining software repositories using mapreduce. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 275–284. ACM, 2010.

[60] W. Shang, Z. M. Jiang, B. Adams, and A. Hassan. Mapreduce as a general framework to support research in mining software repositories (msr). In *6th IEEE International Working Conference on Mining Software Repositories, 2009.*, pages 21–30, 2009.

[61] D. Strein, H. Kratz, and W. Lowe. Cross-language program analysis and refactoring. In *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, pages 207–216, Sept 2006.

[62] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: Graph algorithms for the (semantic) web. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 764–780. Springer-Verlag, 2010.

[63] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, Apr. 2003.

[64] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy. A case study of refactoring large-scale industrial systems to efficiently improve source code quality. In *Computational Science and Its Applications – ICCSA 2014*, volume 8583 of *Lecture Notes in Computer Science*, pages 524–540. Springer, 2014.

[65] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164, 2000.

[66] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 403–414, May 2015.

[67] M. VanHilst, S. Huang, J. Mulcahy, W. Ballantyne, E. Suarez-Rivero, and D. Harwood. Measuring effort in a corporate repository. In *IRI*, pages 246–252. IEEE Systems, Man, and Cybernetics Society, 2011.

[68] A. Winter, B. Kullbach, and V. Riediger. An overview of the gxl graph exchange language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, UK, 2002. Springer-Verlag.

[69] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Methodol.*, 1(3):310–354, July 1992.

[70] L. Zhang, Y. Zou, and B. Xie. A scalable crawler framework for floss data. In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, Internetware '13, pages 8:1–8:7. ACM, 2013.

[71] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.