The Journal of Systems and Software 215 (2024) 112065

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Code search engines for the next generation ☆

Marcus Kessel *, Colin Atkinson

University of Mannheim, 68159 Mannheim, Germany

ARTICLE INFO

Keywords:
Software reuse
Code search
Big data
Big code
Behavior
Dynamic
Semantics

ABSTRACT

Given the abundance of software in open source repositories, code search engines are increasingly turning to "big data" technologies such as natural language processing and machine learning, to deliver more useful search results. However, like the syntax-based approaches traditionally used to analyze and compare code in the first generation of code search engines, big data technologies are essentially static analysis processes. When dynamic properties of software, such as run-time behavior (i.e., semantics) and performance, are among the search criteria, the exclusive use of static algorithms has a significant negative impact on the precision and recall of the search results as well as other key usability factors such as ranking quality. Therefore, to address these weaknesses and provide a more reliable and usable service, the next generation of code search engines needs to complement static code analysis techniques with equally large-scale, dynamic analysis techniques based on its execution and observation. In this paper we describe a new software platform specifically developed to achieve this by simplifying and largely automating the dynamic analysis (i.e., observation) of code at a large scale. We show how this platform can combine dynamically observed properties of code modules with static properties to improve the quality and usability of code search results.

1. Introduction

There are many different forms of software reuse, with many different goals, but one of the most canonical forms is the incorporation of pre-existing, self-contained software components in new applications to reduce the amount of code that has to be rewritten from scratch (Mili et al., 1995, 1998). The key enabling platform for this form of reuse is the *Code Search Engine* (CSE), inspired by mainstream Internet search engines. While the goal of mainstream search engines is to find relevant Internet artifacts, CSEs focus on retrieving relevant code artifacts represented in a formal programming language.

General-purpose Internet search engines only started to become useful and attract serious interest once the number of available Internet artifacts reached a certain critical mass about the turn of the millennium.[1] In the same way, CSEs only started to become truly useful and attract significant research interest (about 10 years later) when the number of code artifacts retrievable over the Internet reached a critical mass thanks to the rise of the open source movement (Lerner and Tirole, 2001) and supporting repositories (Hummel and Atkinson, 2006). Since then, the number of papers on CSEs has expanded rapidly, with over 80% of the 100 or so papers published on CSEs having been written since 2008 (Grazia and Pradel, 2022).

While the main problem faced by the first generation of CSEs was populating their databases with meaningful numbers of code artifacts, the current generation of CSEs has the opposite problem — coping with the vast, and rapidly exploding, number of code artifacts stored in modern Internet repositories. For example, more than 85 million new projects were created on GitHub in 2022 alone (GitHub, 2022), and this is only one of several large software hosting sites. This explosion in reusable source code means that many newly-written software components today are similar, or identical to, code that already exists (Yang et al., 1999; Inoue et al., 2020; Rahman et al., 2018). Managing and efficiently leveraging this huge number of open source code artifacts is a natural *big data* problem. Indeed, most of the current research into CSEs aims to find novel ways of applying recent results from data science to improve their performance and utility. This makes a lot of sense because source code is in essence text and is thus inherently amenable to data science techniques. Natural-Language Processing (NLP) and semi-structured data analysis techniques are particularly applicable, since they can be used to analyze and exploit the meaning of the identifiers (i.e., names) chosen by programmers. Given that programmers usually try to pick names for identifiers collectively, in systematic ways, the *naturalness hypothesis* (Allamanis et al., 2018) also assumes that code even exhibits some of the patterns and features of natural

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: marcus.kessel@uni-mannheim.de (M. Kessel), colin.atkinson@uni-mannheim.de (C. Atkinson).

[1] The original paper by the Google founders was published in 1998 (Brin and Page, 1998).

https://doi.org/10.1016/j.jss.2024.112065
Received 6 December 2022; Received in revised form 28 February 2024; Accepted 12 April 2024
Available online 6 May 2024

languages documents, which makes it even more amenable to NLP pattern matching techniques.

Unlike purely natural language documents, however, software artifacts have two additional properties that make their comprehensive analysis especially challenging. The first is that they are represented in languages with formally defined grammars, which allows them to be parsed into concrete, well-defined data structures such as abstract syntactic trees (ASTs). This makes it possible to analyze and compare the syntactic structure of code artifacts using a range of formal and algorithmic methods such as feature vectors, graph and solver-based matching algorithms as well as data science techniques such as machine learning (e.g., classification). Since the formal structural properties of software are so important, dedicated platforms have been developed to support the syntax-based analysis of code at "big data" scales, such as Boa (for the ultra-large scale analysis of ASTs) (Dyer et al., 2015), and SourcererCC (for detecting code clones at a large scale) (Sajnani et al., 2016).

The second fundamental property of software artifacts that makes their comprehensive analysis especially challenging is their dynamic behavior (a.k.a., semantics) when executed on a suitable computing platform. Full and precise analysis of a software's dynamic (i.e., run-time) behavior lies beyond the reach of the logico-deductive and data science based approaches used on syntactic structures due to Rice's theorem. This is a fundamental theorem of computer science which states that all non-trivial semantic properties of programs, including their behavioral semantics, are undecidable, which means that no general algorithm for deciding whether a software component truly implements a particular piece of functionality can exist (Rice, 1953). Statistical techniques such as NLP are not subject to Rice's theorem, of course, but rely on programmers using accurate and explanatory identifiers in their code. Since programmers can just as easily choose inaccurate or misleading identifiers as good ones, they are notoriously unreliable at predicting behavior (cf. vocabulary mismatch problem Furnas et al., 1987). Furthermore, the resulting lack of a reliable link between the claimed semantics of code (as implied by its identifiers) and its *true* run-time semantics has a direct, negative impact on the precision and recall of traditional CSEs, since they rely on these static analysis techniques. It reduces the precision of code searches because it increases the number of false positives in the result set (i.e., it includes code components whose identifiers falsely imply they have the desired semantics), and it reduces the recall of code searches because it decreases the number of true positives in the result set (i.e., it fails to include code components whose identifiers falsely imply that they do not have the desired semantics).

Since the only practical way of gaining insights into the true semantics (i.e., run-time behavior) of code components is to *run* them, several early generation CSEs such as Sourcerer (CodeGenie) (Lazzarini Lemos et al., 2007; Bajracharya et al., 2014), S6 (Reiss, 2009) and Merobase (CodeConjurer) (Hummel, 2008; Hummel et al., 2008), attempted to address this weakness by including dynamic (i.e., execution-based) observation of behavior into the software search process. Technically, this applies an approach that was first referred to as behavioral sampling (Podgurski and Pierce, 1992), but CSEs that employ it are usually referred to as "test-driven" or "test-case driven" CSEs (Sim and Gallardo-Valencia, 2015; Robillard et al., 2014). In this paper we use the term "test-driven search engines" and the acronym TDS engine. Although these early test-driven search engines were able to demonstrate some limited success in increasing code search precision, they suffered from several major weaknesses —

1. *Small corpora*: Since the curation of corpora of executable software systems was (and still largely is) performed by hand (e.g., Dietrich et al. (2017), Palsberg and Lopes (2018), Martins et al. (2018a) and Fraser and Arcuri (2014)), the number of software artifacts available to CSEs was very small by today's standards. Automating as many of the corpus curation tasks as possible, including build automation, is key to obtaining large-scale collections of executable systems from today's online software repositories.
2. *Poor scalability and performance*: Since early TDS engines were invariably built on the monolithic computing platforms available at the time, they took a long time to generate search results. Efficiently executing software systems at an ultra-large scale requires fully-blown clustering technology supporting the vertical and horizontal scaling of workloads that takes advantage of today's big data technologies and hardware.
3. *Declarative query languages*: The only practical way to realize test-driven code searches on a large body of code is to first perform a "standard" text-based search and then apply the defined tests to each of the returned candidate code artifacts. The approach is therefore inherently complex and involves multiple steps. The first generation of CSEs hid this complexity by only supporting searches defined in declarative query languages. To request a search, users had to define the desired properties of the sought after software artifacts, but had little if any opportunity to influence how the search results were produced.
4. *Platform-specific test description language*: Since tests represent one of the inputs to the test-driven code search process, the way they are defined has a significant impact on the usability of CSEs. The first generation of CSEs only accepted tests written in mainstream, code-based test definition technologies like JUnit (JUnit, 2022), which can be extremely verbose, and are difficult to combine with the text elements of the core search query. They also tangle the test inputs (i.e, stimuli) with the expected outputs (i.e., software responses).
5. *Limited adaptation technology*: Since it is rare for two distinct implementations of a given piece of functionality to use exactly the same interface (e.g., in terms of method and class names, parameter types and orders etc.), many code components that actually deliver the desired functionality will be missed unless an appropriate *adapter* is created to adapt them to the interface expected by the tests. The first generation of TDS engines had very primitive adaptation capabilities.
6. *Simplistic Ranking Approaches*: The ranking criteria used to order search results is a key factor in the perceived usefulness of all search engines, since research shows that users rarely look beyond the first 10 returned results (Wang et al., 2009; Joachims et al., 2017). However, because they do not execute the code, the current generation of CSEs are unable to consider anything other than statically-deduced metrics when comparing the retrieved candidates for ranking (e.g., vectors comparison etc.). This means that the true values of some of the most potentially interesting metrics (e.g., true behavior, execution speed, resource usage etc.) cannot be used as ranking criteria.
7. *Lack of dynamic metrics about the result set*: The perceived usability of search engines is also influenced by the amount of information they provide about the candidates in the result set, enabling users to manually analyze the top ranked candidates. Again, CSEs that do not actually execute the code components in the result can only create statically-inferred estimates of dynamic properties of the components rather than measure their true values.

This paper is based on the premise that to significantly improve the usability, precision and recall of code search services, the next generation of CSEs needs to address these weaknesses. Moreover, they need to do so at a scale and efficiency commensurate with the static analysis techniques supported by data science and platforms like Boa. In this paper, we show how this is possible by using a new platform designed to support the large-scale observation and analysis of software. This platform, known as the *La*rge-*S*cale *S*oftware *O*bservatorium, or LASSO for short (Kessel, 2023), was not specifically designed as a

search engine but rather as a generic software observatorium. However, because efficient code retrieval, systematic testing and workflow customization on a large scale are fundamental requirements for such a software observatorium, it provides advanced solutions to all the aforementioned problems.

