Empirical Software Engineering *(preprint)*

Redundancy-free Analysis of Multi-revision Software Artifacts

Carol V. Alexandru · Sebastiano Panichella · Sebastian Proksch · Harald C. Gall

30. April 2018

Abstract Researchers often analyze several revisions of a software project to obtain historical data about its evolution. For example, they statically analyze the source code and monitor the evolution of certain metrics over multiple revisions. The time and resource requirements for running these analyses often make it necessary to limit the number of analyzed revisions, e.g., by only selecting major revisions or by using a coarse-grained sampling strategy, which could remove significant details of the evolution. Most existing analysis techniques are not designed for the analysis of multi-revision artifacts and they treat each revision individually. However, the actual difference between two subsequent revisions is typically very small. Thus, tools tailored for the analysis of multiple revisions should only analyze these differences, thereby preventing re-computation and storage of redundant data, improving scalability and enabling the study of a larger number of revisions. In this work, we propose the *Lean Language-Independent Software Analyzer* (𝕃ISA), a generic framework for representing and analyzing multi-revisioned software artifacts. It employs a redundancy-free, multi-revision representation for artifacts and avoids re-computation by only analyzing changed artifact fragments across thousands of revisions. The evaluation of our approach consists of measuring the effect of each individual technique incorporated, an in-depth study of 𝕃ISA's resource requirements and a large-scale analysis over 7 million program revisions of 4,000 software projects written in four languages. We show that the time and space requirements for multi-revision analyses can be reduced by multiple orders of magnitude, when compared to traditional, sequential approaches.

Software Evolution and Architecture Lab - s.e.a.l.
Binzmühlestrasse 14
CH-8050 Zürich
Switzerland
E-mail: {alexandru,panichella,proksch,gall}@ifi.uzh.ch

1 Introduction

Static analysis is crucial for modern software development. It is omnipresent both in software engineering research and practice and has a broad range of applications. For example, it can be used to enforce coding guidelines [70], allocate resources and estimate future effort [29,49,59,85], suggest refactoring opportunities [20, 29, 62, 80], detect bugs [46, 66] and duplicates [15, 42], or assert quality properties [25]. Over the years, many approaches went beyond pure static code analysis and also leverage historical information related to the development of open-source projects. This information not only represents an interesting source for building tools, e.g., to discover new patterns and anti-patterns [57,64,73,84] or for change and bug prediction [29,46,48,66,67], but is also the body of a whole line of research that is interested in understanding the evolution of software systems.

Software evolution research deals with questions concerning the development and change occurring within a software project. Naturally, this research relies on *historical data* about structured and semi-structured artifacts. Thus, researchers have created a wide spectrum of approaches to extract pertinent information from software repositories that facilitate studies on software development and software evolution [12, 24, 31, 33, 60, 69]. They have also performed historical analyses of popular software projects, such as the Firefox web browser [18, 91] and projects coming from open source software ecosystems (e.g., the Apache Software Foundation) [13, 19, 39, 61, 87].

Due to the lack of tools that are able to efficiently analyze many revisions at once, researchers usually fall back to regular static analysis tools for extracting information and monitoring properties of software systems over time. These tools often can only be executed on each individual revision [12] which makes this approach hard do adopt when the goal is to study large projects. Indeed, large projects typically feature a vast number of revisions and analyzing all of them individually is computationally expensive. To make these studies feasible, researchers typically analyze a limited number of revisions by either focusing on release revisions or through sampling a subset of the available revisions.

However, such coarse-grained representations threaten the representativeness of the results [65], because details might be lost in the process. A concrete example from the area of testing is the detection of test smells [77, 90]. The state-of-the-art tools DECOR [63] and JDeodorant [83] cannot analyze multiple revisions of test artifacts. For instance, JDeodorant requires compiling source code and importing each project into an Eclipse[1] workspace. As a result, empirical studies on test smells are typically limited to a specific set of major releases from selected software projects [14, 90].

Some researchers have studied the repetitiveness [68] or uniqueness [76] of changes. Others have sketched tools to identify and remove redundancies between program revisions in an analysis. For example, Kawrykow et al. propose a tool for detecting non-essential code differences [44] and Yu et al. [89]

[1] http://www.eclipse.org/

propose a technique to reduce changes to those relevant for a particular development task. However, no concrete solution directly addresses the problem of redundancy in software evolution analysis itself.

A second problem involving redundancy is the analysis of projects consisting of different programming languages. Not all, but many software analysis tools are limited to one specific language (e.g., linters, bug detection tools and metric computers) and researchers analyzing multi-language projects or multiple projects written in different languages need to apply multiple tools that sometimes diverge in how exactly they compute and report measurements or they need to implement new tools from scratch. This is the case even though many concepts (like methods, attributes and cyclomatic complexity) are very similar in different programming languages. While multi-language program analysis has been explored in recent literature [11,50], working approaches are usually based on a concrete metamodel [26,78,82] for the analysis or manipulation of multi-language software code. However, these models are comparatively heavy-weight and always require a translation from an existing program representation (such as an AST) to an instance of the model. Furthermore, they are not designed with performance as a primary goal in mind.

Given these circumstances, we base our work on two observations: *individual revisions of a software system typically only have minor differences* and *analyses of different programming languages exhibit significant overlap.* We argue that avoiding redundancies in the representation and subsequent analysis of multi-versioned, multi-language artifacts can facilitate studying a larger number of more diverse projects and investigating a more fine-grained history involving a substantially higher number of revisions. Thus, this research can be condensed into two main research problems:

$\mathbf{P}_1$: *Redundant analysis must be minimized to effectively perform large-scale software evolution studies on full program histories.*

$\mathbf{P}_2$: *Analyzing projects involving more than one language on a larger scale requires multi-language abstractions to be both expressive* ***and*** *scalable.*

As a solution to both problems, we present the *Lean Language-Independent Software Analyzer* ($\mathbb{L}$ISA), a multi-revision graph database and computational framework for analyzing software artifacts across thousands of revisions with minimum redundancy. Even though P_1 and P_2 are two distinct problems, a common solution for both required us to consider certain trade-offs. For example, domain knowledge can be used to optimize the performance of multi-revision analyses for one specific programming language, but the solution would no longer be generic and language independent. Likewise, generic metamodels that support several languages tend to be heavyweight, whereas the high-speed requirement forces us to find a more lightweight solution. Ultimately, we believe that a generic solution is of greater value for the scientific community and we have designed $\mathbb{L}$ISA as a generic approach for representing and analyzing structured artifacts that exist in countless different revisions. While this paper is focused on one particular use case, namely the static analysis of plain-text source code and its parsed abstract syntax trees (ASTs),

the core framework is agnostic to the type of multi-revision graph that is the subject of an investigation.

Our experiments show that the techniques we present in this paper, which could be individually applied to existing tools, are highly effective. When combined, they reduce the computational effort and resource requirements to around 4% of what would be required for an analysis of individual revisions (depending on the programming language). We also show that our light-weight approach of applying generic analyses to different programming languages is very easy to use and implement. These achievements enabled us to analyze the complete source code in every revision of 4,000 projects written in 4 different programming languages.

In summary, this paper presents the following main contributions:

- We conceive several techniques to remove redundancies in multi-revision software analysis and implement a framework, 𝕃ISA, for loading, representing and concurrently analyzing hundreds of thousands of program revisions at minimal marginal cost for each additional revision.
- We introduce *lightweight entity mappings*, a novel approach enabling ad-hoc mappings between artifact-specific entities and cross-artifact semantic concepts without an actual translation to a pre-defined metamodel. These mappings identify the parts that need to be observed over the entire history of a software artifact and play a crucial role in the redundancy reduction.

This paper is an extended version of our previous work [9]. Compared to the previous paper, it includes the following original contributions:

- We significantly extend the size of the artifact study by an order of magnitude and analyze 4,000 projects written in 4 different languages.
- We integrate Python into the framework to demonstrate 𝕃ISA's extensibility and perform a Python-specific artifact study. We further demonstrate this extensibility in a detailed description of all required steps to add support for the Lua programming language.
- We augment the description of our approach with a running example, directly measuring the effectiveness of each technique we describe.
- We perform an in-depth analysis of the resource requirements and scalability of 𝕃ISA given different amounts of memory and CPU cores.
- We further discuss current limitations of 𝕃ISA as well as the potential future uses for other kinds of software evolution research.

We provide a comprehensive replication package[2] together with this work. It contains all scripts and runners that we have used in our experiments, the 1.6 GB of fine-grained code measurements for the 4,000 projects analyzed in our study, and the required scripts to obtain and interpret the original source code. Furthermore, all source code, including alternate versions and analyses of our evaluation, is available as open source in the project repository.[3]

[2] `https://doi.org/10.5281/zenodo.1211549`

[3] `https://bitbucket.org/sealuzh/lisa`

The rest of this paper is structured as follows. In Section 2, we present the different techniques to reduce redundancies in multi-revision and multi-language research implemented in 𝕃ISA, as well as the limitations of our approach. Section 3 provides an in-depth evaluation regarding the performance, resource requirements and adaptability of 𝕃ISA. In Section 4, we discuss our experience with 𝕃ISA in performing practical software evolution studies and the insights we gained from our experiments. Furthermore, we elaborate on future work, both in terms of how to improve 𝕃ISA and how it may be used in future research scenarios. We re-visit related work on analyzing software evolution in Section 5 and relate it to our own contribution. We conclude the paper in Section 6.

2 Creating a Lean Language-Independent Software Analyzer

In the following sections, we describe several techniques which significantly reduce the redundancies inherent to analyzing multiple revisions of a program. The combined workflow and architecture diagram shown in Fig. 1 summarizes the following components. We present a multi-revision graph representation of source code which vastly reduces both the memory required to represent source code of multiple program revisions and the computational effort required for its analysis (①, Section 2.1.3). By operating directly on bare Git repositories (②) and by parsing source code asynchronously across multiple revisions (③, Section 2.1.2), available computing power is utilized more efficiently, and by sharing state across multiple revisions at the AST node level, redundant storage of data is avoided entirely (④, Section 2.1.5). Using an asynchronous computational paradigm (⑤, Section 2.2), one or more analysis suites (⑥, Section 2.2.1), can be executed in parallel without conflicts or blocking waits. Flexible mappings between parser-specific node types and

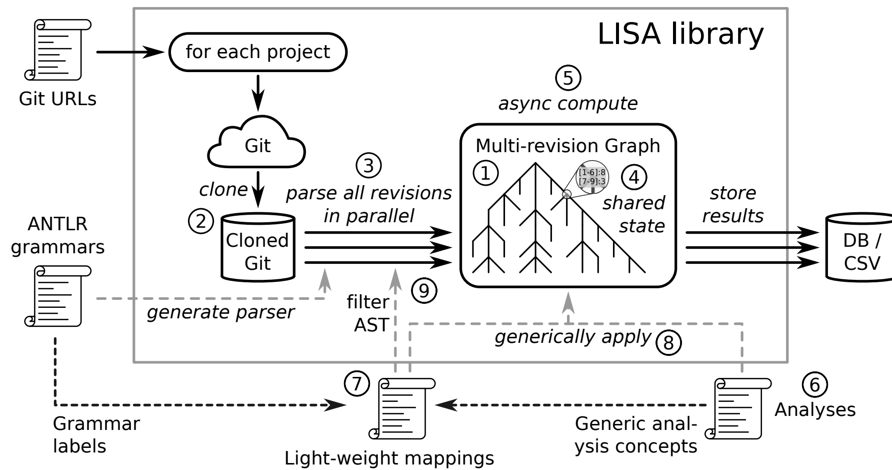


Fig. 1: Workflow and architecture overview.

analysis-specific keywords (⑦, Section 2.3) not only enable the analysis of multiple programming languages with the same analysis instructions (⑧), but also avoid representing nodes which are unnecessary for a particular analysis (⑨), further reducing the workload of any specific analysis. All the techniques described in this paper could be applied independently to improve different aspects of existing applications. We combined them in our research tool, 𝕃ISA.

Running example. While Section 3 contains an in-depth evaluation, we provide a running example throughout the approach to illustrate the immediate effect of each technique on the analysis of multiple revisions. We do this by sharing runtime metrics and by comparing 𝕃ISA to traditional approaches and handicapped versions of itself, where single features are disabled selectively for the same analysis. This allows us to make statements on the effectiveness of individual techniques, all else being equal. We chose the *Reddit* project as an example: Reddit is a social media site currently ranking 8th on Alexa [5], a renowned web traffic analysis index. Reddit is primarily written in Python and JavaScript and the source code powering Reddit is available on GitHub. The latest revision contains roughly 70,000 lines of Python- and 50,000 lines of JavaScript code. We chose Reddit because it is an industrial-scale product, is composed of more than one language, is large enough for a representative discussion, but small enough such that handicapped versions our tool can still produce results within reasonable time.

We enable both the Python and JavaScript parsers in 𝕃ISA and apply the two analysis suites used in the two artifact studies described in Section 4 simultaneously: the object-oriented analysis suite computes complexity metrics and counts entities such as classes, methods and variables, while the Pythonic analysis suite detects Pythonic idioms [71]. Throughout the approach, we will highlight information relating to the running example as follows:

Example 1: If one were to analyze each of the 7,947 revisions of the *Reddit* project independently, for example by checking out the source code and applying an analysis tool to the working directory, a total of 2,195,805 files and 595,232,615 lines of code would need to be checked out, parsed and analyzed. A *handicapped* version of 𝕃ISA which follows this exact procedure, needs 6.9 seconds on average to check out sources, compute code metrics and detect Pythonic idioms for a single revision of Reddit (on a server with 64 GB memory and 12 CPU cores, including checkouts and data persistence). Over all revisions, this quickly adds up to a total runtime of 15 hours and 9 minutes. Using the proper, fully optimized version of 𝕃ISA, the same analysis completes in 15 minutes and 43 seconds: 5 seconds to clone the project from GitHub, 2 seconds to extract all commit metadata and build an incremental list of the 14,495 files – spanning across all revisions – that actually *need* to be parsed, 12 minutes to parse them, 4 minutes for the actual analysis and 11 seconds to store the results. That is 119ms on average per revision, or 8.4 revisions per second.

Listing 1: Boilerplate for using a new ANTLRv4 grammar with 𝕃ISA

```
1 object AntlrCSharpParser
2 extends AntlrParser[CSharp4Parser](AntlrCSharpParseTree) {
3   override val suffixes = List(".cs")
4   override def lex(input: ANTLRInputStream) = new CSharp4Lexer(input)
5   override def parse(ts: CommonTokenStream) = new CSharp4Parser(ts)
6   override def enter(parser: CSharp4Parser) = parser.compilation_unit()
7 }
```

2.1 Multi-revision graph representation of source code

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

2.1.1 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. This graph is used to model arbitrary representations of program artifacts. For example, 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 and store their literal token as metadata, while edges store no metadata. The syntax tree path of a node could consist of its type and an index to discern siblings, e.g., `/src/Main.java/ComponUnit0/Class0/Method4` for the 5th method of a class contained in a file `Main.java`. How vertices are identified is up to the parser which adds the vertices to the graph. 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 modifier. A graph is also suitable for modelling 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. In this case, vertices could be identified by their proper names, as long as the identifiers are unique. For example, a Java method with the signature `String getName(int id)` could be identified by `org.example.core.Main/String.getName(int)`. The metadata can also serve as a container for additional analysis data, e.g., 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. 𝕃ISA uses Signal/Collect [79], a low-level graph framework, to store the graph and exposes an interface with just two mandatory

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 sort of source data can be used in conjunction with 𝕃ISA, as long as it can be represented as a graph. 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* Java parser, the JDK *Nashorn* JavaScript parser, the native *cpython* Python parser and for ANTLRv4. ANTLR is a parser generator for which existing grammars can be found for many languages.[4] Given an ANTLR grammar, it is possible to create a new 𝕃ISA-compatible parser with only a few lines of boilerplate code, as shown in Listing 1.

2.1.2 Asynchronous metadata extraction and source code parsing

Traditional tools typically check out individual revisions from the version control system (VCS) to act on source code contained in files and folders. Using Git, for example, this means first reading all files contained in a specific revision from the Git database and writing them to a working directory, before actually parsing them and then moving on to the next revision and repeating this process for every single revision. It also means that for any specific revision, even when parsing files in parallel, it would always be necessary to wait for the largest files to finish parsing before being able to move on to the next revision.

Our implementation separates this process into two separate stages, both of which can be parallelized efficiently: 1. identifying all relevant files to be parsed in the entire history of the project, and 2. actually parsing the files. Whereas centralized version control systems like SVN or CVS rely on a server to checkout files for specific revisions into local working directories, Git is a decentralized solution where a local file-based database contains the data of all revisions. This local Git repository is called *bare* if no working copy has been checked out. The Git database is a content-addressable file system, consisting of a key-value store (to uniquely identify objects such as file blobs) and a tree structure. Using the tree, the identifiers of both commits and blobs contained therein can be resolved. It is thus possible to access such blobs directly via their ID [22]. To use 𝕃ISA with Git, we implemented 𝕃ISA's `SourceAgent` interface, which mandates the preliminary creation of key-value pairs (p, B) for each file in any revision of the project, where p uniquely identifies a file by its path and B denotes a chronologically ordered sequence of sources (i.e., Git blob ids) for the file in different revisions. This process can conveniently be parallelized: 𝕃ISA traverses the Git graph database and uses a pool of workers which act on individual revisions and extract those blobs whose identifier matches the

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

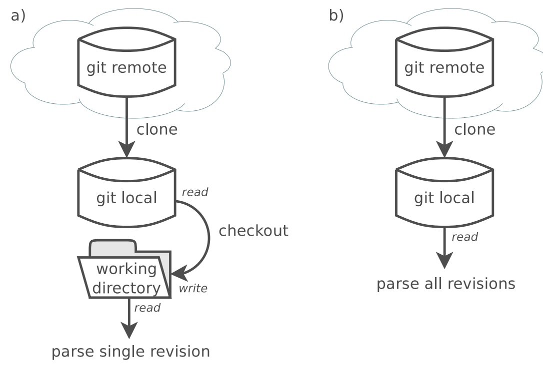


Fig. 2: Contrary to the traditional method of checking out (i.e., reading and writing) *all* files in each revision and then parsing the source code from the file system (a), parsing the source code directly from a bare Git repository requires only a single read operation per *relevant* file (b).

suffix of any enabled parser. Using this technique, files of different revisions are extracted simultaneously and without the need to ever check out a working copy of the code, as illustrated in Fig. 2. Then, in the second step, each pair can be assigned to one of many parallel workers, which parse the sources for a uniquely identified file sequentially, while sources for different files can be read independently from each other, even if they belong to different revisions.

The approach has *five* distinct performance benefits over the traditional approach: 1. 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. 2. It avoids the unnecessary checkout of any other assets (e.g., images) contained in the project, which are not even relevant for the analysis. 3. 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. 4. Files are only read in revisions in which they were actually changed. 5. It avoids having to wait on the parsing of large files in every single revision. Instead, 𝕃ISA can efficiently parallelize the parsing of many different files from different revisions.

This technique, requiring direct access to data blobs of different commits, is easily applied to decentralized version control systems like Git and Mercurial. For centralized VCS like SVN, CSV or TFS, which, in principle, support checking out single files of specific revisions, a low latency network and a fast server would be required, although converting such repositories to Git would likely be the better option.

Example 2: The parallelized metadata and file blob identification for the *Reddit* project took just 1.63 seconds using 𝕃ISA. For comparison, simply

checking out all 7,947 revisions of the project by looping over `git log` and using `git checkout` takes 9 minutes and 40 seconds on the same hardware. This is a fair comparison because in both cases, actually parsing the source code is not included. This shows, unsurprisingly, yet conclusively, that directly accessing Git eliminates the time spent on checking out source code, which can be significant even for medium-sized projects such as *Reddit*. To exactly identify the difference between asynchronously parsing files of all revisions compared to parsing them from checked-out files one revision at a time, we tracked the time spent parsing (excluding any other activity) of the properly optimized version of 𝕃ISA and a handicapped version which uses checkouts, yet also parses files of a single revision using parallel workers. Measuring only the actual time spent parsing files into the graph, the asynchronous version spends 11 minutes and 38 seconds, while the handicapped version needs 37 minutes and 5 seconds. This illustrates the significant slowdown caused by the explicit checkouts and having to wait for the largest files to finish parsing in each revision.

2.1.3 Multi-revision graph representation.

A single commit often changes only a tiny part of any file. This means that the graph representations of a file in 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 an edge exists in a revision is determined by whether or not the two connecting vertices exist. In this paper, we will generally denote ranges as pairs of dash-separated numbers in brackets. For instance, a node that exists in revisions 10 to 15 and 19 to 20 is denoted as {10-15,19-20}. 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.

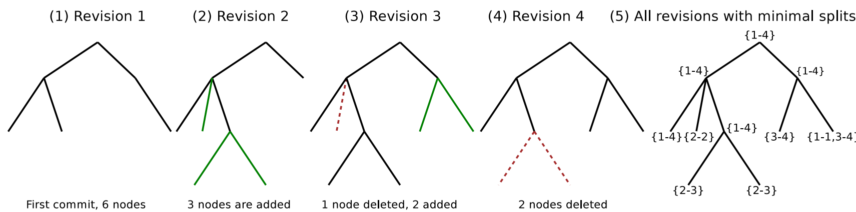


Fig. 3: Exemplary ASTs of four revisions and their shared representation, where revision ranges indicate which revisions each node exists in.

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. Figure 3 shows an example of four revisions of a tiny, exemplary AST. Individually, these revisions contain 30 AST nodes. However, the shared representation (5) contains only 10 nodes. 5 of them exist in all revisions ({1-4}), while the others exist in one or more revisions, but not all of them. In total, 11 ranges are necessary to capture all 30 vertices contained in the four original revisions without any information loss. Thus, we can calculate a *vertex compression ratio* of $11/30 = 0.36$.

Example 3: All 7,947 revisions of the *Reddit* project contain a total of 2,195,805 files. One would need 1,003,984,942 vertices to represent the ASTs contained in these files individually. Using the multi-revision graph representation, only 2,014,720 vertices, together containing 2,991,074 ranges, are required. Thus, the *vertex compression ratio* using $\mathbb{L}$ISA to analyze *Reddit* is $2{,}991{,}074/1{,}003{,}984{,}942 = 0.0030$ after parsing. In other words, the multi-revision representation only needs about 0.3% of the vertices required to represent all revisions individually.

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" AST node, 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.
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 was not changed in a revision, then, as per the previous section, it will not be parsed. This leads to erroneous gaps in the revision ranges of a vertex. For example, if a file is changed 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 code within that file is actually present in-between these two revisions, necessitating a range extension for a final configuration of {5-9,10-10}. If a vertex contains the exact same metadata in two subsequent revision ranges, it is further possible to merge them into a single range, e.g., {5-10}.
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 and it would be redundant to store the unchanged name and complexity twice.

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.*

2.1.4 Sparse graph loading algorithm.

We explain the 𝕃ISA range compression, 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 1. The vertex 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 1 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. I.e., in revisions with the same letter the metadata is identical. In this example, the node has the same metadata in revisions 0 to 3 ('A'), then the node is deleted in revision 4, then it reappears

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 (Assignment: ←, Reference: →).

```
 1: procedure updateVertex(v, rev, meta)
 2:     createRange(v.ranges, rev, meta)
 3:     queue(compressRanges(v.ranges, v.mark, rev − 1))
 4: procedure createRange(ranges, rev, meta)
 5:     new ← ({rev, rev} → meta)
 6:     ranges ← ranges + new
 7: procedure compressRanges(ranges, mark, end)
 8:     if end = null then                                          ▷ Case 1
 9:         mark ← ranges.head
10:     else
11:         next ← ranges.get(end + 1)
12:         if mark = null & next = null then                       ▷ Case 2
13:             doNothing
14:         else if mark = null & next ≠ null then                  ▷ Case 3
15:             mark ← next
16:         else if mark ≠ null & next = null then                  ▷ Case 4
17:             extended ← ({mark.start, end} → mark.meta)
18:             ranges ← ranges − mark + extended
19:             mark ← null
20:         else if mark ≠ null & next ≠ null then
21:             if mark.meta = next.meta then                       ▷ Case 5
22:                 merged ← ({mark.start, next.end} → mark.meta)
23:                 ranges ← ranges − mark − next + merged
24:                 mark ← merged
25:             else if mark.meta ≠ next.meta then                  ▷ Case 6
26:                 extended ← ({mark.start, end} → mark.meta)
27:                 ranges ← ranges − mark + extended
28:                 mark ← next
```

with different metadata in revision 6 (‘B’) and then the metadata is changed again in revision 8 (‘C’). 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 was not parsed again. In revisions 4 and 5, f was parsed, but v did not appear in the source code of those revisions.

When a client adds a vertex to the graph and this vertex already exists, 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}.

Table 1: 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	✓	✓		✓	✓	✓	✓		✓	

Change of internal representation during parsing ↓

Revision	Internal revision range representation	*End*	*Mark*	*Case*
0	A A	-	- {0-0}	1
1	A A A	0	{0-0} {0-1}	5
2	A		{0-1}	
3	A A A	2	{0-1} {0-3}	5
4	A A	3	{0-3} -	4
5	A A	4	- -	2
6	A B A B	5	- {6-6}	3
7	A B		{6-6}	
8	A B C A B C	7	{6-6} {8-8}	6
9	A B C		{8-8}	
final	A B C A B C	9	{8-8} -	4

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 1 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 their data is identical (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 modified again. The algorithm finds the *marked* range {1-2} and another 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 1). 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.

Example 4: We find a total of 2,195,805 JavaScript and Python files across all revisions of *Reddit*. However, 𝕃ISA identified 14,495 blobs during the preliminary Git metadata extraction step, which represent changed versions of these files that actually need to be parsed.The graph loading algorithm transparently takes care of extending the ranges of nodes belonging to previously parsed files and thus creates the lossless representation of all files.

All operations in this algorithm can be executed in constant or 'effective constant' time (where assumptions about hash distribution and vector length

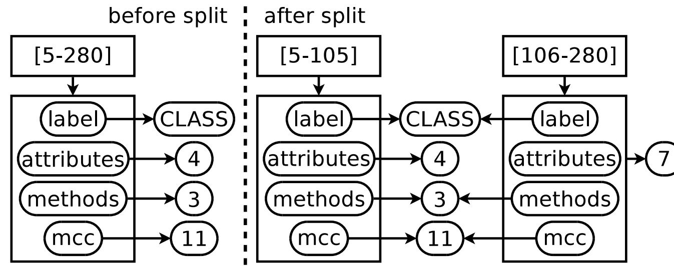


Fig. 4: Range splitting during the computation does not duplicate any data thanks to the use of immutable data structures for storing analysis data.

exist[5]), including (i) identifying or marking the last range to be extended: the reference (or 'mark') is an attribute within the node and ranges of a single node are added chronologically, so there is no need to look up anything, (ii) creating a new range within a node or extending an existing range: immutable Scala map lookups and extensions happen in effective constant time; (iii) adding new vertices: vertices are stored in an `IntHashMap`, which has effective constant time inserts; (iv) queuing a compression task: looking up a vertex to deliver a compression task to also happens in effective constant time; (v) comparing adjacent ranges: vertex metadata (including analysis data) is stored as Scala case classes, which implement a hash-based `equals` function that executes in effective constant time.