The goal of this paper is to demonstrate how a software observatorium like LASSO can support a new generation of code search use cases and significantly increase the quality of the results. It does this in four main ways —

1. it shows how the current functionality offered by LASSO significantly enhances the range, power and customizability of code search use cases that can be supported (Section 3),
2. it explains how LASSO's implementation of these services, together with LASSO's underlying corpus of executable software, provides novel solutions to the weaknesses defined above (Section 4),
3. it demonstrates how LASSO's scripting language and functionality can be used to concisely, efficiently and scalably implement a "classic" TDS engine as a user-facing service, which we refer to as LASSO TDS (Section 5),
4. it presents a study showing how LASSO TDS, and thus the technology it is built on, significantly increases code search recall and precision relative to first-generation TDS engines (Section 6).

LASSO TDS is included for two reasons. First, it represents the state-of-the-art in "classic" CSEs, where users initiate searches by means of declarative query languages, which is still a highly sought after service. Second, it provides a functionality reference point against which the advantages of the large-scale observation technology offered by LASSO can be gauged. Although the first generation TDS engines are no longer publicly available, the principles by which they were realized are well known and can serve as a baseline for comparison.

The LASSO platform has been described before in previous publications. The most comprehensive and detailed description is the PhD dissertation of the first author of the paper (Kessel, 2023). This also includes an introduction to LASSO's search service and an overview of its realization. Other papers that have used LASSO in test-driven software experiments (Kessel and Atkinson, 2024) or the implementation of other new services have also provided high-level overviews of its features and implementations (e.g., Kessel and Atkinson (2019b,a) and Kessel and Atkinson (2022)). The unique features of the current paper are a comprehensive overview of LASSO from the perspective of reuse-oriented code search, a more detailed description of how it overcomes the seven weaknesses above, and a new study based on 20 real-world examples that presents concrete evidence of the benefits these advances bring to the state-of-the-art in TDS engines.

The rest of the paper is structured as follows. The next section describes the context to the work, discusses the history and background of CSEs, and discusses related work. Section 3 then presents *LASSO* and its various CSE-related capabilities from the perspective of users (i.e., a black box perspective), including an overview of the languages and data structures that the platform offers. Section 4 continues with a white-box perspective of *LASSO*, providing a detailed description of the underlying structures, services and data sources that support the previously described capabilities. Section 5 then turn to *LASSO TDS*, describing both its user-facing services and GUI as well as its implementation using *LASSO*'s scripting language and data structures. With *LASSO TDS* having been presented, Section 6 presents a study that shows how the realization strategy and underlying technology used to build the service provides clear benefits over traditional TDS engine implementations. Section 7 presents a discussion of the insights that can be gained from the presented experiment, before Section 8 concludes the paper with a summary of the presented work, an appraisal of its potential significance and some remarks about how it could be built on in the future.

2. Background

In order to provide the context for the rest of the paper and explain the contributions the LASSO platform can make to code search and software reuse, in this section we describe relevant background material and related work. Another goal is to establish consistent terminology. Note that a recent survey on techniques for searching for code can be found in Grazia and Pradel (2022).

2.1. Code search and software recommendation

In general, any technique or activity that exploits existing software in the development of new applications can be regarded as a form of software reuse (Krueger, 1992; Frakes and Kyo Kang, 2005). This ranges from the use of exemplary snippets of software from forums such as Stack Overflow (Abdalkareem et al., 2017) and the mining of successful patterns from repositories, to the testing of tools and hypotheses using large software data sets (Dyer et al., 2015). However, the canonical form of reuse is the incorporation of pre-existing, self-contained software components within new applications to reduce the amount of code that has to be rewritten from scratch. A lot of research was done on this form of reuse in the 1980s and 1990s (Mili et al., 1995, 1998), but its impact was limited by the small software data sets that could be obtained at that time. The field received a new lease of life in mid 2000, when open source software repositories started to become accessible on the Internet and efficient, full-text search tools such as Lucene (The Apache Software Foundation, 2022c) became available to index and analyze their contents as textual documents. These advances gave rise to a range of tools, referred to as code search engines (CSEs) or code recommendation systems (CRSs), which aimed to tackle the core problem of implementation-saving reuse — finding existing software entities that match the needs of a new application or use case.

CSEs (Sim and Gallardo-Valencia, 2015) and CRSs (Robillard et al., 2014) often go hand in hand and basically only differ in the way users interact with the search technology. CSEs essentially require users to perform a proactive search by creating some kind of explicit *query*, whereas code recommendation systems typically do not require users to perform explicit code searches, but *suggest* potentially useful reuse candidates to them by observing their current development work. CRSs usually depend on CSEs to carry out searches over large populations of code units, but generate the queries automatically based on the code the software engineer appears to be developing. Well known examples of such dependencies include the CodeGenie recommendation tool (Lemos et al., 2007) driven by the Sourcerer (Bajracharya et al., 2014) search engine and the CodeConjurer (Hummel et al., 2008) recommendation tool driven by the Merobase (Hummel, 2008) code search engine. Since this paper is focused primarily on the code search side of the technology, we do not mention recommendation tools in the rest of the paper.

Interface-driven code search (IDS)

Since code is a form of semi-structured, text-based data, the majority of dedicated CSEs is essentially based on full-text search. However, the query languages of most CSEs assign a special meaning to the core components of source code such as methods, classes etc. This can extend to the level of allowing users to specify the interfaces of the software abstractions they are looking for in terms of one or more method signatures (Zaremski and Wing, 1995; Hummel, 2008; De Paula et al., 2016). The goal of such interface-driven queries is to find all code models that implement the specified interface. Since this can range from a single method to a large collection of classes and other source code components, the term *software component* is often used as an all embracing term for the entities returned. We also use this term in this paper. We regard a software component as any self-contained collection of components that collectively *appear to* implement the desired interface. For clarity, we refer to the interface and logical behavior of the software component being sought in a search, as a *functional abstraction* and any particular collection of components that realizes that interface and behavior as an *implementation* of that functional abstraction.

Test-driven code search (TDS)

The term "appear to" in the previous paragraph is important, since it reveals one of the main weaknesses of contemporary CSEs — the need to infer the semantics (i.e., run-time behavior) of software components from the identifiers used to name classes, methods and parameters, and if available, the comments that accompany the formal code statements. Although these inference techniques have become quite sophisticated through the use of advanced NLP techniques like word stemming, word embeddings (e.g., code2vec Alon et al., 2019), query expansion (Nie et al., 2016) and topic modeling coupled with AI-based language models and neural networks (Gu et al., 2018), they can never fully overcome the idiosyncratic choices of software engineers when selecting identifiers. As mentioned previously, the problem of establishing whether two software components implement the same functional abstraction (i.e., the same behavior) is formally undecidable (cf. Rice's theorem (Rice, 1953)) so that no general purpose algorithmic solution is possible.

The only practical way of removing the influence of identifier choices in assessing the functional equivalence of software systems is therefore to compare their responses to a sample set of stimuli from their input spaces. This approach, first proposed under the name of *behavior sampling* in the early 90s (Podgurski and Pierce, 1992), has been shown to be effective provided that the set of stimuli is of sufficient size and quality (Podgurski and Pierce, 1993; Kessel and Atkinson, 2019b). In an effort to support true semantic searches alongside text-based searches, several CSEs such as CodeGenie (Lemos et al., 2007), S6 (Reiss, 2009), Hunter (Wang et al., 2016) and Merobase (Hummel, 2008) therefore incorporated some form of behavior sampling capability under the name of test-driven or test-case-driven search. In particular, S6 and Merobase seamlessly added semantic (i.e., test-driven) searches on top of interface-driven searches. For the rest of this paper, we use the term *test-driven search* to refer to behavioral-sampling that enhances traditional search processes with the sampling of input/output mappings. The precise way this sampling is performed can vary, however. For example, Satsy (Stolee et al., 2014) is a CSE that samples input/output mappings by so-called static execution — that is, static program analysis in terms of symbolic execution and constraint solving. Even though this approach is promising, the current limitations of constraint solving restrict its applicability to tiny code snippets. Moreover, no measurable engineering goals based on run-time characteristics can be supported by this technique.

Code-driven code search (CDS)

Instead of requiring a classic interface-driven search query and a set of test cases to define a test-driven search, it is possible for a CSE to perform a test-driven search with only a single implementation of the desired functionality as input. In such a code-driven search (or code-to-code search) scenario, the CSE first has to extract the desired interface from the input code and then automatically generate a set of tests with which to execute retrieved candidates. Although this use case may at first seem counter-intuitive, because by definition an implementation of the desired functionality already has to exist, it may be a low quality or untrustworthy one. Reusing mature, tried-and-tested open source implementations of the same functionality may therefore have advantages. Obviously, in this CDS scenario, the input implementation is used as the oracle to determine the functional equivalence of candidate alternative implementations when executed. The FaCoY search engine was developed to support precisely this CDS use case (Kim et al., 2018), but uses a static analysis approach of the kind mentioned above, and so is limited by the intractability of establishing functional equivalence statically. However, observation-based TDS technology can be extended to support code-driven searches as long as effective test inputs can be generated automatically. The main limitation in classic, execution-based implementations of CDS, therefore, is the quality of the automatically generated tests.

2.2. Clone detection

A field of software engineering research that overlaps with, and is relevant to, CSE research is code clone detection (Roy et al., 2009). Clone detection research is interested in detecting many more forms of similarity between code components, and does not necessarily require them to be functionally equivalent. Code clones are usually classified into four different categories (Koschke, 2007) depending on what kinds of similarity the code components exhibit —

- type-1: textual similarity (the code components are identical, except for non-meaningful elements such as white spaces, carriage returns and comments),
- type-2: lexical similarity (the code components have exactly the same structure, except for systematically adjusted identifier names and literal values),
- type-3: syntactic similarity (the code components are quite similar syntactically, but differ in some way at the statement level),
- type-4: semantic similarity, the code components have similar semantics, but may differ syntactically (e.g., the code components may be implemented completely differently but have similar semantics in terms of run-time behavior).

This terminology, however, does not clearly define what clones are and what the employed similarity measures look like. Several authors have therefore proposed additional subtypes of clones such as exact clones for type-1, renamed and parameterized clones for type-2, near-miss clones for type-3 and semantic clones for type-4 (Svajlenko and Roy, 2016).

Although the code clone research field has different concrete goals to the CSE field, its results are very relevant. In particular, CSEs usually employ type-1 and type-2 clone detection algorithms to improve the diversity of their results. Software engineers are usually interested in "different" implementations of the functionality they are searching for rather than implementations that differ only in some trivial way such as type 1 and 2 clones. TDS engines could therefore be characterized as semantic clone detection engines, but we avoid this terminology in this paper, since it seems counter-intuitive to refer to two algorithmically different implementations of a functional abstraction as clones. Instead, we only refer to type-1 and type-2 clones, according to the aforementioned taxonomy, as clones.

2.3. Large-scale software analysis platforms

Ensuring that the components returned in a code search implement the sought after functionality is obviously a critical evaluation criterion for practitioners. Nonetheless, numerous other factors can impact the relevance of search outcomes. These range from straightforward, size-oriented code metrics like cyclomatic complexity (e.g., McCabe (1976)) to more intricate measures such as the similarity of code duplicates. As these metrics can be fairly hard to measure, a CSE must be able to execute advanced analytical algorithms on a massive scale, akin to the realm of big data. Furthermore, users should be empowered to specify new relevance criteria by describing novel analytical algorithms in an abstract manner, without the need for significant manual coding effort. The current generation of CSEs only provides limited support for high-level relevance criteria and does not give users the ability to define new metrics in an abstract fashion. However, platforms in the emerging field of large-scale software analysis, such as the Boa platform described in Dyer et al. (2015), focus on delivering this capability.

The central objective of the Boa platform is to assemble an extensive, ultra-large repository of software components and make them analyzable in an abstract (i.e., syntax-based) manner using a dedicated, high-level, domain-specific language. While Boa was not explicitly developed to support interface-based and test-driven code search, it does allow for keyword-based queries based on the abstract syntax of code

elements. Furthermore, as demonstrated by QualBoa in Diamantopoulos et al. (2016), the platform's analytical capabilities can be harnessed to formulate advanced, reuse-oriented relevance ranking schemas.

Similarly, SourcererCC, an extension of the Sourcerer platform described in Sajnani et al. (2016), facilitates syntactic code clone detection on a massive scale. Nevertheless, a notable drawback of these large-scale analysis engines is their inability to accommodate dynamic (i.e., execution-based) algorithms and metrics.

3. LASSO — User perspective

This section provides a new user-oriented (i.e., black box) overview of the *La*rge-*S*cale *S*oftware *O*bservatorium (*LASSO*) which has been specifically designed to address the limitations discussed in Section 1. Further details about the *LASSO* platform can be found in Kessel (2023). LASSO offers a wide range of (automated) general-purpose analysis services to support two primary objectives: (a) the development of novel, or enhanced, solutions to specific software engineering challenges, such as diversity-driven test generation (Kessel and Atkinson, 2019, 2022), and automated curation of executable *live data sets* (Kessel and Atkinson, 2019a), and (b) the performance of behavior-based, software engineering experiments (Kessel and Atkinson, 2024), such as studies on functional equivalence in behavior sampling (Kessel and Atkinson, 2019b). What sets *LASSO* apart from other (ultra) large-scale software analysis platforms like Boa (Dyer et al., 2013, 2015) is its unification of dynamic analysis services with static, syntax-based services at an (ultra) large scale.

LASSO's ability to select and compare software through large-scale behavior sampling means that it has the basic mechanisms needed for classic TDS. However, the languages and data structures supported by the platform can also support a much richer set of code search use cases geared toward software component reuse. In this regard, the three main features of LASSO are —

- *Executable Corpus*: reuse candidates can be harvested from LASSO's single, underlying, executable corpus that offers a systematic repository model to incorporate a variety of data sources,
- *Component Processing Pipelines*: customized code search strategies supporting rich combinations of reuse criteria can be defined as pipeline scripts that serve as executable code search descriptions as well as reusable templates to analyze and measure reuse candidates,
- *Software Analytics*: LASSO's specialized data structures can be used to support offline software analytics that can guide the decision-making process in reuse (i.e., to apply and enforce reuse criteria).

In all search use cases, a search engine's job is to find and return concrete items of a particular kind that the user (usually) does not yet have, but is able to imagine and describe. In the case of mainstream Internet search engines, when users are searching for information about a certain topic, they imagine the perfect document with exactly the right information they are looking for, and provide "a brief description" of that document in the form of a textual query. The search engine then finds and returns concrete documents that match the imagined perfect document, ranking them according to how well they match. In the case of a CSE, when users are searching for a certain "concrete piece of software", they imagine the "perfect piece of software" that delivers exactly the right behavior and provide a "brief description" in a certain form. The CSE then finds and returns "concrete pieces of software" that match the imagined "perfect piece of software", ranking them according to how well they match.

In order to provide intuitive ways of referring to these concepts, LASSO uses the following terminology. A "concrete piece of software" is referred to as a "software component", or just component for short, because the vast majority of code searches focus on parts of a system, not fully fledged systems in their own right. In Java, which is the language currently supported by LASSO, components are composed of one or more interfaces and/or classes. The imagined "perfect piece of software", on the other hand, is referred to as a "functional abstraction" (loosely similar to a "coding problem" Chen et al., 2021) in order to emphasize that it is abstract and not necessarily incarnated in a concrete form. Since LASSO aims to be aware of true behavior as well as syntactic form, conceptually a functional abstraction is not only characterized by its name and its interface (i.e., method signatures), as is the case in syntax-based CSEs, but also by its "behavior". Conceptually, the behavior of a functional abstraction is characterized by all possible "*actuations*" (i.e., stimulus–response pairs) through which the functional abstraction can be invoked. Finally, the "brief description" in the context of a behavior-aware CSE has to encompass a syntactic characterization of the functional abstraction's interface as well as a semantic characterization of its behavior – that is, a "behavioral sample" in the form of a set of test cases.

Using this terminology, the job of a CSE can be characterized as returning a set of software components that "implement" the functional abstraction sought after by the user, described as a syntactic interface specification and a set of test cases. A software component is said to "implement" a functional abstraction if its behavior (i.e., set of actuations) subsumes the behavior (i.e., set of actuations) of the functional abstraction. In practice, LASSO supports functional abstractions that range from a single (often stateless) method, as in coding problems that require utility/auxiliary functionality (Lazzarini Lemos et al., 2009) (e.g., a method sorting an array), to stateful abstractions (e.g., a queue data structure realized as a class comprising methods characteristic for its behavior).

3.1. Executable software corpus

The creation and maintenance (i.e., curation) of executable software corpora has traditionally been performed by hand (Allamanis et al., 2018; Palsberg and Lopes, 2018) which is tedious and extremely time-consuming. This has historically been the "Achilles' heel" of behavior sampling approaches, since it impedes their practical application at a large scale. To reduce this impediment, therefore, it is necessary to automate the creation of executable software corpora to the greatest extent possible. Ideally, an executable corpus should contain a large number of diverse, non-trivial, up-to-date, real-world software components to boost reuse opportunities (Do et al., 2005; Wohlin et al., 2012; Terra et al., 2013; Barr et al., 2015; Dietrich et al., 2017; Martins et al., 2018a).

LASSO addresses this problem by taking advantage of the modern build automation ecosystem provided by Maven (The Apache Software Foundation, 2022b) which is widely used by industry practitioners. More specifically, it automatically synthesizes build scripts for software projects to increase the likelihood that the software components in a corpus will be executable.

The corpus can be constructed from a variety of data sources. LASSO's current corpus has been constructed from a vast assortment of Java software components obtained from Maven Central (Sonatype, 2022) and various other prominent software engineering repositories such as SF110 (Fraser and Arcuri, 2014). The underlying repository model allows for the integration of additional repositories, including commercial ones, to leverage reuse opportunities in as many contexts as possible.

3.2. Component processing pipelines

LASSO gives users control of search pipelines by enabling them to write scripts to access, manipulate and analyze software components from the aforementioned executable corpus. This is achieved using the *LASSO Scripting Language (LSL)*. LSL allows adaptable search strategies and reuse criteria to be defined as component pipelines which

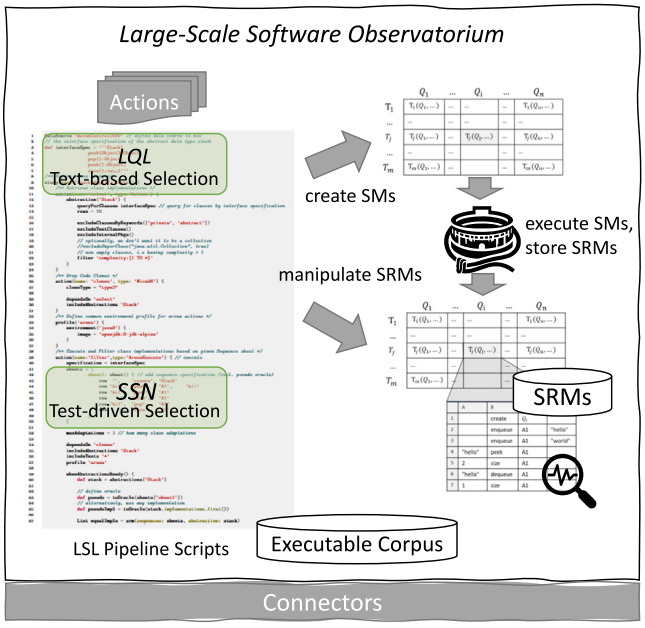


Fig. 1. LASSO - High-level overview of concepts.

serve as templates that can be reused. Using LSL, users can create comprehensive analysis pipelines that incorporate multiple actions to effectively analyze software components extracted from LASSO's corpus as demonstrated in Fig. 1. Typical pipeline elements involve actions to —

1. retrieve software component candidates that exhibit certain syntactic characteristics from the corpus (**LASSO Query Language**),
2. specify tests to stimulate the selected component candidates using a specially designed language (**Sequence Sheet Notation**),
3. execute the specified tests on the selected component candidates in an "*arena*", and store the resulting actuations in a specially tailored data structure (**Stimulus Response Matrices**),
4. enrich the data structure with additional software metrics for filtering and analysis purposes (**Software Measurement and Analysis**),
5. process the collected data in a multidimensional data structure using mainstream data analytics platforms (**Software Analytics**).