Avoiding any complex operations is possible, because, as previously explained in this section, blobs for the same uniquely identified file are parsed chronologically. It would be possible to redesign this algorithm such that files could be parsed in arbitrary order, and then merged, but it would necessitate looking up which existing revision range of a vertex is closest to the newly added revision, which may come after or before the added revision. The resulting algorithm would be much slower than what we propose, relying on the sequential parsing of blobs referencing a specific path.

2.1.5 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 the following section), the metadata in different ranges may start to diverge to a certain degree. For example, a vertex representing a class 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, but it also de-

[5] See `http://goo.gl/7grx8L` for more information on the performance of Scala collections.

fragments vertex ranges where necessary, merging adjacent ranges which store the same metadata.

However, storing completely separate metadata for each revision range would again introduce significant data duplication: for the aforementioned class vertex, 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 4 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 parts of the 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 11 of Listing 2.

Example 5: As pointed out before, the vertex range compression ratio for *Reddit* after parsing was 2,991,074/1,003,984,942 = 0.0030. During the analysis, the number of ranges contained in all vertices increases from 2,991,074 to 3,072,421, indicating that 81,347 ranges had to be split to accommodate differing analysis results. This results in a vertex range compression ratio of 3,072,421/972,713,105 = 0.0031 after the analysis has finished - a marginal increase.

2.2 Multi-revision artifact 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. Specifically, each vertex can *signal* other vertices on outgoing edges to transmit arbitrary messages, *collect* incoming signals, *modify* its own state or *edit* the graph itself by adding vertices and edges. We refer the reader to [79] for an in-depth description of the paradigm.

2.2.1 Formulating analyses

To formulate analyses in 𝕃ISA, the user defines a *data structure* for analysis data (which 𝕃ISA integrates into the metadata of a vertex), a *packet*, which carries data along edges, a *start* function governing how and where in the graph the first signals are emitted for this particular analysis, and a *collect* 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). Furthermore, analyses are written with only one revision in mind, even though 𝕃ISA runs them on all revisions, taking care of the necessary range splits and multi-revision deduplication transparently.

Listing 2: Implementation of McCabe's Complexity in LISA

```
1  /* This implementation of McCabe's complexity is language agnostic. It
2   * uses the ChildCountHelper trait, which keeps track of whether a
3   * vertex has received analysis-specific data from all child vertices.
4   */
5  object MccAnalysis extends Analysis with ChildCountHelper {
6
7    /* Data structure that will hold the complexity value for any vertex
8     * in the graph. By default, the McCabe's complexity of a vertex is
9     * 1 and it is not persisted.
10    */
11   case class Mcc(persist: Boolean, n: Int) extends Data {
12     def this() = this(false, 1)
13   }
14
15   /* The start function determines where in the graph the first signals
16    * are sent. The domain indicates what kind of vertex (e.g. which
17    * programming language) we're dealing with. It transparently maps
18    * the symbols used in the analysis (e.g. 'fork below) to the labels
19    * used by the parser. The state is an immutable object which we act
20    * on - e.g., to send data packets or modify metadata. The start
21    * function must return a new state. Note that in Scala, return
22    * statements are optional, thus the last statement in a method is
23    * automatically used as a return value.
24    */
25   override def start = { implicit domain => state =>
26     // Proceed only if this vertex is a leaf node
27     if (leaf(state))
28       // add an outgoing packet (directed to travel up the tree) to the
29       // state of this vertex. This is done using the '!' operator. The
30       // operator transparently returns the updated state.
31       state ! new MccPacket(
32         // The packet contains 1 by default or 2 if this vertex is a
33         // branch itself, which is possible when using a filtered graph.
34         if (state is 'fork) 2 else 1
35       )
36     // Return the existing state if this is not a leaf vertex.
37     else state
38   }
39
40   /* The definition of the packet that travels along an edge in the
41    * graph. It carries the complexity value (calculated at lower levels
42    * in the tree) and defines the collect function.
43    */
44   class MccPacket(val n: Int) extends AnalysisPacket {
45     override def collect = { implicit domain => state =>
46
47       // The McCabe's complexity of this vertex by definition is
48       // the existing mcc + the incoming mcc - 1
49       val mcc = state[Mcc].n + n - 1
50       // Using allChildren[Mcc], the ChildCountHelper checks whether
51       // this vertex has received an MccPacket from all its children.
52       allChildren[Mcc](state)(
53         // If not, we just store the current aggregated value
54         incomplete = state + Mcc(false, mcc),
55         // If we have collected all child values...
56         complete = {
57           // the mcc value we send out is the existing value of this
58           // vertex, + 1 if this vertex is a branch itself.
59           val newN = if (state is 'fork) mcc + 1 else mcc
60           // Persist the value only if this vertex is a class, method
61           // or file vertex, since we don't want to store the
62           // complexity of intermediate nodes in the tree.
63           val persist = state is ('class, 'unit, 'method, 'file)
64           // store the value (using the '+' operator) and add an
65           // outgoing packet to the state of this vertex (using '!').
66           state + Mcc(persist, mcc) ! new MccPacket(newN)
67         }
68       )
69     }
70   }
71 }
```

As an example, Listing 2 contains an implementation of McCabe's cyclomatic complexity [58]. Formally, McCabe's complexity, is defined as $E-N+2P$ where E is the number of edges in a control flow graph, N the number of nodes and P the number of connected components. All practical tools, however, simply use a complexity of 1 as a baseline and add 1 for each fork in the control flow (e.g., `if` or `while` statements). The data structure used by this analysis (defined on line 11; initialized on line 12) contains a boolean to determine whether or not to persist the computed value for a given vertex and a single integer to hold the actual complexity value. To add data to some state, the `+` operator can be used with any defined `Data` structure. The `!` operator is used to add analysis packets to be sent up the tree. The start function causes an `MccPacket` to be sent out from each *leaf* vertex in the graph (lines 25-37). 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 `collect` function accumulates incoming values according to the definition of cyclomatic complexity (line 49) and once all child values have been received, it sends the complexity of the local vertex to its parent (closure starting at line 56). During the 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 63, where the `persist` property is set `true` if the vertex is a class, method or file. This updated `persist` value is stored together with the updated `mcc` value in line 66. The user could easily add other vertex types (e.g., blocks or closures) if needed.

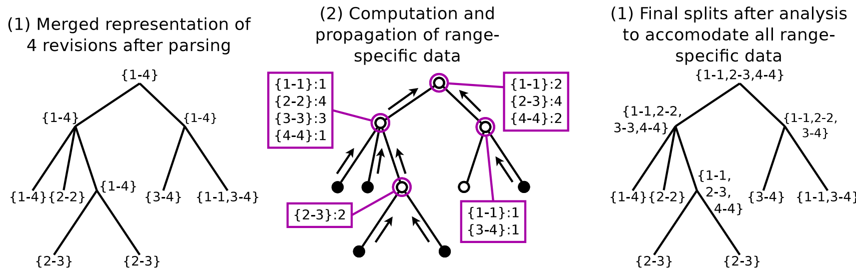


Fig. 5: Let ● be statement nodes, and ○ be function nodes, this example illustrates a statement count analysis and how it affects range splitting. Not only do vertices store range-specific data, but the traveling data (indicated with arrows) carries range-specific data, too. Note that the travel direction of packets depends on the analysis and that analyses can also add additional edges, for example to connect graphs of different files.

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 2.1, 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 revisions is vastly reduced. In Fig. 5, for example, the two bottom most leaf nodes are statements which occur in range {2-3}. Both vertices send a single packet to their parent, which collects this data and transparently splits its only range {1-4} into ranges {1-1,2-3,4-4} to accommodate the data. It then propagates the data further up the tree, where additional splits (and merges) are necessary. In the final configuration, the root node still only has three ranges, because its metadata (including the statement count) is identical for both revisions 2 and 3. This example shows that even on the sub-graph level, a single packet sent regarding an analysis may affect a whole range of revisions.

Thus, thanks to the multi-revision graph representation and a non-centralized, asynchronous computational paradigm to accompany it, we avoid redundancy in analyzing code of multiple revisions at the sub-graph level. 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 number of comma-separated-value (CSV) persistors, for example to store aggregated data over the entire program for each revision, or also on the file level. In general, the `persist` property indicates to the persistor whether a field should be persisted for any vertex in the graph. Global aggregations (for example summing up the method count for all classes) are specified by writing an `extract` function, which obtains some data from any single vertex, and an `aggregate` function, which combines any two extracted values. This way, the aggregation can be executed asynchronously and in parallel over the entire graph.

2.3 Multi-language analysis genericism

Programming languages share certain concepts and many code-related analyses can be expressed for different languages. Thus, while the concrete syntax tree representations can vary greatly for different languages and parsers, their relative structure can be fairly similar. That is, different *entities* in the AST, such as classes, methods or control flow statements have the same collective

Listing 3: Demo.java

```
1 public class Demo {
2   public void run() {
3     for (int i = 1; i < 100; i++) {
4       if (i % 3 == 0 || i % 5 == 0) {
5         System.out.println(i);
6       }
7     }
8   }
9 }
```

meaning when arranged in the same *relative structure*. For example, when comparing the ASTs parsed from a Java and a Python program, the exact sequence, labelling and nesting of vertices leading from a root node to the leaf nodes of a method may differ greatly. However, *relative 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 *relative 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 3 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 straight-forward: there needs to be a mapping between the entity types used in an analysis (e.g., concepts like methods and control flow forks) to the actual labels used by the parser when constructing parse trees (e.g., `MethodDeclaration` and `IfStatement`, `ForStatement` etc.). Formally, this is expressed as 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 formulated as a map from symbols to sets of strings. The implementation of McCabe's complexity in Listing 2 notably does not contain any language-specific labels. Thus, Listing 4 shows an example which is sufficient to be used with the `MccAnalysis` in Listing 2. It specifies how the entities mentioned by the analysis (`'method` and `'fork`) 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 in Listing 3, 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

Listing 4: A vertex label mapping for Java.

```
1  object AntlrJavaParseTree extends Domain {
2    override val mapping = Map(
3      'method -> Set("MethodDeclaration"),
4      'fork   -> Set("IfStatement",
5                     "ForStatement",
6                     "WhileStatement",
7                     "CatchClause",
8                     "SwitchLabel",
9                     "TryStatement",
10                    "DoStatement",
11                    "ConditionalExpression")
12   )
13 }
```

`||` 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.

Example 6: To analyze *Reddit*, 𝕃ISA creates 2,014,720 vertices representing AST nodes from the source code which have been mapped to a symbol. If we disable the filtering in a handicapped version, 8,510,693 nodes are added instead - an increase by a factor of 4.2. Due to the increase in memory consumption, the unfiltered analysis actually required a machine with 128 GB memory instead of 64 GB (with the same number of cores). Furthermore, the additional nodes force the analysis packets to travel much longer distances across the graph, thus the computation itself also takes much longer. We re-ran the analysis using the optimized version of 𝕃ISA on the same (high-memory) hardware, where it needed roughly 15 minutes for parsing and 3.5 minutes for the analysis, executing ~871 million signal and collect operations. With filtering disabled, parsing takes 1 hour and 35 minutes (6.3 times longer) and the computation takes 25 minutes (7.1 times longer), while ~3.1 billion signal/collect operations are executed (3.5 times more). This means that at this scale, the ability to precisely select which kinds nodes in the original source data are relevant for an analysis can make a large difference.

2.4 Limitations

We are aware that our current implementation of the 𝕃ISA framework suffers from several limitations, that were not relevant for the experiments presented in this paper, but that we enumerate in the following sections.

Vertex identification. A technical limitation arises from how we naively identify vertices supplied by the currently integrated parsers. When parsing sources, the identifier of a vertex always corresponds to the file and syntax tree location of a node (as explained in Section 2.1). Hence, renaming a file or reordering methods in a class bypasses this identification scheme and causes the creation of 'new' vertices, even if the file or methods themselves have not changed. This reduces the effectiveness of the range-compression. Note, however, that the actual computation of results in 𝕃ISA is robust to these kinds of changes. For example, when computing a metric for classes and persisting the results based on the *class name* (as done in the class-level persistor that ships with 𝕃ISA), the internal identification of the vertex representing that class has no relevance and the results are the same no matter which file the class was contained in across different revisions. However, 𝕃ISA would not be able to currently track whether a method or class has been renamed.

The underlying problem of identifying code entities is generally hard to solve, especially in a language-independent manner [47,88], and might require an identification scheme specific to a particular experiment. For example, to

effectively differentiate several overloaded Java methods that share the same name but have different signatures, it would be necessary to identify them in a fully-qualified way that also considers the type and names of the parameters.

Support for additional version control systems. The current implementation of 𝕃ISA supports a direct extraction of source code from Git repositories. While Git is arguably the most popular version control system today, at least in open source, future work might require the analysis of projects that are managed in different version control systems. Two alternatives exist: either the existing repository is converted to Git, e.g., through conversion tools such as `svn2git` or `git-cvsimport`, and the project is then analyzed with the existing tooling for Git repositories. Alternatively, the `SourceAgent` interface can be implemented for different version control systems. Note that the asynchronous parsing of revisions is a Scala trait which can be added to other source agents as well, as long as they provide some method of accessing files from different revisions independently. Thus, adding additional version control systems is straightforward and only a matter of spending engineering effort.

Partial order of commits. Given a repository URL, 𝕃ISA processes all revisions that are ancestors of the tip of the default branch. By default, this traversal is done in reverse chronological order. Modern version control systems like Git heavily rely on forks and joins in their commit history by the use of branches. The history in such branches is no longer sequential and the commits have to be ordered using some particular strategy. While branches typically work on unrelated tasks, it can occasionally happen that committers create conflicts by changing the same file simultaneously. If all commits are flattened out to a sequence by the aforementioned traversal strategy, it could appear as if the same file is changed back and forth again, as the versions from two different branches alternate, which negatively impacts the range compression – but again, not the computation of results for any particular revision. In an informal comparison of different traversal strategies, we could not find a single flattening strategy that is always preferable over the default one.

2.5 Summary of the approach

Compared to traditional approaches, 𝕃ISA avoids redundancies at every juncture of a multi-revision 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 artifacts. Both vertices and metadata present in multiple revisions are stored only once and the computations themselves are also executed only once for each revision range at the vertex/sub-tree level, avoiding the expensive recomputation 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.

3 Evaluation

The primary goal of this research is to accelerate the analysis of multi-revision artifacts in a software evolution context. To evaluate 𝕃ISA's effectiveness and to showcase its applicability, we evaluate 𝕃ISA along three dimensions.

Performance: How fast can 𝕃ISA analyze projects having different numbers of revisions, different amounts of source code or containing code written in different programming languages?

Resources: What are 𝕃ISA's hardware requirements in terms of memory consumption and CPU power?

Adaptability: How flexible is 𝕃ISA with regard to analyzing new problems or additional programming languages?

To analyze these dimensions, we perform a sequence of experiments, in which each experiment has a primary focus on one aspect:

Generality (Analyses): To illustrate 𝕃ISA's *support for varying kinds of code analyses*, we formulated two distinct analysis suites in 𝕃ISA: one to compute object-oriented metrics [53] applicable to any programming language supporting those concepts and another to detect 'Pythonic idioms' [71] in Python programs (Section 3.1).

Large-Scale Performance: To *measure performance in a large-scale scenario*, we analyzed 4,000 projects; 1,000 projects each written primarily in Java, C#, Python and JavaScript, applying the object-oriented analysis suite to all 4,000 projects and applying the Pythonic analysis suite to the Python projects only. During the execution we captured detailed time-based performance statistics (Section 3.2).

Impact of redundancy removal techniques: To evaluate the *impact of different redundancy removal techniques* of 𝕃ISA, we created several handicapped versions in which each technique is disabled once, and apply them to our running example, the *Reddit* project (Section 3.3).

Resource Consumption: To *measure resource consumption*, we performed an additional study in which we have created a sample of all available projects by selecting projects with differing sizes and number of revisions. We have then (i) captured the memory footprint during the analysis to determine memory requirements, and (ii) re-ran the analysis with different amounts of available memory and CPU cores to determine the impact of resource availability on the computation speed (Section 3.4).

Generality (Languages): Finally, we have adapted 𝕃ISA to another programming language, namely Lua, to *illustrate generalizability to other programming languages*. We show that the unchanged object-oriented metrics analysis suite can be applied in the new context (Section 3.5).

This evaluation omits the comparison of 𝕃ISA with other tools, since we have already presented this comparison in previous work [8] using a less optimized version of 𝕃ISA. The results are summarized in Section 3.2.

All our experiments are implemented as short Scala programs that are being compiled and executed on server hardware using simple Bash scripts. 𝕃ISA's hardware requirements grow with project size, number of revisions, and the size of changes between revisions. While most projects could be analyzed on commodity hardware, we ran all computations of the large-scale study on a powerful, shared server.[6] To ensure successful runs for the largest projects, we dedicated 512 GB memory and 32 cores independent of project size. The *Reddit* running example was analyzed on instances with 12 cores and although we established that 20 GB of memory are sufficient to analyze the project, we decided to dedicate either 64 GB or 128 GB of memory, depending on the requirements of the handicapped versions, ensuring that equal resources were used in each individual comparison between an optimized and handicapped version of 𝕃ISA.

3.1 Implementing analyses in 𝕃ISA

In this section, we discuss how the two distinct analysis suites used in the remainder of our evaluation were formulated. We also discuss advantages as well as shortcomings of the current approach.

Analysis formulation. Analyses in 𝕃ISA are formulated as Scala source code according to the Signal/Collect paradigm as outlined in Section 2.2.

The object-oriented analysis suite computes 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 detects the BrainMethod [53] code smell. It also computes the number of direct children and total number of vertices beneath each vertex and captures the number of files and lines of code for each project and revision. We chose this set of metrics over others like coupling or inheritance depth, because we are working on plain-text ASTs and source code parsers cannot produce the necessary information to infer such metrics (we discuss possible alternatives to ASTs in Section 4.2). Implementing these analyses required 203 lines of Scala code (not counting comments, blank lines and single closing braces).

For the "Pythonic" analysis suite, we wanted to detect the following idioms, as described in [71]: List Comprehensions, Generator Expressions, Lambdas, Named Decorators, Named Tuples, Default Dicts, Ordered Dicts, Dequeues, `finally` blocks, magic functions, nested functions as well as `yield` and `with` statements. This required writing 180 lines of Scala code in 𝕃ISA.

We ensured the correctness of the formulated analyses (and thus the range compression algorithm) for each language by creating sample projects[7] which contain a large number of code combinations varying across multiple revisions, such as differently nested `if` statements, nested classes and methods, etc. We

[6] SuperMicro SuperServer 4048B-TR4FT, 64 Intel Xeon E7-4850 CPUs with 128 threads, 3TB memory in total

[7] The example projects for each language, ending in '-example-repository', can be found online at `https://bitbucket.org/account/user/sealuzh/projects/LISA`

then manually calculated the expected code metrics, confirming that they match the results procured through 𝕃ISA.

Discussion. The Signal/Collect paradigm requires a certain mindset when formulating analyses. For example, to count the methods in a class, `'method` nodes in the AST need to *signal* the number 1 upward in the AST, while `'class` nodes need to *collect*, add up and persist these numbers. This is very straightforward to implement and applies to many counter-based metrics to be computed. The implementation for McCabe's complexity, shown in Listing 2 is slightly more complex, as it behaves differently depending on whether all child vertices of a node have already signaled, or not. A different approach is necessary when counting some of the Pythonic idioms: in some cases, the idioms are not identifiable from the type of an AST node, because they are all expressed using the same underlying node type, such as a function call. In this case, the content of the node, i.e., the literal call string, needs to be examined. To find brain methods, a purely local analysis is possible: it simply looks at the complexity, control flow nesting depth, vertex and variable count already computed by other analyses for this method, and makes a local determination, simply storing a boolean if the metrics indicate a brain method.

Thanks to the lightweight mappings, analyses are generally formulated for all languages supporting a certain concept at the same time. That said, analyses formulated using the Signal/Collect paradigm in Scala can be overly verbose. For example, a simple method counting consists of 13 lines of code, out of which 7 are scaffolding required by Signal/Collect. This increases the complexity of all analyses that we implemented, and is a major, if not the most important drawback of the current implementation of 𝕃ISA. We are considering better solutions (e.g., a domain specific language) in Section 4.2.

3.2 Analysis speed

We want to know how quickly 𝕃ISA can analyze the entire history of a large number of projects. In this section, we describe how we selected the projects and how the analyses we formulated were applied. Then we discuss 𝕃ISA's performance characteristics.

3.2.1 Data gathering

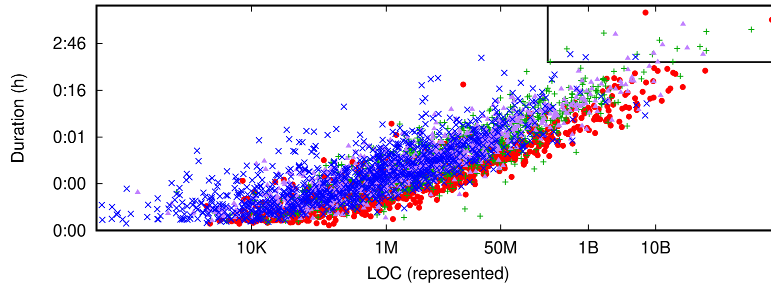
Project list. We first queried the GitHub API for Java, C#, JavaScript and Python projects, sorted by their number of stars. The GitHub API returns a maximum of 1,000 results per query, thus we obtained a list of 4,000 repositories containing code written, primarily, in these 4 languages. Some of these repositories do not contain actual software projects, but rather tutorials and other documentation. We cleaned the list of projects by searching for strings like 'awesome-', a commonly used prefix for repositories containing just a collection of links, or 'tutorial', 'book' and a few others. About a dozen of these repositories remained unnoticed and were only identified after analyzing them

and obtaining few, if any data. All such repositories were finally replaced by selecting top projects gathered from research reports and other sources for popular projects that were not already on the list [1, 6].

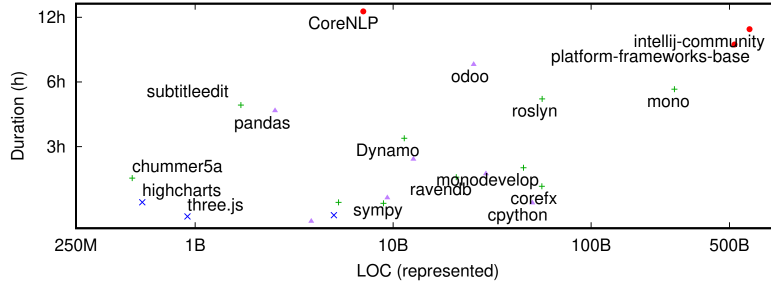
Parsers. We used all three different ways of integrating additional data sources in 𝕃ISA to parse the source code mentioned in Section 2.1. For Java and `C#`, we fed *ANTLRv4* grammars into 𝕃ISA to automatically generate the parsers. For JavaScript, we adapted the *Nashorn* parser which ships with JDK8, because the ANTLR-generated parser used in an earlier study [9] was very slow (for example, parsing the Unitech PM2 project takes almost 12 minutes to parse using the ANTLR parser, but just 36 seconds using the Nashorn parser). Finally, for Python, we call the native *cpython* parser, as it is much more robust in parsing code from different Python versions than ANTLR parsers based on version-specific grammars. For inter-process communication between 𝕃ISA and cpython, we used a simple ad-hoc JSON representation of ASTs.[8] We proceeded to define suitable language mappings to match the AST entities relevant to our analyses to the ones used in the ANTLR grammars, Nashorn API and cpython parser.[9] Note that we enabled only one parser per project, even though 𝕃ISA supports the analysis of multiple languages within the same computation (as demonstrated in the running example used in Section 2, where both the Python and JavaScript parsers were enabled). We wanted to observe 𝕃ISA's performance characteristics on a language-by-language basis, so that we can assess the impact of the used parser on overall performance.

Analysis execution. We applied our tool to the Git URLs of the 4,000 projects to compute the metrics for all revisions which are ancestors to the Git `HEAD`, including ancestral branches, (sorted using Git's default sorting: reverse chronological) and persisted those metrics at the project-level. This means that we aggregated a global value (for example the total method count, or the sum of all complexities or unique control flow paths) for every revision of each project. Thus, for every project, we obtained a CSV file where each line contains the aggregated metrics on a single revision of the project. Note that several alternative persistors exist, for example to aggregate data at the file, class or method level, however these were not needed for any practical studies we have performed so far. In total, this gave us 1.6 GB of data on the 7,111,358 revisions analyzed. The 512 GB of memory we provided 𝕃ISA with were sufficient for all but three projects. The analyses of the *Android Platform Frameworks*, *JetBrains IntelliJ* and *Stanford CoreNLP* projects did not crash, but would not complete in reasonable time. For these projects, we split the analysis into several pieces, specifying start and end commits for a range of revisions to be analyzed by 𝕃ISA. Naturally, this split implies that some code needs to be parsed twice: when the first, e.g., 50,000 revisions are processed, no redundancies occur as usual. However, to analyze the subsequent 50,000 revisions, all code that was added previously needs to be parsed again. We discuss how this problem could be solved in Section 4.2.

[8] The JSON representation can be found in the repository, here: `https://goo.gl/oMDxzv`
[9] All parser integrations and mappings can be found here: `https://goo.gl/6pT7sG`



(a) All projects. The area in the top right box is enlarged in Fig. 6b



(b) Very large projects only.

Fig. 6: Analysis duration vs. LOC contained in revisions for several Java (●), C# (+), JavaScript (×) and Python (▲) projects.

Performance Monitoring. 𝕃ISA captures 32 different runtime statistics, including compression ratios, durations for each individual part of an analysis and size metrics such as the number of files and lines parsed. At the end of an execution, these statistics are stored in a separate CSV file.