We give an overview of each of these elements in the subsequent subsections.

3.2.1. LASSO Query Language (LQL)

To be efficient, TDS approaches usually retrieve an initial set of component candidates using a low precision, but fast algorithm to keep the number of components to be tested manageable. To this end, LASSO supports the NLP-based, "full-text" selection of software components through keyword, phrase and filter queries. Basic keyword searches and filtering techniques are applied in this preliminary step to retrieve software components from LASSO's corpus that appear to have the desired behavior based on their static properties such as identifier names and comments.

An advanced form of full-text search, which also takes the interfaces of functional abstractions into account, is IDS (cf. Section 2.1). This exploits the fact that functional abstractions with similar functional behavior often have similar interfaces (De Paula et al., 2016; Kessel and Atkinson, 2019b). In LASSO, interface-based queries are defined using the LASSO Query Language, LQL, which offers a simple, UML-like syntax inspired by Hummel's MQL (Hummel, 2008) to describe the interface of a functional abstraction by its name and method signatures, including input and output parameters. For example, in LQL, a query for a method implementing the Base64 url-safe encoding abstraction[2] would have the form —

```
Base64 {
    encodeUrlSafe(byte[])->String
}
```

Likewise, the interface of a FIFO queue data abstraction may be specified as follows —

```
Queue {
    enqueue(Object)->boolean
    dequeue()->Object
    peek()->Object
    size()->int
}
```

Interface descriptions in LQL can be augmented with additional filters to limit the number of software components of interest. These filters cover a range of diverse filtering criteria including metrics and API-related criteria (e.g., a queue has to implement the `java.util.Collection` interface).

3.2.2. Sequence sheets

There are two key steps involved in the dynamic analysis of software components based on their run-time behavior — creating executable descriptions of how to stimulate (i.e., test) the components and recording how the components respond to these stimuli. Together, these constitute recorded actuations of the components.

As previously discussed in Section 1, current approaches for test definition often struggle to effectively link these two aspects. Many test definition methods rely on annotating code with *assertions* that specify the expected return values of software components at specific points in the code (e.g., unit testing frameworks such as JUnit (2022)). However, because test methods can leverage the full flexibility of structured programming, the exact sequence of component method invocations during testing is often unknown until run-time. Furthermore, the actual responses of the components to the stimuli are typically not recorded unless an assertion fails and an informative feedback message or report is generated, indicating a discrepancy between the expected and actual response. Testers can usually only obtain information about the actual responses of the system under test through manual debugging activities, for example using an IDE's debugger or monitoring code execution.

These approaches, therefore, significantly limit the analyzability of the behavior of software components in reuse tasks. Moreover, given the frequent "many-objective" goals of practitioners (e.g., code quality attributes and functional sufficiency Kessel and Atkinson, 2016), they often miss interesting software component candidates. In general, having full actuation records available allows practitioners to make more informed decisions.

To tackle this issue, LASSO introduces a unified approach for specifying sequences of stimuli for software components and documenting their corresponding responses, in the form of a new *Sequence Sheet Notation (SSN)*. Unlike fully-fledged programming languages such as Java, SSN does not include control flow constructs such as loops or conditional statements, which means that it is not Turing-complete. However, this limitation is typically not an issue when writing tests (cf. Ammann and Offutt (2016)), as each potential (reachable) test path written in a Turing-complete programming language like Java can be expressed as an individual uni-path test in SSN. This approach offers the benefit that the precise method invocation sequences are known in

[2] A binary-to-text encoding schema (Josefsson, 2006).

Functional Abstraction
name: Queue (FIFO)
Interface: *Queue {*
enqueue(Object)->boolean
dequeue()->Object
peek()->Object
size()->int }

Stimulus Sequence Sheet
testQueueElements(p1:Queue)

	A	B	C	D
1		create	?p1	
2		enqueue	A1	1
3		enqueue	A1	2
4		peek	A1	
5		size	A1	
6		dequeue	A1	
7		size	A1	

Body

Actuation Sequence Sheet
testQueueElements(QueueImp)

	A	B	C	D
1		create	QueueImp	
2	TRUE	enqueue	A1	1
3	TRUE	enqueue	A1	2
4	1	peek	A1	
5	2	size	A1	
6	1	dequeue	A1	
7	1	size	A1	

Fig. 2. Example stimulus and actuation sequence sheets for *Queue* abstraction.

advance, and allows for the recording of responses alongside the stimuli that triggered them.

Like normal methods in object-oriented programming languages, sequence sheets have two essential parts, and can thus be viewed from two perspectives: (1) the *body* (or white-box perspective) contains the individual invocations of the subject software component(s) that occur when the sheet is executed, and (2) the *signature* (or black-box perspective) comprises the test sheet's name and any required parameters or returned values that are passed when it is executed.

There are two types of sequence sheets: *stimulus sequence sheets* and *actuation sequence sheets*. The difference is that stimulus sequence sheets only define the invocations to be made on the software component under test when the sheet is executed (i.e., they specify a sequence of statements ready for execution), while actuation sequence sheets provide full information about the stimuli and responses. Actuation sheets therefore augment the invocation details in a stimulus sheets with response records.

Fig. 2 shows an example of both kinds of sheets, displayed in spreadsheet form, for the FIFO *Queue* example. The left-hand side shows a stimulus sequence sheet that can be applied to a *Queue*, identified as a formal input parameter. The right-hand side shows a corresponding actuation sequence sheet which is generated by executing the stimulus sheet on a concrete *Queue* which is passed as the actual parameter of an invocation of the sequence sheet. The signature of the actuation sheet shows that, in this case, the concrete implementation is *QueueImp*.

In greater detail, the top left-hand side of the figure identifies the name of the abstraction, *Queue*, as well its signature represented in LQL notation. The bottom left-hand side of the figure shows the signature and body of the parameterized stimulus sequence sheet for exercising a queue's methods. The signature gives the name of the sheet, `testQueueElements`, and indicates that it receives one parameter, $p1$ which is a *Queue* (i.e., the component under test).

The rows of a sequence sheet represent invocations of methods of the component under test, including the input parameters. Actuation sheets also show the executed component's response (i.e., output parameters). One of the columns of a sequence sheet identifies the name of the method that is called in each invocation, in the case of Fig. 2 this is column *B*. The columns to the right of column *B* (i.e., *C* and *D*) contain the input parameters to each invocation, while the column to the left of *B* (i.e., *A*) shows the output parameters or responses from the invoked component. In the case of the stimulus sheet on the left, there are no output values shown, while in the case of the actuation sheet on the right, the actual responses from the executed queue class implementation are shown. In general, there is no limit to the number of output parameters a sequence sheet can handle. However, when dealing with software components written in mainstream object-oriented languages like Java, only one output column is required.[3]

The input parameter of the first invocation in a sequence sheet serves to pass the component under test to the sequence sheet, as indicated by the cell reference in the sequence sheet (i.e., *A*1). The `create` method is a special "pseudo" method whose execution generates an instance of the component under test. This abstract approach is employed to initialize the subject component, since obtaining an instance of a class in the conventional manner (e.g., using the `new` keyword in Java) is not always feasible, especially when dealing with a large set of diverse components that necessitate the creation of adapters to overcome interface mismatches (see Section 4.2.3).

Note that method invocations in a sequence sheet are not restricted solely to the component(s) being tested. They can also include invocations to methods of other objects. This allows for more intricate test scenarios, such as those involving complex test data providers and helper functionality, which are commonly used in unit testing practices.

3.2.3. Stimulus Responses Matrices (SRMs)

Sequence sheets are primarily designed for describing individual tests conducted on individual software components. To extend this capability to encompass the description of multiple tests on multiple components (i.e., to automate the mass testing of software component candidates), LASSO introduces a new data structure known as the *Stimulus Response Matrix (SRM)*. Essentially, an SRM contains multiple actuation sheets, each corresponding to the invocation of multiple stimulation sheets targeting various implementations of the functional abstraction under investigation. In typical code search scenarios, it is assumed that all tests (i.e., stimulation sheets) and all implementations adhere to, or support, the same functional abstraction. As illustrated in Fig. 3, the columns within the SRM represent the different components being tested, while the rows represent the different tests.

This illustration highlights the fact that SRMs can be viewed from two perspectives – a black box perspective and a white box perspective. Each cell, row T_j and column Q_j for instance, corresponds to the

[3] Note that some other popular programming languages like Python do support multiple output parameters.

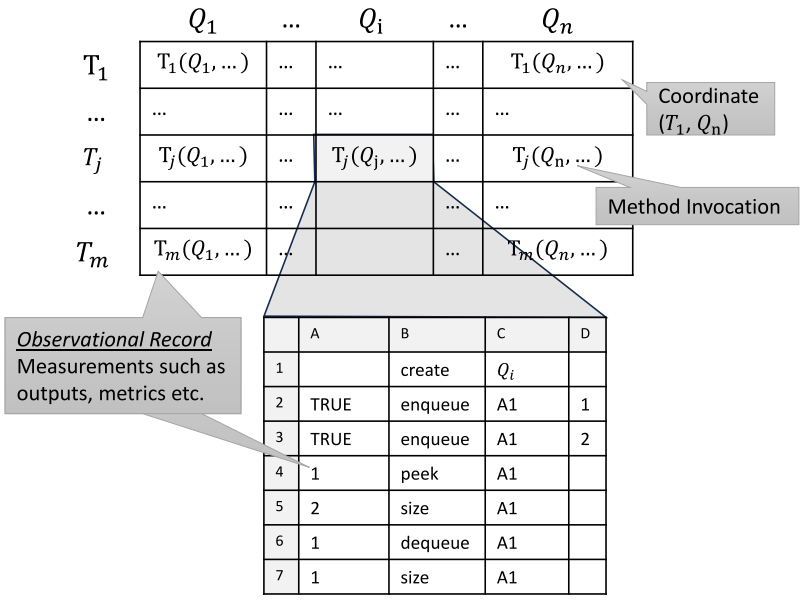


Fig. 3. A stimulus/response matrix (SRM).

application of the stimulation sheet for test instance T_j to the component implementation for column Q_j. In the black box perspective, only the signatures of each invocation are displayed, whereas in the white box perspective the whole actuation sheet is shown. An SRM thus offers an organized and cohesive data structure for archiving and navigating all stimuli, responses, and measurements related to multiple tests of multiple component implementations, enabling comparisons of their behaviors. Furthermore, this structure can also accommodate additional types of data (i.e., observational records) beyond those pertaining to functional behavior (e.g., dynamic metrics).