3.2.2 Performance characteristics

Data overview. Table 2 shows runtime statistics and metadata for the analysis we performed, while Fig. 6a and Fig. 6b visualize the projects by plotting their size in terms of lines of code on the x-axis versus the analysis runtime on the y-axis. The largest projects we analyzed, in terms of the number of revisions, were the *Android Platform Frameworks* for Java (285k revisions), the *Mono* programming language for C# (111k revisions), the *Yahoo User Interface Library* for JavaScript (25k revisions), and the *Odoo suite of business web applications* for Python (108k revisions). These projects alone account for 7.5% of all analyzed revisions. Other large and renowned projects in our dataset include Elasticsearch, Google Guava and Apache Spark for Java, the Roslyn compiler, CoreFX libraries and PowerShell for C#, popular web frameworks such as JQuery, Angular and D3 for JavaScript and the Django CMS, the Requests library and SciPy for Python. Analyzing the source code of all 7.1

Table 2: Artifact study A: number of revisions, files and lines analyzed; compression ratios achieved; and runtimes during analysis.

Metric		Java	C#	JavaScript	Python
# Projects		1,000	1,000	1,000	1,000
Revisions analyzed	total	1,991,638	1,419,317	1,478,521	2,269,785
	largest	[1]285,050	[2]111,775	[3]25,381	[4]108,236
	average	1,991	1,419	1,478	2,269
	median	235	462	603	565
Files represented	total	10,786,513,588	3,163,589,790	381,717,977	1,260,140,128
	largest	[5]6,727,250,559	[2]1,289,015,226	[6]62,178,658	[4]181,075,631
	average	10,786,513	3,163,589	381,717	1,260,140
	median	9,089	71,195	9,924	17,403
Files parsed and analyzed	total	10,246,923	9,189,319	4,363,335	5,544,396
	largest	[5]1,648,265	[7]579,858	[3]267,447	[4]663,654
	average	10,246	9,189	4,363	5,544
	median	484	2,004	1,006	842
Lines represented	total	1,560,176,972,238	643,839,524,403	43,307,476,687	281,001,712,196
	largest	[5]631,736,446,886	[2]263,533,058,384	[8]7,334,015,580	[9]50,493,471,752
	average	1,560,176,972	643,839,524	43,307,476	281,001,712
	median	1,120,037	8,262,959	582,631	2,382,768
Lines parsed and analyzed	total	4,070,090,931	3,238,816,779	619,061,540	2,762,035,407
	largest	[1]895,632,889	[10]348,628,995	[3]179,612,297	[4]223,881,075
	average	4,070,090	3,238,816	619,061	2,762,035
	median	99,331	428,155	35,582	219,057
Range (AST) compression factor	average	0.034	0.023	0.028	0.041
	median	0.023	0.012	0.017	0.024
	worst	[11]0.362	[12]0.359	[13]0.523	[14]0.349
	best	[5]0.000053	[2]0.000105	[15]0.000136	[16]0.000276
Parse filtering factor	average	0.152	0.109	0.467	0.208
	median	0.154	0.110	0.478	0.209
	worst	[17]0.233	[18]0.202	[19]0.667	[20]0.274
	best	[21]0.056	[22]0.044	[23]0.012	[24]0.017
Runtime	cloning	2h 55m	2h 32m	1h 18m	1h 22m
	metadata extraction	1h 21m	17m 18s	8m 11s	18m 35s
	parsing	1d 15h	2d 16h	18h 19m	1d 17h
	analysis	2h 42m	3h 23m	3h 28m	7h 52m
	persistence	7h 37m	2h 18m	22m 33s	1h 51m
	total	2d 5h	3d 0h	23h 38m	2d 4h
	shortest	[25]1.292s	[26]1.506s	[27]1.365s	[28]1.630s
	longest	[29]12h 47m	[2]5h 33m	[30]1h 39m	[4]7h 14m
	average	3m 14s	4m 21s	1m 25s	3m 8s
	median	8.350s	40.499s	14.383s	24.448s
	total avg./rev.	97ms	183ms	57ms	83ms
	median avg./rev.	41ms	88ms	25ms	48ms

1: android_platform_frameworks_base, 2: mono_mono, 3: yui_yui3, 4: odoo_odoo, 5: JetBrains_intellij-community, 6: nodejs_node, 7: Azure_azure-powershell, 8: SeleniumHQ_selenium, 9: python_cpython, 10: dotnet_roslyn, 11: hongyangAndroid_android-percent-support-extend, 12: tg123_commandlinefu.cn, 13: bartonhammond_snowflake, 14: corna_me_cleaner, 15: Automattic_wp-calypso, 16: trustedsec_social-engineer-toolkit, 17: Freelander_Android_Data, 18: dotnet_standard, 19: airbnb_react-native-maps, 20: kennethreitz_flask-sockets, 21: addthis_stream-lib, 22: Unity-Technologies_ScriptableRenderLoop, 23: timuric_Content-generator-sketch-plugin, 24: fxsjy_jieba, 25: spring-projects_spring-mvc-showcase, 26: madskristensen_ShortcutExporter, 27: olistic_warriorjs, 28: drathier_stack-overflow-import, 29: stanfordnlp_CoreNLP, 30: highcharts_highcharts

million program revisions took 8.3 days at 101ms per revision on average, i.e., close to 10 revisions per second. Calculating the per-revision runtime for each project in each language yields a median runtime per revision of 30ms for Java, 79ms for C#, 19ms for JavaScript and 42ms for Python.

Redundancy removal. The range compression technique we propose is extremely effective: The range compression factor in Table 2 reflects the ratio of the number of *revision ranges in AST vertices* needed to represent multiple program revisions, without loss, versus the actual number of vertices represented by all revisions individually. On average for each language, this compression ratio ranges between 0.023 and 0.041, meaning that to analyze multiple revisions of a program, 𝕃ISA needs just 4% of the memory and computational resources required to analyze each revision separately. For very large projects, the range compression factor can be extremely low. For example, during the analysis of the *Android Platform Frameworks*, the compression factor reached 0.000053. For analyzing *Mono*, the compression factor was 0.000105. This means that especially when analyzing large projects, the multi-revision graph compression saves significant overhead. But even for projects consisting only of a few dozen revisions, the range compression factor can quickly drop below 0.4 for Java, C# and Python projects. For JavaScript projects, compression ratio factors as high as 0.53 were observed for very small projects. JavaScript projects are distinctively different because code is generally contained in fewer files (as reflected in Table 2); it is not unusual for an entire project to be managed in a single file. This means that (i) the ratio of actual changes in a single commit to the amount of source code that needs to be re-parsed is greater – even if only one line is changed in a commit, an entire file needs to be re-parsed (i.e., changes in languages where code is spread out over a larger number of files are quicker to re-parse), and (ii) the ASTs for each file are larger. This means that changes more easily introduce hard-to-avoid redundancies. Due to how AST nodes are identified, high churn within a file – possibly at high levels in the AST – causes more vertices and ranges to be created than for languages were ASTs are shallower due to them being distributed across many files.

When performing computations by first loading code from several revisions and then analyzing it in a second step, selecting exactly which parts of an AST are relevant for an analysis can be extremely advantageous. The effective filtering factor largely depends on the parser and how it constructs ASTs. For Java and C#, using ANTLRv4 grammars, this factor is roughly 0.15 and 0.11 respectively. For JavaScript as parsed by the Nashorn parser, this factor is 0.46, i.e., still less than half of all vertices are loaded into the graph. For Python, the factor is 0.21.

Performance compared to other tools. A fair one-on-one comparison to other tools is not feasible, as each tool has different feature sets, restrictions and capabilities. In previous work [8], we compared the performance of a 𝕃ISA prototype, which lacked many of the performance-enhancing techniques discussed in Section 2, except for the range compression, to two existing analysis

tools, namely inFusion [2] and SOFAS [34] for analyzing the *AspectJ* project. In that comparison, the prototype took 1:31 minutes 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 1,000 revisions, whereas using the other tools, each additional revision to be analyzed incurs the same cost, which actually increases with the growth of the project in later revisions.

We can however compare the performance of 𝕃ISA directly to the original prototype and by extension, to the tools used in the original study [8]. To analyze all 7,778 revisions of AspectJ, the prototype spent 650ms on average per revision, while 𝕃ISA spent only 45ms. 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 31ms on parsing and 4ms on the analysis. The resulting average of 45ms per revision (including metadata extraction and persistence) is just above the median average across all Java projects we analyzed, as shown at the bottom of in Table 2.

The parsing speed improvement can be attributed to both the filtered parsing, enabled by the lightweight mappings (Section 2.3), the asynchronous multi-revision parsing algorithm and the parallelized Git metadata retrieval (Section 2.1) and persistence (Section 2.2). 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 is directly connected to the filtered parsing, as the signals need to travel much shorter distances within the graph.

3.3 Measuring the performance benefit of individual features

We wanted to determine the impact of each individual redundancy removal technique implemented in 𝕃ISA. Some of the impact, namely the compression ratio achieved by the multi-revision graph representation and the reduction in file access through the incremental approach, can be quantified by simply observing the metrics reported by 𝕃ISA. However, to also measure the impact in terms of actual performance benefits, we needed to suppress each of the optimized features described in Section 2 and make a one-on-one comparison between the optimized version and each handicapped variant.

Creating and comparing handicapped versions of 𝕃ISA. We did this by creating separate source branches[10] of 𝕃ISA, with one individual part of the library code replaced by a more common, less optimized implementation, and then compiling a drop-in replacement JAR used during the computation. Thus, we introduced the following handicaps:

[10] These are the handicap-* branches visible in the 𝕃ISA open source repository

Table 3: Impact of redundancy removal techniques on total execution time.

Variant	Runtime [hh:mm]	Slowdown factor
Optimized 𝕃ISA	0:15h	1.0
Analyzing revisions individually (A)	15:09h	57.8
Parsing sources in sequence (B)	0:42h	2.7
Not filtering AST node types (C)	2:25h	7.1

(A) checking out each revision into a working directory, instead of reading them directly as blobs from the Git database, and thus analyzing each revision in sequence,
(B) parsing source code of different revisions in sequence, instead of parallelizing the parsing across all revisions,
(C) loading all AST vertices provided by a parser into the graph instead of discarding irrelevant nodes.

Effectiveness of different techniques. The results of all comparisons summarized in Table 3 highlight that each individual technique contributes significantly to the runtime reduction.

Specifically, from the runtime statistics, we learn that by sharing state across multiple revisions at the level of AST nodes, 𝕃ISA represents the complete state of the project in every revision using just 0.3% of the nodes that are actually present across all revisions, without any information loss. We also learn that by not parsing and analyzing each revision individually, only 0.7% of files and 2.8% of lines (those with actual changes) needed to be read.

Handicapped variant A needs over 15 hours to analyze all 7,947 revisions of *Reddit* in sequence, compared to 𝕃ISA's 16 minutes, a factor of 57.8. All else being equal, parsing source code of different revisions in sequence as done by handicapped variant B impacts the analysis duration by a factor of 3.2: the handicapped variant needs 37 minutes to parse all revisions, while 𝕃ISA only needs about 12 minutes. This affects the total runtime by a factor of 2.7, as the handicapped variant needs 42 minutes to execute. By not filtering those AST nodes which are unnecessary for a particular analysis, 4.2 times more nodes needed to be stored and involved in the computation, as done by handicapped variant C. The execution needed 3.5 times more operations and took 7.1 times longer, for a total runtime of 2 hours and 25 minutes instead of 𝕃ISA's 16 minutes.

3.4 Resource requirements

𝕃ISA stores the entire multi-revision graph and all computed data in memory. The majority of projects on GitHub, even among popular ones, are fairly small and could be analyzed on commodity hardware. However, projects with many revisions or a large code base can require significant amounts of memory and it is hard to guess, how much is needed, exactly. To better estimate how many

resources in terms of memory and CPU power 𝕃ISA requires for analyzing a particular project, we designed the following three experiments.

Experiments. We designed two experiments to learn about 𝕃ISA's memory consumption and one to determine the impact of available CPU power. Profiling memory consumption of the JVM is non-trivial mainly due to the autonomy of its garbage collectors in deciding how to manage memory and the possibility of transparently shared memory (e.g., when using Strings or other immutables). In 𝕃ISA, we use the G1 garbage collector (G1GC) as we found it to provide the best performance. However, this concurrent garbage collector (GC) behaves differently under different circumstances. It desperately tries to free memory even when almost no 'real' work is being done and memory allocation largely depends on the amount of available memory. Furthermore, the JVM over-allocates memory in batches. Memory consumption could be monitored by simply observing the consumption of the JVM at the operating system level, but this would be inaccurate because of how the JVM over-allocates memory. GUI tools such as VisualVM or JProfiler could be used, but this would be hard to automate. Another option would be to write a custom java agent to read out memory statistics using the Java Management Extensions (JMX), but finally we chose to use the GC log and simply observe how much memory the GC recognizes as containing 'live objects'. This gives a fairly reliable reading because the measurement is done directly during garbage collection. Based on this, we performed the following three experiments on a subset of the projects in our large-scale study:

1. **Experiment 1 (full memory)**: We re-ran the analysis of each project providing the full 512 GB of memory and 32 cores, like in the original large-scale study. We enabled GC logging and stored the logs for each analysis. This allows us to observe how much memory 𝕃ISA allocates and uses under ideal circumstances.
2. **Experiment 2 (limited memory)**: Here, we re-ran the analyses three times, again providing 32 cores, but lowering the available memory to 128 GB, 32 GB and 8 GB. We also limited the total runtime to 3 hours. This allows us to observe behavior when memory availability is low, namely (i) projects that can no longer be analyzed within reasonable time, and (ii) the impact on resource allocation, garbage collection, and hence, overall performance.
3. **Experiment 3 (limited CPU)**: In this last experiment, we provided the full 512 GB for all analyses, but limited the number of available CPU cores to 16, 8 and 2, respectively. This allows us to accurately determine how well 𝕃ISA makes use of additional computing resources.

All experiments were performed on the same hardware as the large-scale study. Even though memory and CPU cores have been reserved for exclusive use, some variability is expected, because the infrastructure is still a shared environment with other users running compute jobs on the same hardware.

Table 4: Projects selected for the resource study

Metric	Size	Java	C#	JavaScript	Python
Lines Parsed	150M	Alluxio	Mono Develop	Yui3	Saltstack
	15M	Terasology	PowerShell	Plotly	Ansible
	1.5M	JetBrains IdeaVim	AspNet OpenIdConnect.Server	Leaflet	Eventlet
	150K	Crawler4J	EntityFramework.Extended	noVNC	Facebook PathPicker
	15K	Android AppMsg	RaspberryPi.Net	Parse Server	Jinja Assets Compressor
Revisions	25K	Elasticsarch	DotNet Corefx	Yui3	Theano
	13K	Apache Groovy	MediaBrowser Emby	Jquery Mobile	Mozilla Kuma
	7K	QueryDSL	Castleproject Core	Paper.js	Boto
	3.5K	Google J2ObjC	AspNet MVC	Unitech PM2	Thunder
	1.5K	Bukkit	Newtonsoft.json	Parsley.js	Pika

Project selection. We wanted to select projects for each language across a wide range of sizes, including both small and very large projects. However, we also wanted to compare the resource consumption across different languages, thus the selected projects from different languages should be comparable in size. To achieve both these goals, we applied the following selection strategy:

1. For each of the four programming languages in our large-scale study, we sorted the projects twice: once according to the number of revisions analyzed, and once according to the total number of files parsed during the analysis. We believe that these two metrics may serve as good proxies for the total amount of work necessary.
2. For each of these two metrics, we compared the four resulting sorted lists and determined the maximum value for which comparable project exist in every language. For both the number of revisions and the number of files parsed, the Yui3 project turned out to be the largest JavaScript project available (whereas larger projects exist for the other languages, which we did not include in the study because JavaScript projects of corresponding size were not available).
3. We selected projects with comparable metrics for the other three languages. These projects serve as the upper bound within each metric.
4. Finally, we selected 4 additional projects for each language and metric, each successive one being smaller by a constant factor. We found that 2 for the number of revisions and 10 for the number of lines parsed gave us a good distribution across all project sizes.

As such, we ended up with 10 projects per language, where 5 were selected according to the number of revisions and 5 according to the number of lines parsed and the requirement to have a similarly-sized counterpart in the other 3 languages. Table 4 displays the selected projects.

Note that the smallest projects included in this study are still comparatively big. 75% to 81% of projects in our dataset have fewer than 1.5K revisions. In terms of lines parsed, 7% to 40% of projects (depending on the language) are smaller. We did not choose even smaller projects, because in our experience, $\mathbb{L}$ISA can easily deal with smaller projects even on low-end machines, which we confirm in this study.

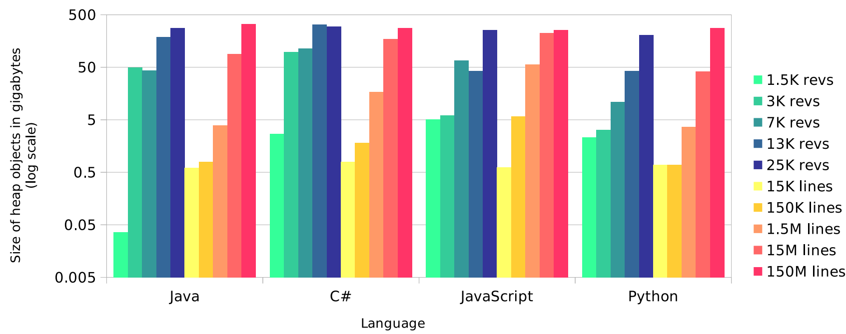


Fig. 7: Peak size of allocated objects on the heap as reported by the G1 garbage collector for projects of different sizes. Note that the number of lines refers to the amount of code **actually parsed** for the analysis. The real amount of source code represented by all revisions is larger by several orders of magnitude as described in Table 2.

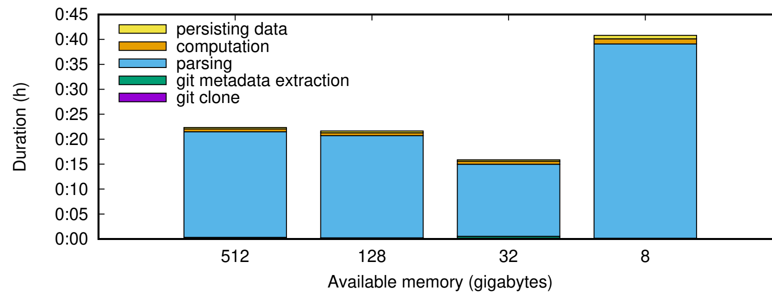


Fig. 8: Analysis runtimes for the medium-sized PowerShell C# project (5,554 revisions, 17,330,091 lines parsed representing 3,263,786,422 lines in all revisions) on server instances with different amounts of memory, all running on 32 CPU cores.

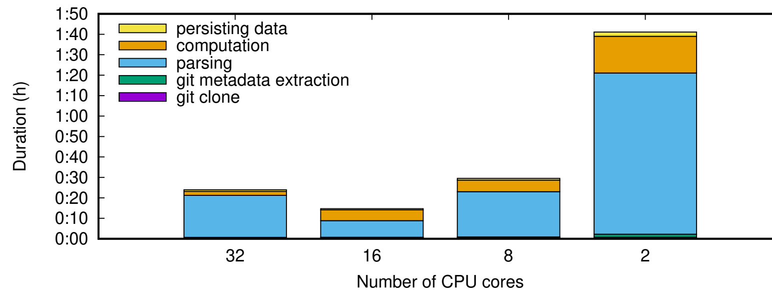


Fig. 9: Analysis runtimes for the medium-sized Mozilla Kuma Python project (13,954 revisions, 1,231,549 lines parsed representing 1,304,524,320 lines in all revisions) on server instances with different numbers of cores, all running with 512 GB of memory.

Experimental results. Figure 7 displays the results of the first experiment. Unsurprisingly, larger projects need significantly more memory to be processed. That said, the memory requirements for different projects with similar size metrics may still heavily diverge. For example, the four projects with roughly 15M lines parsed required between 40 GB and 220 GB of memory during analysis. From the second experiment, we learn that additional memory does not accelerate the computation for projects where more memory is available than required. Figure 8, however, shows that when the available memory is just barely sufficient, the processing time increases significantly. Analyzing the medium-sized PowerShell project with only 8 GB of memory available doubled the processing time, since the garbage collector needed to constantly free up memory. Larger projects either crashed with an out-of-memory error or timed out after 3 hours, which is reflected in Figs. 10 and 11. In terms of size, it appears that projects with up to 3,000 revisions and 1.5M lines parsed can be analyzed on basic commodity hardware (this covers 82% of the 4,000 projects we analyzed). The third experiment gave us both expected and unexpected results. Figure 9 shows that the total processing time steadily decreases from 1:40h using 2 cores to 30min using 8 cores and 15min using 16 cores. Surprisingly, using 32 cores increased the processing time. A similar trend is visible in Figs. 12 and 13. Even more surprisingly, the computation time shown in Figure 9 was halved from 16 to 32 cores, while the parsing time doubled. We discovered that this was caused by a flaw in the configuration of 𝕃ISA, in that it uses a fixed-size pool of 28 workers to parallelize parsing (while during the computation, it uses a number of workers equal to the number of available cores). It appears that this causes no problems when there is a lower number of cores available, i.e., some workers share the same core. However, when there is a larger number of CPUs available, performance is negatively impacted. This might simply be due to the workers not getting pinned to a specific core which reduces the efficiency in memory access and caching. We plan to further investigate this issue and mitigate it in a future release of 𝕃ISA.

Memory is the main limiting factor in analyzing large projects using 𝕃ISA. However, it remains difficult to estimate in advance, how much memory is needed. We discuss possible solutions to this problem in Section 4.2.

3.5 Adapting 𝕃ISA to other programming languages

As a final demonstration of 𝕃ISA's adaptability, we go through the process of adding support for another programming language, namely Lua, to 𝕃ISA.[11] As discussed in Section 2.3, programming languages share some similarities, especially regarding structure (like how different AST node types are nested), even though they use different grammars and absolutely differ in certain ways. For example, Lua does not have classes as a language-level concept. Nevertheless,

[11] This demonstration is part of the lisa-quickstart repository: `https://bitbucket.org/sealuzh/lisa-quickstart/`

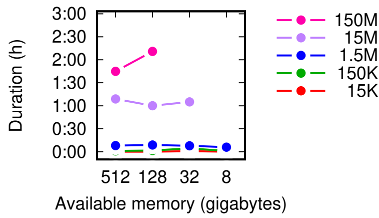


Fig. 10: Runtimes of JS projects with a different number of lines parsed.

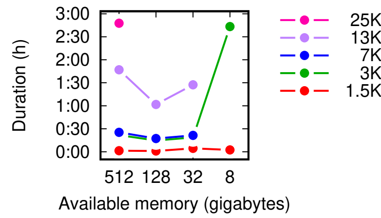


Fig. 11: Runtimes of C# projects with a different number of revisions.

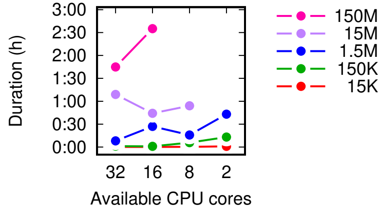


Fig. 12: Runtimes of JS projects with a different number of lines parsed.

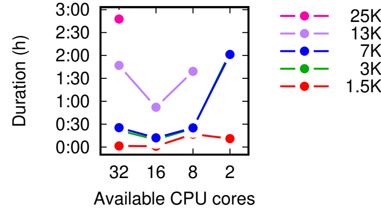


Fig. 13: Runtimes of C# projects with a different number of revisions.

other object-oriented analyses implemented in LISA still apply, for example counting attributes or computing cyclomatic complexity of functions and files.

Implementation. Listing 5 contains the complete implementation and demonstrates how little work is necessary to support additional languages in LISA. The following steps are necessary.

1. Copying an existing *ANTLRv4* grammar into the `src/main/antlr4` directory. We used the grammar made available at `https://github.com/antlr/grammars-v4`.
2. Defining a language mapping from the symbols used by the object-oriented analysis suite to the labels used in the grammar (lines 15 to 26).
3. Defining the boilerplate class which integrates the ANTLR-generated parser in LISA (lines 28 to 33).

The implementation took us roughly 30 minutes, mainly to look up and map the labels used by the ANTLR grammar. Using this minimal implementation, LISA's generic analyses (which are part of the `UniversalAnalysisSuite` used on line 44) can thus be applied to Lua projects. Metrics for any matching node types (i.e., methods, blocks and files) will be computed. Nothing more is necessary to persist these metrics at the file level. However, if the goal is to persist metrics for individual functions, a little more work is necessary. The problem is, that in this Lua grammar, function names are not stored as literals of the function nodes (labeled "Function" by the grammar). Instead, they are stored in child nodes, labeled "Funcname", beneath the function nodes. This means that to identify function nodes by their name, it is necessary to signal the function name from the "Funcname" to the actual "Function" nodes,

Listing 5: Adding Lua support in 𝕃ISA

```
1  package org.example
2  import ch.uzh.ifi.seal.lisa.core._
3  import ch.uzh.ifi.seal.lisa.core.public._
4  import ch.uzh.ifi.seal.lisa.antlr._
5  import ch.uzh.ifi.seal.lisa.module.analysis._
6  import ch.uzh.ifi.seal.lisa.module.persistence.CSVPersistence
7  // Parser classes generated by ANTLRv4. Beyond copying the grammar into
8  // the correct directory, no additional effort is necessary, as LISA
9  // automatically generates parsers for available grammars.
10 import org.example.parser.LuaLexer
11 import org.example.parser.LuaParser
12 // Mapping semantic concepts used by the existing object-oriented
13 // analyses (Scala symbols on the left) to parser-specific labels, as
14 // found in the Lua grammar file (sets of strings on the right).
15 object LuaParseTree extends Domain {
16   override val mapping = Map(
17     'file       -> Set("Chunk"),
18     'block      -> Set("Block"),
19     'statement  -> Set("Stat"),
20     'fork       -> Set("DoStat", "WhileStat", "RepeatStat", "IfStat",
21                        "ForStat", "ForInStat"),
22     'method     -> Set("Function"),
23     'variable   -> Set("Var"),
24     'field      -> Set("Field")
25   )
26 }
27 // Boilerplate wrapping the ANTLR generated parser for use with LISA
28 object AntlrLuaParser extends AntlrParser[LuaParser](LuaParseTree) {
29   override val suffixes = List(".lua") // which blobs to read from Git
30   override def lex(input: ANTLRInputStream) = new LuaLexer(input)
31   override def parse(tokens: CommonTokenStream) = new LuaParser(tokens)
32   override def enter(parser: LuaParser) = parser.chunk()
33 }
34 // This remaining code is not necessary for adding support. It simply
35 // shows how the newly added parser is used to analyze a Lua project.
36 // The UniversalAnalysisSuite contains LISA's object-oriented analyses
37 // which operate on the semantic concepts mapped above.
38 object LuaAnalysis extends App {
39   val url = "https://github.com/Mashape/kong.git"
40   implicit val uid = "Mashape_kong"
41   val gitLocalDir = s"/tmp/lisa/git/$uid"
42   val targetDir = s"/tmp/lisa/results/$uid"
43   val parsers = List[Parser](AntlrLuaParser)
44   val analyses = UniversalAnalysisSuite
45   val persistence = new CSVPersistence(targetDir)
46   val sources = new GitAgent(parsers, url, gitLocalDir)
47   val c = new LisaComputation(sources, analyses, persistence)
48   c.execute
49 }
```

where the metrics are computed. This requires an additional 18 lines of code. See `https://goo.gl/YnNd7j` for an extended version of the Lua support that includes name resolution. Note that this really is only necessary if functions need to be identified in the resulting data set.