The distinction between stimulus and actuation sequence sheets is also reflected in the two types of matrices associated with these sheets – *Stimulation Matrices (SMs)* for stimulus sequence sheets, and *Stimulus Response Matrices* for actuation sequence sheets. In essence, as illustrated in Fig. 3, SRMs are systematically organized collections of actuation sheets, and thus may also be referred to as *actuation matrices*. Similarly, *Stimulation Matrices (SMs)* are systematically organized ensembles of stimulus sheets that define what stimulations must be applied to the alternative component implementations.

The so-called *arena* is the actual test-driver that populates SMs in LASSO (see Fig. 1) and applies the stimulations to the implementations to obtain an SRM. Users can then explore and navigate SRMs to analyze and compare the observed behaviors of the components.

3.2.4. LASSO Scripting Language (LSL)

The glue that allows all the aforementioned ingredients to be simply and effectively used together is the *LASSO Scripting Language (LSL)*. LSL is a domain-specific extension of the Groovy programming language (The Apache Software Foundation, 2022d) running on the Java Virtual Machine, and allows users to have a unified view of an entire data collection and analysis pipeline in LASSO. This is achieved by means of reusable and composable *actions* that represent high-level, abstract steps in an analysis pipeline. As well as the basic data structures supported by Groovy, LSL actions can create and manipulate the set of data structures which include SMs, SRMs and sequence sheets described in the previous subsections.

Given that LSL analysis pipelines draw inspiration from the dataflow programming paradigm (Johnston et al., 2004), an LSL script illustrates the progression of data (i.e., data is held by functional abstraction data containers) as a sequence of actions. In our context, data moving from one action to the next is typically represented in the format of SRMs. Each action functions as an analytical step capable of manipulating the data in the flow, but creates data that cannot be altered by subsequent actions (i.e., immutable). Copies of this data are provided to other actions, allowing pipelines to safely pause and resume their processing without interference from previous or concurrent tasks. Additionally, this enables future scripts to access and utilize the generated information at each step.

Although there is no strict classification system for action types within LSL, they generally fall into two categories: (1) those that focus on creating stimulation matrices for input to the arena, usually involving code retrieval and filtering steps, and (2) actions that are concerned with generating, augmenting and analyzing SRMs. Actions can be combined and nested in any way, and may either create or consume data structures.The *Records* data abstraction provides an abstract representation of the kinds of information stored by LASSO when an SRM is generated, such as responses, execution traces, dynamic metrics etc. As can be seen in the diagram, an SRM contains multiple *Records*, one associated with each cell. The *Environment* and *Scope* data structures essentially store information about the conditions under which the software components are executed (i.e., execution profile) and criteria used to determine what code is part of the code units of the component and what code is external for the purpose of calculating metrics.

Listing 1 shows a simple TDS example of an LSL script that defines a pipeline for selecting and analyzing implementations of the *Queue* abstraction introduced above. Approximately the first two thirds of the script (up to and including Line 33) are concerned with preparing the ingredients for submission to the arena as an SM (through the text-based selection of components in the *select* action and the definition of the stimulus sheet in the *filter* action), while the final third is responsible for executing the SM and conducting a script-driven analysis of the results stored in the output SRM returned by the arena.

The first line of the script identifies the executable corpus from which the candidates should be retrieved (here a recent snapshot of the Maven Central repository). The second line then uses LQL to define the text-based search query for an interface-driven code search of the kind defined previously. The next line then declares a study block that represents the pipeline of analysis steps in terms of actions. To begin, the first action *select* (starting at Line 8) conducts an interface-driven

```
1  dataSource 'mavenCentral2023'
2  def interfaceSpec = '''Queue {
3                          enqueue(Object)->boolean
4                          enqueue()->Object
5                          peek()->Object
6                          size()->int }'''
7  study(name:'Queue-TDS') {
8      action(name:'select', type:'Select') {
9         abstraction('Queue') { // interface-driven code search
10             queryForClasses interfaceSpec
11             rows = 10
12             excludeClassesByKeywords(['private', 'abstract'])
13             excludeTestClasses()
14             excludeInternalPkgs()
15         }
16     }
17     action(name:'filter',type:'ArenaExecute') {
18         sequences = [ // stimulus sheet incl. oracle values
19                'testQueueElements': sheet(p1: 'Queue', p2: 1,
                    ↪ p3: 2) {
20                    row '',     'create', '?p1'
21                    row true,   'enqueue', 'A1', '?p2'
22                    row true,   'enqueue', 'A1', '?p3'
23                    row '?p2', 'peek',    'A1'
24                    row 2,      'size',    'A1'
25                    row '?p2', 'dequeue', 'A1'
26                    row 1,      'size',    'A1' }]
27         features = ['cc'] // additional code coverage
            ↪ measurements

28         dependsOn 'select'
29         includeAbstractions 'Queue'
30         profile('jdk17profile') {
31             scope('class') { type = 'class' }
32             environment('jdk17') { image =
                ↪ 'maven:3.6.3-openjdk-17' }
33         }

34         whenAbstractionsReady() {
35             def queue = abstractions['Queue']
36             def expectedBehaviour = toOracle(srm(abstraction:
                ↪ queue).sequences)
37             // returns a filtered SRM
38             def matchesSrm = srm(abstraction: queue)
39                     .systems // select all systems
40                     .equalTo(expectedBehaviour) //
                        ↪ functionally equivalent
41             // continue pipeline with matched systems only
42             queue.systems = matchesSrm.systems
43         }
44     }
45 }
```

Listing 1: Simple LSL pipeline script - test-driven selection for *Queue* implementations

Table 1
LASSO's Maven Central corpus statistics (snapshot January 2023).

Unit	Total	Unique
Artifacts	184,464	184,464
Compilation units	8,884,430	6,947,672
Classes (non-abstract)	6,682,724	5,281,170
Classes (abstract)	531,732	397,691
Constructors	10,700,527	4,064,347
Methods (non-abstract)	75,335,199	28,916,079

code search to return at most 10 candidates (here Java classes), based on the given *Queue* interface specification in LQL notation, and defines a number of filters to exclude undesired candidates (i.e., candidates with undesired visibility restrictions or origin).

The candidates returned are then stored and passed to the second action "filter" (starting at Line 17) which `dependsOn` on the former action and registers for the data structure created in the former action. This action actually defines a new SM based on the definition of a (parameterized) stimulus sheet (similar to the one introduced before, but two distinct values are passed as parameters `?p2` and `?p3`). Note that stimulus sheets can contain oracle information (i.e., expected outputs) in the response columns (here the first column).

The SM is then passed to the arena to be processed. This is configured in Line 30 to have the desired execution profile. Users have the possibility to define the actual sandbox environment including the desired run-time version (i.e., execution profile, here Java's JVM is set to OpenJDK 17). This is realized using modern containerization technology. Since the arena also supports additional measurements such as obtaining dynamic metrics, scopes for measurements can be defined as well. Finally, the output SRM from the arena (starting at Line 35) is analyzed using script-driven analysis to establish functional equivalence with the given oracle information in the first column of the stimulus sheet.

3.3. Advanced software analytics

In general, LSL pipelines of the kind discussed above describe the actions performed at execution (i.e., observation) time, while the dynamic and static data is being collected. LSL provides a range of actions that can be used to analyze the data online during the pipeline's execution, which we refer to as script-driven analysis.

The key difference to existing test definition approaches, however, is that SRMs populated by the arena not only contain all the records necessary to conduct complex comparisons of component behaviors, they can also be serialized and analyzed in a sophisticated data-driven manner. This opens up the possibility of shifting from the traditional *online*, run-time analysis and comparison of software components performed by unit testing frameworks to *offline* analysis and comparison processes. This allows for more flexible decision-making in the reuse process, since judgments about functional equivalence or similarity can be postponed to a later phase (even outside the platform's context). Existing test-driven code search engines, in contrast, typically apply strict matching criteria (i.e., all tests need to pass) and have a non-transparent matching process which gives users no feedback about their responses and hence no control over the applied matching criteria.

LASSO allows the recorded SRM data to be exported in popular data formats like data frames (i.e., tabular representations) for deeper, *offline* analysis using the full power of mainstream data analytics platforms. Since observational records are stored centrally in a database, they persist the results obtained by LSL script executions. Users can then connect to the database in the range of ways commonly available today (e.g., JDBC access), or export CSV files and other common storage formats like Parquet.

4. LASSO - Architecture and implementation perspective

This section delves deeper into how the concepts and features introduced in the previous section are implemented with a focus on how they facilitate search and analysis capabilities. It therefore provides a white-box perspective on the LASSO platform with a new focus on how it implements the aforementioned LASSO TDS service.

Fig. 1 gives an overview of the platform's distributed architecture and its core components. The platform was developed using Java and utilizes Apache Ignite's clustering solution (The Apache Software Foundation, 2022a) to provide a distributed database management system and computations for high-performance computing.

4.1. Curating a corpus of executable software

LASSO is designed to facilitate the mass retrieval, execution and observation of Java software components. The classes and interfaces comprising the components are obtained from various data sources and integrated into the executable corpus through a systematic curation process. This process is supported by a practical repository layout and storage model inspired by the Maven ecosystem. Internally, the platform utilizes Maven's "Project Object Model" (known as POM) to

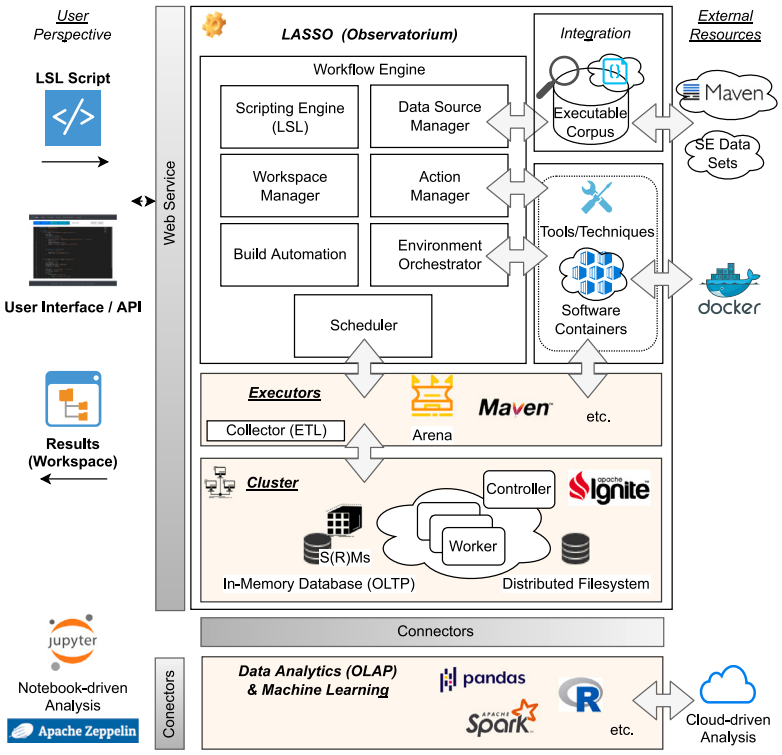


Fig. 4. LASSO platform overview — distributed architecture (Kessel, 2023).

generate build scripts for the acquired Java compilation units, with the goal of rendering them executable. This approach allows Java projects containing diverse types and structures to be transformed into the platform's supported format. Additionally, when new data sources are incorporated into the executable corpus, the Java modules they contain undergo static analysis to construct a searchable, full-text index of the code components. This index facilitates text-based code searches, including interface-driven searches, using Solr/Lucene (The Apache Software Foundation, 2022c).

Table 1 presents some basic statistics about the primary source of executable data for LASSO which is Maven Central.

At present, this corpus encompasses a total of 184,464 indexed Maven artifacts, 8,884,430 indexed Java compilation units (stored as class documents), and 75,335,199 indexed Java methods (specifically, non-abstract methods stored as method documents). These statistics distinguish between the overall count and the number of unique entities stored in the index. The uniqueness criterion is determined by a straightforward comparison of string hashes (MD5 hashes) of class and method bodies. The (physical) size of the index of this data source is about 440 GB. It is important to note that in large repositories like Maven Central, identical code clones are common occurrences (cf. Lopes et al. (2017)). They often result from historical copy-and-paste reuse (Sim and Gallardo-Valencia, 2015), the creation of comprehensive artifacts that integrate third-party dependencies, or multiple releases and variations.

4.2. Interface-driven code search

An essential part of any TDS technology is the pre-retrieval of a preliminary set of implementation candidates that appear to exhibit the desired behavior, otherwise the execution overhead would be unmanageable (i.e., testing the entire corpus in the worst case). This can be achieved through various NLP-based textual retrieval methods, but the most effective method is to retrieve candidates based on their exposed interface(s) using IDS. However, the effectiveness of IDS searches is often limited due to signature mismatches caused by variations in naming choices made by developers or differences in the position and types used for parameters. This issue, partially related to the vocabulary mismatch problem (Furnas et al., 1987), can lead to low recall. To address this challenge and improve signature matches, LASSO employs several optimization techniques such as fuzzy matching, word similarity and type expansions techniques. These strategies help to identify similar signatures despite variations in naming conventions or parameter types used by developers.

As explained in the previous section, an interface signature in LQL comprises identifier and type information. Therefore, IDS utilizes one or both of these elements for matching purposes. The primary objective is to prioritize matches that meet both identifier and type criteria first, followed by further relaxing the restrictions on identifiers and types. This is based on the assumption that components closely matching the identifiers are more likely to deliver the desired functionality. In contrast, pure behavior sampling methods do not rely on specific identifiers but rather focus on type information. As such, they remain unbiased by exact identifiers and only consider types. This essentially leads to a reformulation of the interface mismatch problem to a prioritization (or optimization) problem where the candidates that "appear to" exhibit the desired behavior are placed earlier in the result list, and hence are tested first in behavior sampling (i.e., test execution) step.

We address this problem by applying the following optimization techniques to prioritize components with interface signatures that appear to closely match the desired behavior —

- tokenizing identifiers and types (i.e., word stemming etc.),
- maintaining different combinations of signature representations (cf. Hummel (2008)),
- expanding signature identifiers with similar identifiers,
- expanding signature type information based on type hierarchy and relaxation.

The first two optimization techniques can only be applied when the index is first created, but the second two can be applied both at index creation time and during the execution of a specific search (i.e., at query time). The last optimization technique seeks to (a) establish compatible types using type hierarchy analysis, and (b) if feasible, convert types to compatible ones (for instance, changing INT to LONG).

4.2.1. Interface representation

The initial step involves tokenizing the interface signatures obtained from components during the indexing process and from functional abstractions (i.e., interfaces represented in LQL) during query processing. This tokenization process includes breaking down identifiers and types in accordance with the standard naming conventions of the programming language, which involve using upper and lower camel case for class and method names, respectively.

There are two reasons for keeping various variations of signature representations. The first is rooted in the concept of addressing common signature mismatches using a set of rules or guidelines. These rules encompass scenarios related to parameter order, how type names are represented (whether fully-qualified like `java.lang.String` or simpler like `String`), and also include representations with differing levels of detail (for example, excluding identifiers). The second reason is to provide a range of query options and capabilities to design a fallback strategy to further improve recall.

4.2.2. Query formulation and expansion

Recall that LQL is used to query the corpus' index in an IDS. LQL is based on an ANTLR4 grammar that is used to parse the interface signature into a tree structure which is then used for translation into appropriate SOLR subqueries. Several subqueries are constructed which target different variants of the signature as defined by the representations discussed before. Basically, all those signature-related subqueries are weighted according to their importance, from complete and exact, to incomplete and similar, and are combined in order to form a fallback strategy that aims to improve the recall of components.

We apply two additional query expansion techniques (Carpineto and Romano, 2012) to further improve recall in IDSs by augmenting the constructed query with additional subqueries of lower importance (i.e., lower weight). Firstly, we expand the search space of method identifiers by taking advantage of word embeddings, an NLP technique that represents words as number vectors and allows their semantic and syntactic properties to be identified including their co-occurrence to measure word similarity. We take advantage of word embeddings by querying the CODE2VEC model trained by Alon et al. (2019) for method names. Note that in this case we obtain similar words for an identifier based on the entire method identifier (no splitting).

Secondly, we allow for the expansion of signature types by including any compatible or convertible types found in their hierarchical (object-oriented) structure (i.e., super types) or through a predetermined list of convertible types (e.g., `String` to `byte[]`). Overall, with regard to query construction, the objective is to exactly match signature types first, then to relax the matching strictness while allowing for some flexibility with identifier matching.

4.2.3. Adapter synthesis

Using the previously described method, the goal is to maximize the inclusion of components by loosening the criteria for matching interface signatures. It is crucial to emphasize that this approach identifies components whose interface signatures are theoretically compatible but require additional work to make them callable by the stimulus sheets that assume the concrete interface signature of the functional abstraction being sought. To make the initial set of textually retrieved components callable by the sequence sheets, it is necessary to synthesize adapters that effectively delegate the stimuli to the actual interface of the component being tested.

Tackling the problem of interface mismatches through automated adapter synthesis is a challenging endeavor. The run-time adaptation of components is performed in the arena test-driver, which executes SMs and produces SRMs, using Java's meta-programming reflection feature (Li et al., 2019; Lilis and Savidis, 2019). The adaptation mechanism employed in the arena follows a similar idea to the aforementioned technique for retrieving components textually. Since the adaptation problem essentially suffers from a combinatorial explosion of its search space, it has to be treated as an optimization problem. The search space of possible adaptations is mostly influenced by the number of methods possessed by a component, their input parameters as well as their type compatibility. When attempting to match a certain configuration of methods to a functional abstraction, many different permutations of methods and their parameters often need to be tried to find a perfect match. Early approaches such as the brute-force approach used in MEROBASE (Hummel, 2008) are highly inefficient, since they simply test all combinations until a match is found.

We apply a more systematic and goal-oriented strategy to prioritize combinations of methods that are likely to yield a match. The prioritization scheme is based on the idea of improving adaptation performance by selecting method permutations to execute using the following information —

- type hierarchy of each method parameter (input/output) by walking up the inheritance hierarchy of a type to find assignable types up to the root type,
- primitive (wrapper) type casting and relaxation,
- switching parameter type positions,
- looking up inherited methods of matched classes (overridden methods in subtypes are favored),
- ranking method permutations based on a prioritization schema that weights matches based on closeness.

Our priority scheme for prioritizing method permutations is based on type closeness, location of the matched method in the inheritance hierarchy of the current class and, optionally, naming conventions. We use a loose weighting approach to compute the ranks of method permutations similar to that used by Wang et al. (2016). The adaptation technique employed also offers the possibility of defining new adaptation operators which are basically categorized into producer and method operators. While producer operators have the responsibility of obtaining an instance of a Java class (e.g., constructors or factory methods), method operators realize more advanced strategies to adapt and invoke methods (e.g., converting types like string into byte array). For space reasons, in this paper it is not possible to provide a complete list of the operators provided by the platform. A complete list can be found in Kessel (2023).

4.3. Workflow engine

The core of the platform is the workflow engine which oversees the management of script pipelines and the delegation of tasks to other platform components (see Fig. 4). The scripting engine plays a crucial role in parsing and interpreting LSL scripts, while the workspace manager handles the provisioning and loading of work spaces for script executions. The data source manager serves as both the point of integration for new data sources for the executable corpus and the manager for data sources configured in LSL script executions.

Similarly, the action manager serves as a repository for the available reusable actions and manages their status. These actions are designed to interface with external tools and research techniques, allowing them to be employed within LSL scripts for specific analyses and comparisons. Consequently, users and integrators have the ability to develop and share custom actions tailored to their specific tools and techniques.

4.3.1. Script execution

LSL scripts are executed in a distributed manner. When an LSL script is submitted to LASSO, the manager node employs a partitioning strategy with the objective of establishing a distributed execution plan. This plan assigns actions and selected components to a group of available worker nodes within the cluster, based on directed acyclic graphs (DAGs) which are extracted from LSL pipelines. Currently, the default partitioning strategy generates blocks of components in a round-robin fashion, and these buckets are subsequently allocated to worker nodes. Each worker node then carries out the entire LSL action on one of these blocks of components. The manager node ensures consistency by consolidating the results and state once all worker nodes have completed their actions. This is achieved through shared state management via an in-memory data grid and a distributed file system.