Discussion. Whereas the formulation of analyses can be rather verbose, as indicated in Section 3.1, adding support for additional languages is very straightforward and simple in 𝕃ISA. Besides the redundancy removal techniques, the flexibility to easily support additional languages is a major benefit of using 𝕃ISA. That said, in our experience, the performance of ANTLR-generated parsers can vary significantly. Sometimes, it is better to use a native or oth-

erwise optimized parser. Adding an external parser is more work than just copying a grammar file, but 𝕃ISA's interface is straight-forward to implement. For reference, integrating the native Python parser required 134 lines of Scala and 80 lines of Python code. Integrating the JRE's Nashorn JavaScript parser required 168 lines of Scala code. That said, especially for exploratory research in the beginning of a study, having the option of simply dropping in an ANTLR grammar to apply 𝕃ISA is a significant advantage.

3.6 Threats to validity

We presented several experiments that demonstrate the effectiveness of the redundancy removal techniques we describe in Section 2 and thus, the capabilities and usefulness of 𝕃ISA. Our conclusions in these experiments rely on our tooling and the experimentation methodology applied. Even though we designed the experiments diligently, their validity might be threatened for several reasons.

Internal validity. This relates to factors or effects on the experiments that influence the results, but that are not controlled. Ignoring these factors can change the interpretation of the results and the conclusion.

Our findings are based on metrics that we compute using 𝕃ISA for a large number of projects and we rely on the correctness of these computations. We ensure their correctness by running them in a controlled environment and by comparing the results to manually calculated results. Human calculation errors have been mitigated by double-checking of the results.

In the same way, it could be that we are missing an effect that would lead to an incorrect measurement of the reported runtimes for the experiments. However, as the same methodology is used to compare the runtimes of the optimized and the handicapped version of the experiments, such an error would not favor one approach over the other. We ran each experiment once only and not several times (averaging the results), however we are still confident that our measurements provide an accurate representation of how the handicapped versions compare.

The hardware we ran our experiments on is a shared environment, and even though resources are reserved for a process, other jobs running on the same hardware can interfere with 𝕃ISA's performance. Furthermore, due to an existing bug in 𝕃ISA, performance benefits when using more than 28 cores seem to be limited. However, all projects in the large-scale study were analyzed using the same resources.

External validity. The generalizability of our findings to other experimental settings are affected by threats to the external validity of our experiments.

The choice of projects used in our experiments might be biased and not representative for other projects. To mitigate this issue, we selected a large sample of popular open-source projects and we analyze four different programming languages. Our sample covers a wide range of application domains, project

sizes, history sizes, and levels of activity, which makes us confident that our results generalize beyond our immediate findings. However, all selected projects are open-source. While some of them are maintained by professional developers, we cannot assume that the same results will be found in closed-source projects. Furthermore, since we chose projects based on popularity, it may be that less popular projects behave differently or yield different aggregated results. We may replicate our study for additional projects and programming languages in future research to extend the generality of our findings.

All our experiments use the domain of static source-code analyses as a means to evaluate 𝕃ISA and we assume that these analysis tasks are representative for other domains as well. Graphs that are constructed for answering different questions and which are based on artifacts other than plain-text source code (e.g., bytecode or on-save snapshots from an IDE) may have different topologies which could impede the effectiveness of the algorithm and lead to reduced performance. However, the underlying graph framework used by 𝕃ISA, Signal/Collect, does not impose any limitation on the kind of graph that is processed. We do not expect that other kinds of analyses would benefit less from using 𝕃ISA.

4 Discussion

We have presented several techniques and demonstrated their effectiveness. Here, we first reflect on what role a tool such as 𝕃ISA plays in software evolution research by reporting on two artifact studies we performed. We then address how we can further improve 𝕃ISA, and assess its potential for future studies.

4.1 Artifact studies

While the focus of our research is on finding ways to reduce redundancies in multi-revision analyses, thus accelerating them, the ultimate purpose of any software analysis tool is to answer concrete questions. So far, we have used 𝕃ISA in two different artifact studies.

Study A: sampling revisions in software evolution research. In this first study, we asked the question, how many revisions in the history of a project can 'safely' be skipped when doing software evolution research without losing too much information. While 𝕃ISA can perform certain analyses on every revision of a project, other, more resource-intensive tools may not, such that researchers will need to consider a trade-off between the number of revisions effectively analyzed and the time and resources available to them. Likewise, some insights may be entirely unobservable without analyzing every revision. Thus, we wanted to find out how much the temporal data diverges from the 'true', high-resolution history with an increasing number of revisions skipped during an analysis. We published the results of this study in previous work [9], and

have since replicated the study based on the new data we computed for 4,000 instead of 300 projects, also confirming the findings for an additional programming language. We found that for object oriented metrics, the error – in terms of difference of area under the curve (AUC) of the real history compared to the sampled history - can be kept below 1% for 75% of projects by analyzing every 20th revision. We also found that, due to using the AUC difference as an error measure, projects with a large number of revisions can sustain much larger sampling intervals, up to several 100 revisions skipped.

Study B: on the usage of Pythonic idioms. The term 'Pythonic' is frequently used by Python developers on question-answering sites, forums and in live interactions, when referring to the *right, idiomatic way* of writing Python code. While research into this topic is in its infancy, a tentative catalogue of Pythonic idioms exists [71]. We implemented detectors for these idioms as laid out in Section 3.1 and using the resulting data, we mapped out how different idioms were used over different time frames. This research is yet unpublished, but we found that during the first year of development, only 3 out of 15 idioms accounted for more than 90% of idioms used, while over several years, the share of those same 3 idioms drops to just over 30%.

Contribution of our approach toward artifact studies. 𝕃ISA enabled us to compute object-oriented metrics for each and every revision of a large and diverse sample of projects in little over eight days, spending 100ms on a single revision on average. Analyzing the complete source code of over 7 million program revisions using traditional means would have been entirely unfeasible. 𝕃ISA allowed us to perform a controlled experiment where we were able to freely choose how many revisions we include in a study. For the Pythonic study, 𝕃ISA also provided a fast, easy to use alternative to existing approaches. Previous work [71] utilized a text-based (as opposed to AST-based) analyzer that is more complex, significantly slower and, naturally, only works on regular files, i.e., on one revision at a time. Implementing the detectors for Pythonic idioms as outlined in Section 3.1 was straight-forward and allowed us to go from planning the experiment to executing it on 1000 Python projects within two weeks.

That said, we analyzed the historical data computed by 𝕃ISA using other means in both artifact studies, mostly using R. Thus, at this time, 𝕃ISA is primarily a provider of raw data, rather than a tool to directly perform evolutionary studies and derive higher-level results. We discuss how 𝕃ISA could be extended to support such studies directly in Section 4.2.

4.2 Future work

This paper presents a novel strategy for representing and analyzing software artifacts from large repositories with long histories. The results in this paper do not close this line of research and can benefit from several further improvements.

Automated memory trade-off. As we discuss in Section 3.4, it can be difficult to estimate in advance how much memory 𝕃ISA needs for an analysis, because it depends on multiple factors: the number of revisions, the code churn between revisions, the size and number of individual files and the kinds of analyses to be run on the graph. While over-provisioning is a working solution, it would be better if 𝕃ISA could avoid taking on more work than it can handle given available resources, which is feasible given the following approach. After the initial Git metadata extraction, where 𝕃ISA creates the lists of code blobs to be parsed from Git as explained in Section 2.1.2, instead of parsing files from all revisions asynchronously, it could only parse a limited number of revisions. Once it runs low on work (because it is still parsing files which had a lot of changes, but stopped parsing others since they are out of the limited range), 𝕃ISA could check the current memory consumption and make a decision whether or not to extend the limited range further and continue parsing additional files. It could do so up until a pre-specified threshold chosen by the analysts, for example 75% of available memory. The threshold would need to be selected by the analyst because only they know how much additional data is expected to be generated by the analyses to be run. Once the threshold is reached, 𝕃ISA would compute and persist data for the loaded revisions, prune the graph to only hold the very last revision analyzed and then continue parsing the next chunk incrementally using the same strategy, thus keeping a low redundancy overhead.

Facilitating recurrent analyses. In the current implementation of 𝕃ISA, the graph representing the history of analyzed artifacts is transient and only kept in memory. Analysis re-runs need to reconstruct the graph from the data source, for example by parsing the source code once again. However, parsing – the use case presented in this paper – requires a significant amount of time for large projects. We tried several alternative graph databases, including Neo4J and RDF4J[12] as alternatives to Signal/Collect, but their performance, both in terms of memory requirements and computation speed were subpar.

Thus, we still need to find a way to persist and incrementally update 𝕃ISA's graphs such that recurrent analyses do not need to load them from the original data source again. This would either require a rigid entity mapping or disabling the filter on the data source altogether, such that the entire source is represented in the graph (similarly to how BOA stores ASTs [27]). However, storing complete trees significantly inflates memory requirements and computational load: Section 3.3 has shown a seven-fold increase in runtime when analyzing the *Reddit* project without the analysis-specific AST filter. One possible workaround to at least mitigate the computational load would be to add additional edges which could serve as 'shortcuts' or 'fast-lanes' for analysis signals to skip irrelevant nodes where possible, although this would further increase the amount of information needed to be stored in the graph.

Analysis formulation. Analyses in 𝕃ISA need to be formulated using 𝕃ISA's adaptation of the Signal/Collect paradigm, i.e., involving individual vertices

[12] `https://neo4j.com/`, `http://rdf4j.org/`

sending and receiving information. While this paradigm itself is straightforward, there are two opportunities for improvement. First, a lot of boilerplate code is required. Thus, it would make sense to develop a DSL for 𝕃ISA to remove all but the most essential parts of what constitutes a graph analysis. Second, a higher-level query language could simplify many common operations that now need to be implemented as separate Signal/Collect analyses (for example resolving method names contained in child nodes, as described in Section 3.5 regarding Lua). Other graph databases provide such query languages, but as discussed above, their performance was inferior to Signal/Collect.

Direct analysis of evolutionary aspects. Graph analyses, which are formulated with only one revision in mind, are transparently applied to revisions and revision ranges by 𝕃ISA. The computed data is persisted, for example to .csv files or a database, and needs to be analyzed using other means in order to answer specific research questions (e.g., by using R for applying statistical tests to the collected data). It would, however, be possible to add a second way of formulating analyses involving multiple revisions, because all necessary data is present in 𝕃ISA, such that evolutionary aspects could be analyzed directly, instead of having to post-process the data using other means.

In addition to these immediate improvements of the 𝕃ISA framework, we also envision several future applications, which are either enabled through 𝕃ISA or that could highly-benefit from using it.

Different data sources and problem domains. 𝕃ISA's core is completely generic and agnostic to the kinds of multi-revision graphs it stores and analyzes and there are many opportunities for loading other kinds of data into 𝕃ISA. For example, several tools exist that can capture *on-save* snapshots of source code under development directly in the IDE (e.g., [74]). These snapshots contain compiled representations of the project, including fully resolved types for all objects. It would be possible to load such graphs into 𝕃ISA and perform more in-depth analyses (such as call graph-, coupling- and dependency analyses) on these snapshots compared to parsed ASTs. It may also be possible to store and analyze byte- or decompiled code of binary versions of a program where the source is unavailable. Beyond program analysis, 𝕃ISA could be used to analyze multi-revision graphs of arbitrary content, for example evolving social networks or multi-versioned RDF data.

Multi-revision compilation. Typically, source code representations with fully resolved typing information are obtained using a compiler. However, compilers are not typically designed to handle multiple revisions simultaneously. It may, however, be possible to integrate 𝕃ISA's multi-revision approach with an existing compiler. In essence, this would entail replacing all state stored internally during compilation with a multi-revision variant. Just like in 𝕃ISA, code fragments which are unchanged in multiple revision would only be compiled once. It would also be necessary to handle dependencies the same way, since different program revisions may need different versions of the dependencies. The engineering effort would likely be significant, but the result would

be a 'multi-revision' compiler that can generate fully connected source code representations for multiple revisions at once.

Cross-language analyses. We have shown in this paper that 𝕃ISA can analyze projects that use multiple programming languages, i.e., when we computed metrics for both JavaScript and Python code of the *Reddit* project in Section 2. However, in the large-scale study, we only analyzed files for one language per project, because we wanted to identify language-specific recommendations for appropriate sampling intervals. That said, all source code is represented within the same graph in 𝕃ISA and it is possible to create additional edges to connect different components in the graph that come from different languages. For example, an analysis involving JavaScript, HTML and CSS files could search for HTML identifiers (e.g., `"#main-nav-bar"`) in the entire graph and connect nodes parsed from different languages using additional edges. In future work we plan to work on analyses that cross language boundaries.