4.3.2. Arena execution and observation

Software components are primarily executed in the arena component of the platform. The primary function of the arena is to create instances of the stimulus sheets from the input SMs on the components and then run them in order to produce SRMs. It handles all the heavy lifting by retrieving all the necessary (Maven) artifacts for the components, loading required classes using a dedicated, isolated (Java) "Classloader" that can be instrumented for additional analyses (e.g., dynamic metrics such as code coverage measurement), and eventually attempting to execute the adapted variants of the components (cf. Section 4.2.3).

Once all tests have been executed on all computed adapters (i.e., based on a certain setting), the results are directly written into the distributed database of the platform. The observational records obtained (i.e., observed responses and additional measurement records) can be analyzed further either in a script-driven or data-driven, offline way.

The execution of SMs is split into manageable batches that are distributed among computing nodes in the cluster, allowing both vertical and horizontal scaling. Each computing node runs an instance of the arena test-driver, enabling multi-threading to run multiple components in parallel.

4.3.3. Sandboxing and extensibility

Running foreign software components from large repositories on local machines can be risky and may have undesired consequences, including malicious behavior (e.g., Fraser and Arcuri (2014)). To prevent this, LASSO isolates the execution of actions and components within secure sandbox environments which are realized using DOCKER containerization. These not only provide fine-grained control over resource allocation and permissions, but also allow users to specify their target (reuse) execution environments (e.g., specific Java version).

Containerization also serves as an extension point for the platform. It enables third parties to create custom environments that contain certain tools and techniques to be used for specialized kinds of analysis. More generally, since LASSO is designed as a general-purpose platform that allows the creation of new and customized analysis services, new tools and techniques can be integrated via its action model (i.e. using the LASSO Action API). Note that the platform already integrates a couple of actions to offer various code analysis options as demonstrated by LASSO's TDS service in the next section.

5. LASSO TDS

As discussed above, the biggest user feature distinguishing LASSO from first generation CSEs is LSL which allows users to (a) quickly and concisely write their own code search algorithms (and algorithms for many other applications as well), and (b) customize basically every aspect of the search process, including the (data) source of the components, the clone removal criteria, the syntactic matching criteria and the behavioral (a.k.a., semantic) matching criteria. From this perspective, we believe that LASSO can be regarded as a second generation CSE (among other things).

However, for users who want to customize the details of their search processes, learning how to use LSL and write effective workflow scripts is a non-trivial task. There is therefore still a big call for "traditional" query-based search services that allow users to define the functional abstractions they are looking for in a much simpler, declarative way. Although LASSO incorporates all traditional CSE capabilities as LSL actions for the purpose of defining new services, these can only be invoked using LSL. Therefore, in this section we show how LSL can be used to efficiently implement a "classic" test-driven search service, with a corresponding "classic" web interface, but with the extra precision, recall and performance afforded by LASSO's technology. This service, which we refer to as LASSO TDS, has three main purposes —

- to demonstrate the flexibility and power of LSL as a language for defining new services on top of LASSO's built-in features,
- to create an online, state-of-the-art TDS service that not only goes beyond the capabilities of the first generation of TDS engines, but also beyond the TDS capabilities currently built into LASSO. This is because it includes a sophisticated way of "relaxing" the matching criteria to increase recall, as explained below.
- it allows the advantages of LASSO to be compared to a known "comparison point" for CSEs which is the traditional implementation of classic TDS services. Although, at the time of writing, all the first generation TDS engines have been deprecated and are no longer accessible, their fundamental implementation strategies are well known and can be used for comparison. LASSO TDS therefore provides the basis for the study in Section 6.

5.1. Strict test-driven search

Listing 2 (Appendix) demonstrates how LSL can be used as a dynamic query language to design a strict TDS service using the queue abstraction as a running example. While this pipeline shares similarities with Listing 1 (i.e., using IDS for text-based selection and behavior-based filtering based on functional equivalence to make the final selection), it also includes two additional actions to filter out component candidates based on the presence of code duplicates and to rank candidates according to their relevance to user-supplied preferences.

First, a set of component candidates is textually retrieved using IDS (cf. Section 4.2). The first block demonstrates an additional filter that rejects candidates that have a cyclomatic complexity less than 2. The next action, `rejectClones`, then rejects type-2 code clones using the Nicad code clone detection tool that is integrated into the platform (Cordy and Roy, 2011). The rejection of clones serves to increase the diversity of the set of component candidates returned by IDS, and thus demonstrates how such a custom reuse criterion frequently desired by users can be explicitly enforced in LSL. The state after each action is also saved to further increase the overall transparency of the search process (i.e., to explore which candidates were rejected).

As explained earlier, the arena test filter goes one step further by allowing users to specify stimulus sheets to improve the precision (i.e., relevance) of the candidate set based on their exhibited behaviors. This version of the TDS service is referred to as "strict" because if a response provided by a candidate does not match the expected response (i.e., output values), according to Java type matching rules, it is rejected. Each analysis step in the pipeline may produce aggregated measures (i.e., observational records as part of SRMs) that can be used to rank the accepted components based on user preferences. An ideal ranking places the components that best match the requirements at the top. Ranking in this example is demonstrated by the integration of the SOCORA approach (Kessel and Atkinson, 2016) which provides non-dominated sorting based on multiple criteria and can be used as an alternative to simple LSL actions that involve imperative sorting on certain attributes. Note that by default candidates are sorted by their IDS relevance.

5.2. Relaxed test-driven search

While classic TDS processes attain a high level of precision by utilizing test filtering (which means rejecting all candidates that do not return the expected outputs), their strict rejection criteria usually reduce recall. To further improve recall over the previously described strict variant of LASSO TDS as well as LASSO's built-in TDS capabilities accessible via LSL, LASSO TDS includes an advanced behavior-based matching algorithm. This uses LASSO's actuation analysis capabilities to generalize the matching criteria, thereby increasing recall without compromising precision.

The concept of relaxed TDS is rooted in the idea that, beyond the candidates identified by the strict matching criteria of strict TDS, there are also variations of expected outputs that users would probably view as functionally equivalent. In a strict TDS engine, users are generally required to assume that there is only one correct set of outputs, making it nearly impossible to consider alternatives. Consider, for example, how a method signals errors when given an invalid input value, such as a `null` reference. Over time, developers have devised various methods for handling invalid inputs. For example, one developer might return a negative integer to indicate an error (e.g., −1), while others might return a `null` value or raise an exception of a specific type. If a user expects only one of these three as the correct output, although the remaining outputs still make sense semantically, the user is likely to overlook interesting software components that only differ in their approach to error handling.

The queue abstraction example provides another illustration of the potential advantages of broadening the matching criteria. In real-world scenarios, there are certain variations in how (Java) queue implementations handle their return values, and these variations may slightly differ from what a reuser might expect in terms of both method signatures and expected outputs. For instance, some queue implementations may return nothing (i.e., `void` type) when the `enqueue` or `dequeue` method is invoked, while others may return a `boolean` type or the actual enqueued or dequeued element. If the reuser is willing to expand the matching criteria to encompass outputs that do not alter the overall desired behavior, a wider range of matching components can be returned. Other examples include the specific formatting of return string values (e.g., new line or space characters may be irrelevant to the reuser), or additional features that were unexpected (e.g., the representation of a hash value in some encoding scheme). Consequently, tailoring the behavioral matching criteria to include more components can potentially provide increased recall while still maintaining precision.

The specific matching criteria are naturally determined by each individual reuser (i.e., the arbiter), and as such, they may vary from one situation to another. Overall, since the reuser is the arbiter in this case, the range of possible generalizations (i.e., allowed outputs) may be huge depending on the underlying functional abstraction.

SRMs empower reusers to create tailored matching criteria by enabling them to cluster components according to their equivalent behaviors (in terms of equivalent outputs). These clusterings can be exploited systematically to automatically recommend clusters of components with behaviors closely aligned to those defined by the reuser. Alternatively, the reuser can explore these clusters to gain a deeper understanding of the different recorded behaviors related to the desired behavior.

As previously noted, it is worth highlighting the fact that the determination of functional equivalence among software components can also be achieved offline, independently of LASSO TDS, using some external data analysis tool that examines the exported SRMs and the observational records (i.e., outputs) they contain.

5.3. Templating and frontends

As mentioned previously, once stable, LSL pipelines can be used to realize a variety of packaged code search services that can be consumed by other applications or search frontends. One way of achieving this is to prepare script templates that are instantiated with a subset of query information collected from the user (similar to popular templating engines or code templating). This script templating capability is used to deliver LASSO TDS in the form of two user-facing applications — an IDE plugin for IntelliJ (see Kessel (2023) for details) and a web-based UI that provides the classic CSE interface and offers several views to hide the details of the pipeline construction process as illustrated in Figs. 5(a) and 5(b).

Fig. 5(a) illustrates how a classic TDS query is defined using the web user interface of LASSO TDS (displayed on the right-hand side). In this interface, users are presented with (1) an editor for specifying the interface of a functional abstraction, such as the queue abstraction, using LQL, and (2) an editor for creating one or more stimulus sheets (i.e., for test-driven filtering). Additionally, users can configure various settings, including extra filters, certain actions and general search and ranking parameters (e.g., rows, etc.). To provide a visual example, the generated LSL script pipeline is displayed on the right-hand side (which is hidden from the user in the UI).

Fig. 5(b) presents a sample excerpt of the results obtained from a TDS for a *Queue* abstraction. Each row in the result provides additional details about the matched component, including its actual interface signature, source code, observational records from the SRM, and supplementary reports generated by the platform as part of the actions executed (e.g., statistics).