Implementing additional MSR research techniques. Besides analyzing metadata and computing code metrics, MSR research sometimes involves other kinds of code analysis techniques, for example code clone detection or program slicing. In general, 𝕃ISA could be used to perform any kind of analysis, however the additional development effort to formulate appropriate analyses in Signal/-Collect can be substantial. We previously implemented a cross-language code clone [15] analyzer in 𝕃ISA[13] which hashes sub-trees and connects AST vertices with matching hashes via additional edges in the graph, thus identifying code clones across programming languages. This works for simple examples[14] but further work is needed to determine the effectiveness of this technique in real-world projects. Program slicing [17] refers to the identification of parts of source code that influence a specific variable or return value. Dependency analysis (i.e., tracing the control flow and assignments influencing a variable or statement) is typically used towards this goal. Since 𝕃ISA so far works on uncompiled code, creating control-flow and dependency graphs can be challenging. Implementing these would require creating additional vertices and edges to connect related AST nodes. Points-to analyses [56] represent another, related task which is easier to perform on compiled code.

5 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 relate LISA to existing code analysis tools and also discuss how 𝕃ISA's light-weight mappings differ from traditional metamodels.

[13] `http://goo.gl/aWWCkN`
[14] `http://goo.gl/VftJUW`

5.1 Code analysis frameworks and multi-revision analysis

Static analysis tools like Soot [51] provide building blocks for new static analyses (e.g., points-to analyses). Reusing these building blocks allows researchers to reduce their own effort when writing these analyses. However, compared to 𝕃ISA, such toolkits are not designed to efficiently analyze multiple revisions.

A whole line of research is dedicated to analyzing historic data of software projects. Fischer et al. [30] track revisions and bug reports for a software project in a common database. Information can be requested in simple queries to facilitate the anticipation of future evolution. Bevan et al. [16] introduced the software analysis framework "Kenyon" that provides a flexible infrastructure for common logistical issues, e.g., configuration retrieval or tool invocation, to facilitate software evolution research. Along similar lines, Gall et al. [31, 32] 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, improving upon previous algorithms for extracting changes [23]. Ghezzi et al. presented a service-oriented framework called SOFAS, enabling collaborative analyses of software projects and facilitating the replication of mining studies [34]. Successively, Ghezzi et al. [35] analyzed several studies that mine software repositories and found that less than one third of them are fully replicable, indicating that replicability and generalizability remain important issues in software engineering research. In the same vein, González-Barahona et al. develop a methodology for assessing the replicability of empirical software engineering studies [37]. Zimmermann et al. [92] propose 'ROSE', which analyzes the full history of a project to predict the location of most likely further changes.

Compared to such frameworks, which have a strong focus on conceptual models, e.g., providing a specific query language or involving different artifact types such as bug reports and version control metadata, 𝕃ISA is a much more narrowly targeted (with the primary goal of reducing redundancies in analyzing multi-versioned graphs), but also much more generic tool. It essentially offers recipes and constructs to perform arbitrary graph analyses, while these frameworks usually have a very specific use case within the software evolution research community.

Several approaches exist that try to improve the scalability of static analyses. 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 [27, 28]. 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.

Allamanis and Sutton [10] propose a novel approach, obtained by building a probabilistic language model of source code based on 352 million LOC of Java, which provides a new way for analyzing software projects. In particular, it enables the definition of new complexity metrics based on the statistical model, that allow the detection of reusable utility classes from the program's core logic.

Le et al. [55] present a technique to merge control-flow graphs of multiple versions of a software system. They do not apply this technique for deduplication though, but use it for patch verification across a small number of control flow graphs (CFGs). A general approach could lift this idea to arbitrary artifacts and scale it to hundreds of thousands of revisions. Another example for a similar technique can be found in TypeChef [43], which uses a variability-aware parser to analyze multiple *configurations* of C programs (i.e., `#ifdef`s) in a shared graph. Compared to 𝕃ISA, TypeChef targets type checking and other architecture aspects in a multi-configuration context rather than the generic analysis of artifacts in a multi-revision context.

An indirect way to improve the scalability of static analysis is to use curated datasets that either make it easy to compile programs (e.g., [81]), that already contain fully-qualified typing information [72] (which makes compilation unnecessary), or which consist of pre-computed data as opposed to the original source code (e.g., [7]). Creating and maintaining these datasets involves manual work though, which is why their size is typically limited.

5.2 Multi-language analysis

Given the multi-language nature of today's software projects, researchers have been working on approaches for multi-language program analysis [11, 26, 50]. These approaches are typically built on metamodels [26, 78] that enable the analysis and manipulation of multi-language software code, thus allowing the generalization of specific static analyses and other software recommendations.

Tichelaar et al. [82] proposed the source code metamodel FAMIX to aid the refactoring of source code of different languages. 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 metamodel represents a trade-off between being too *coarse-grained* to be useful for a wide range of problems and being too *fine-grained* to remain sufficiently language-independent. FAMIX has since been used as a metamodel by other tools [34, 52] and later extensions also model changes (i.e., Hismo [36] and Orion [54]), which allow the analysis of source code evolution. Strein et al. have developed a metamodel for capturing multi-language relationships in source code [78] 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 [75]. 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. Heavyweight models similar to FAMIX, such as KDM [4] or ASTM [3] as well as general-purpose models such as RSF [45] or GXL [86] model not only the kinds of nodes, but also their relationships and structure explicitly. The same is true for M^3 [41], which is specifically designed for use with Rascal Metaprogramming Language [40] and which also includes non-code concepts such as physical and logical source locations. In addition, recent work proposed rather different approaches focused on the execution of specific static analyses on multi-language programs such as the detection of software vulnerabilities [38] and changes in software licenses [21]).

All metamodels that we have described in this section have only been applied in single-revision, single-project settings and, contrary to our lightweight approach, do not aim to be practical in large-scale analyses, where performance is crucial. Thus, the biggest difference comparing our lightweight mappings to existing metamodels 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 metamodel. 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 data source, 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 lightweight mappings can be considered a *view* onto an existing graph structure, rather than a metamodel. Furthermore, code models typically come with predefined entity types and relationships, whereas our lightweight mappings are formulated in the context of a particular analysis.

6 Conclusion

Current research on software evolution is limited by the time and effort required to analyze many revisions of a large and diverse sample of projects. No existing tools focus both on analyzing the source code of many revisions as well as different programming languages. To solve this issue, we have presented our open-source tool, the *Lean Language-Independent Software Analyzer* (𝕃ISA), a generic framework for representing and analyzing multi-revisioned software artifacts. It employs a redundancy-free, multi-revision representation for artifacts and avoids re-computation by only analyzing changed artifact fragments at the sub-graph level. It features several distinct redundancy removal techniques that, in combination, facilitate the rapid analysis of artifacts contained in 100,000s of commits. In addition, it allows using lightweight mappings instead of traditional metamodels for static, structural analyses of source code written in different languages. The lightweight mappings not only represent a flexible, analysis-specific bridge between different languages and formats, 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. After formulating a particular analysis, the selection of projects, the creation of language mappings and the automated execution of analyses using 𝕃ISA are straightforward and enable the quick extraction of fine-grained software evolution data from existing artifacts. Our evaluation demonstrates the performance, resource requirements and adaptability of our approach. 𝕃ISA supports varying kinds of code analyses, exhibits high speed, and implements several individual redundancy removal techniques, each of which is highly effective on its own. On average, 𝕃ISA can represent the full history projects using only 4% of the space that would be required to represent all individual revisions. For very large projects, this ratio can be as low as 0.0053%. Consequently, it can analyze a large number of revisions orders of magnitude faster than traditional approaches, on average at around 100ms per revision, even for very large projects. Furthermore, we've shown that the majority of popular projects on GitHub can be analyzed on commodity hardware while describing how larger projects could be analyzed both on high- and low-memory instances. Finally, we've shown how easily 𝕃ISA can apply existing analyses to additional programming languages.

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 version control systems, data sources, storage methods, and analyses.

Acknowledgements We thank the reviewers for their valuable feedback. This research is partially supported by the Swiss National Science Foundation (Projects №149450 – "Whiteboard" and №166275 – "SURF-MobileAppsData") and the Swiss Group for Original and Outside-the-box Software Engineering (CHOOSE).

References

1. Awesome python. `https://github.com/vinta/awesome-python`. Accessed: 2017-06-20.
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. The top 500 sites on the web. `http://web.archive.org/web/20170626103223/http://www.alexa.com/topsites`. Accessed: 2017-06-26.
6. 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.
7. The Promise repository of empirical software engineering data, 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. C. V. Alexandru, S. Panichella, and H. C. Gall. Reducing redundancies in multi-revision code analysis. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, 2017*, 2017.
10. M. Allamanis and C. A. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, 2013*, 2013.
11. 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*, pages 91–95, 2011.
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. G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, pages 1–43, 2014.
14. G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 56–65, 2012.
15. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance*, 1998.
16. J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. pages 177–186, 2005.
17. D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo. Tree-oriented vs. line-oriented observation-based slicing. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30, Sept 2017.
18. C. Bird, N. Nagappan, P. T. Devanbu, H. C. Gall, and B. Murphy. Does distributed development affect software quality? an empirical case study of Windows Vista. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 518–528, 2009.
19. C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 24–35, New York, NY, USA, 2008. ACM.
20. 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.
21. F. Boughanmi. Multi-language and heterogeneously-licensed software analysis. In *2010 17th Working Conference on Reverse Engineering*, pages 293–296, Oct 2010.
22. S. Chacon and B. Straub. Pro Git. Apress, 2014.
23. 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.
24. 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.
25. F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE software*, 2008.
26. L. Deruelle, N. Melab, M. Bouneffa, and H. Basson. Analysis and manipulation of distributed multi-language software code. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 43–54, 2001.
27. R. Dyer. *Bringing Ultra-large-scale Software Repository Mining to the Masses with Boa*. PhD thesis, Ames, IA, USA, 2013. AAI3610634.
28. 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.
29. M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 2012.
30. 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.

31. 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), 2007.
32. H. Gall, B. Fluri, and M. Pinzger. Change analysis with Evolizer and ChangeDistiller. *Software, IEEE*, 26(1):26–33, 2009.
33. 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.
34. 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.
35. 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.
36. T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.
37. J. M. González-Barahona and G. Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1):75–89, Feb 2012.
38. R. Hadjidj, X. Yang, S. Tlili, and M. Debbabi. Model-checking for software vulnerabilities detection with multi-language support. In *2008 Sixth Annual Conference on Privacy, Security and Trust*, pages 133–142, Oct 2008.
39. L. Hernandez and H. Costa. Identifying similarity of software in Apache ecosystem – an exploratory study. In *2015 12th International Conference on Information Technology - New Generations*, pages 397–402, April 2015.
40. M. Hills, P. Klint, and J. J. Vinju. *Program Analysis Scenarios in Rascal*, pages 10–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
41. A. Izmaylova, P. Klint, A. Shahi, and J. J. Vinju. M3: an open model for measuring code artifacts. *CoRR*, abs/1312.1188, 2013.
42. E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 2010.
43. 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.
44. D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 351–360, 2011.
45. 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.
46. 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.
47. 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.
48. 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.
49. E. Kocaguneli, T. Menzies, and J. Keung. On the value of ensemble effort estimation. *Software Engineering, IEEE Transactions on*, 38(6):1403–1416, 2012.
50. 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.
51. P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The Soot framework for Java program analysis: A retrospective. In *Cetus Users and Compiler Infastructure Workshop*, CETUS'11, 2011.

52. 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.
53. M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
54. J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 76(12), 2011.
55. 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.
56. J. Lundberg and W. Löwe. Points-to analysis: A fine-grained evaluation. 18, 12 2012.
57. 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.
58. T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.
59. 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.
60. T. Mens. Introduction and roadmap: History and challenges of software evolution. In *Software Evolution*, pages 1–11. Springer Berlin Heidelberg, 2008.
61. T. Mens, M. Claes, P. Grosjean, and A. Serebrenik. Studying evolving software ecosystems based on ecological models. In *Evolving Software Systems*, pages 297–326. 2014.
62. T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.
63. N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.
64. 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.
65. M. Nagappan, T. Zimmermann, and C. Bird. Representativeness in software engineering research. Technical report, Microsoft Research, 2012.
66. 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.
67. A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. API code recommendation using statistical learning from fine-grained changes. In *International Symposium on Foundations of Software Engineering.* ACM, 2016.
68. H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013.
69. 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.
70. S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 161–170, 2015.
71. J. J. M. Picazo. Analisis y busqueda de idioms procedentes de repositorios escritos en python. Master's thesis, Universidad Rey Juan Carlos, Madrid, Spain, 2016.
72. S. Proksch, S. Amann, S. Nadi, and M. Mezini. A dataset of simplified syntax trees for C#. In *International Conference on Mining Software Repositories.* ACM, 2016.
73. S. Proksch, J. Lerch, and M. Mezini. Intelligent code completion with bayesian networks. *Transactions of Software Engineering and Methodology. ACM*, 2015.
74. S. Proksch, S. Nadi, S. Amann, and M. Mezini. Enriching in-IDE process information with fine-grained source code history. In *International Conference on Software Analysis, Evolution, and Reengineering*, 2017.

75. 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.
76. B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The uniqueness of changes: Characteristics and applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 34–44, May 2015.
77. B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Trans. Software Eng.*, 33(12):800–817, 2007.
78. 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.
79. 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.
80. 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.
81. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, 2010.
82. 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.
83. N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Software Eng.*, 35(3):347–367, 2009.
84. 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.
85. 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.
86. 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.
87. W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of API changes and usages based on Apache and Eclipse ecosystems. *Empirical Software Engineering*, 21(6):2366–2412, 2016.
88. 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.
89. Y. Yu, T. T. Tun, and B. Nuseibeh. Specifying and detecting meaningful changes in programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 273–282, Washington, DC, USA, 2011. IEEE Computer Society.
90. A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.
91. T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.
92. 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.