6. Demonstration and evaluation

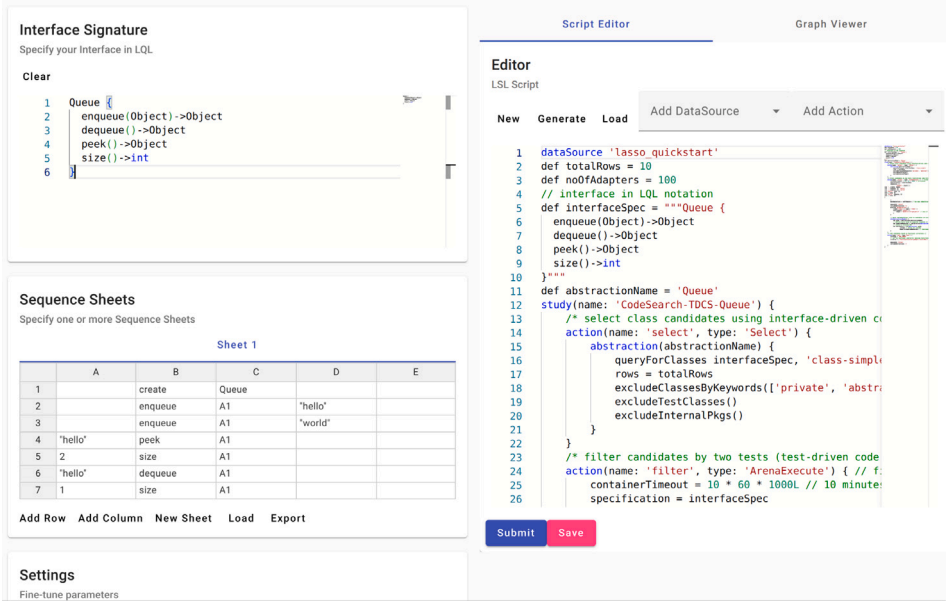
Having introduced the concepts and approach behind LASSO, and LASSO TDS in particular, in this section we provide initial experimental evidence of the effectiveness of LASSO TDS as a *strict* (i.e., traditional) as well as a *relaxed* TDS engine to support code search use cases. The data and materials used in this study are publicly available at Kessel and Atkinson (2023).

It is important to stress that it is not possible to perform a full benchmarking experiment that compares the performance of our approach to existing approaches for two reasons. The first is that there are currently no other TDS engines to compare our prototype to. As mentioned previously, the first generation of TDS engines were primarily developed in the late 2000s in research projects and at the time of writing have all been discontinued. It is therefore not possible to make any quantitative statements about our prototype's performance relative to other technologies in terms of size, speed, precision and recall, since there are no other test-driven CSEs currently available for execution.

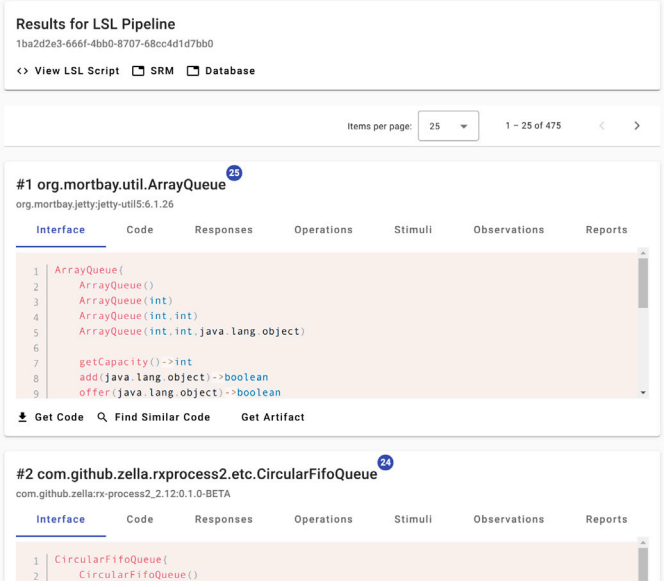
The second reason is that a comparison of the precision and recall quality metrics with standard, static, analysis-based approaches would require a common baseline or benchmark. In other words, it would require the underlying repositories of the search engines to be fully incorporated into LASSO's repository to allow our algorithms to be applied to the same search space.

At present, achieving this goal is made difficult by a mix of conceptual and practical obstacles. One such obstacle is that the repositories of other CSEs are seldom shared with importability in mind, or thoroughly described, especially in terms of achieving code executability. Typically, these repositories do not prioritize or even consider code executability, making it challenging to use them for benchmarking TDS. Furthermore, individual code units may not be ideally suited for the task at hand (e.g., loose code snippets missing contextual information that hinders executability), adding to the complexity of the process.

To provide evidence that the new features and mechanisms described above do provide value and advance the state of the art, we



(a) (Left) TDS code search editor (LQL + stimulus sheets) that leads to the LSL pipeline on the right (hidden from user)



(b) Search results for a TDS of a *Queue* abstraction

Fig. 5. LASSO TDS - Web UI based on Templating.

adopted an alternative study design involving 20 non-trivial, real-world code search problems related to reuse. The goal of the study was twofold. The first subgoal was to demonstrate the uniqueness of the presented technology and provide evidence that the test-driven filtering step and adaptation technology improve the precision of the search results, relative to the initial text-based search (i.e., IDS). In this case, LASSO TDS is used as a strict TDS engine as described in the previous section. We refer to this approach, with strict behavioral matching criteria, as S-TDS in the remainder of this section. The second subgoal of the study was to demonstrate how the relaxed TDS approach, referred to as R-TDS in the remainder of this section, improves the levels of recall that can be achieved relative to S-TDS without compromising precision.

To this end, we attempted to match and characterize diverse, Java class implementation candidates from a recent snapshot of Maven Central (January 2023) based on several advanced criteria. In contrast to the obsolete first-generation TDS engines, the goal was to demonstrate how various features of LASSO, including IDS, stimulus sheets, adaptation and SRMs, can be used to uncover Java implementations of the chosen functional abstractions. To accomplish this, we made use of LASSO's SRM data structure in two ways to identify functionally equivalent implementations. For S-TDS, we automatically applied behavioral matching criteria (i.e., script-driven), since the expected outputs were provided in the stimulus sheets for the search problems. In addition to the expected responses specified in S-TDS, to realize R-TDS, we analyzed LASSO's SRM records offline, employing an external data analytics tool to manually identify groups of responses from potential implementations that align with the expected responses for our studied functional abstractions.

Finally, to gain a deeper understanding of these matched class implementations, we also measured them using dynamic metrics related to their size, specifically line and cyclomatic complexity (cf. McCabe (1976)) coverage. In the following subsections, we present the specifics of our study design.

6.1. Design

The 20 selected functional abstractions were all analyzed using the same LSL study pipeline script template similar to the one in Listing 2 (except the ranking action). Thus, the given LSL pipeline design realizing S-TDS applies the same basic reuse criteria for all abstractions. Moreover, it is important to stress that even though it prepares a final result set of candidates from a pipeline perspective, it also collects and retains all the SRM-related records in order to realize R-TDS in a data-driven manner (i.e., enable behavioral clustering analysis). In a nutshell, it combines static, textual selection based on IDS with dynamic observations about Java classes using stimulus sheets.

The search pipeline design follows a strict test-driven approach to select matching implementations for each functional abstraction. The same reuse criteria are applied consistently across all functional abstractions, with each abstraction's interface represented using LQL and tests represented using stimulus sheets (including the expected outputs). In the following subsection, we explain the purpose and significance of each action and its crucial configuration parameters.

6.1.1. Actions

Textual selection. The first action, `Select`, performs a textual IDS search, which is explained in detail in Section 4.2. This process identifies Java class candidates that match the interface of the abstraction being analyzed. The top N Java class candidates for each functional abstraction are then returned, ranked by their textual similarity scores. Parameter N was set to 1000, a value which we empirically determined based on the data source used for the study.

To further refine the set of relevant Java class candidates (i.e., express more reuse criteria), we defined additional filters to exclude those assumed to be less important for reuse purposes. These excluded classes include other private classes (based on their visibility concept in Java), abstract classes that may not represent a concrete implementation, and classes belonging to testing facilities or internal packages (e.g., Java JDK).

Code clone detection. Since class code duplicates are usually of little value for reuse tasks, after obtaining the initial set of textual candidates, we took steps to enhance their diversity using Nicad's code clone detection tool (version 6.2) to identify and reject any type-2 syntactic clones (i.e., clones with identical source code). Given the order in which IDS search returns class candidates, we then used this information to identify clusters of potential clone candidates. Subsequently, we removed all class clones except the one with the highest textual score. In cases where multiple candidate scores were equal, we retained the first candidate returned by the textual selection step.

Arena execution. The final action utilizes the arena test driver of the LASSO platform. This step focuses on observing and analyzing the runtime behavior of the diverse set of Java class candidates collected through the IDS step and extensive filtering as explained before. The purpose is to make these class candidates testable in order to execute the stimulus sheets on them.

In the arena action, we limited the number of software adapters computed for each class candidate to a maximum of 100 (a value empirically determined based on the complexity of the interface signatures used in the study). These adapters were then executed on the provided set of stimulus sheets. It is important to note that there may be multiple adapters for a single class candidate that exhibit the same (desired) behavior. Developers often provide various methods with similar functionality but different parameter types, or methods that delegate to other methods, for instance.

The primary objective in the context of the study was to gather SRMs for every adapter implementation so that they could also be analyzed later in an external data analytics tool as well as in the automated pipeline script. To achieve this, we collected two essential types of observational data. Firstly, we recorded the responses generated by each adapted implementation in response to the provided stimuli sheets (i.e., inputs). This allowed us to evaluate and match their exhibited behavior. Secondly, we utilized the arena's internal measurement facility to gather information on the size of matching implementations after the adaptation process. To accomplish this, we employed JaCoCo's code coverage measurement facility to obtain dynamic metrics related to line and branch coverage for each adapted implementation using class-level measurement scope. This provided additional information about the candidates that behaviorally match the desired functional abstraction.

LASSO's grid execution environment was configured to utilize the most up-to-date (long-time support) version of OpenJDK 17 available on DOCKER HUB. This ensured that all worker machines in LASSO's grid ran the candidates in the same controlled environment, to guarantee comparable measurements (i.e., validity of the measurements).

6.1.2. Search problems

The experiment was conducted on 20 non-trivial, real-world functional abstractions sourced from a variety of coding platforms. These abstractions were selected manually to cover various abstract data structures, mathematical operations and auxiliary functions, including popular representations of character sequences (e.g., Base64, JSON, and hashes). We defined each functional abstraction's "typical" interface in LQL and created one or more stimulus sheets to characterize its functionality. Due to space constraints, only a summary of these 20 functional abstractions is provided in Table 2 – each abstraction's name, a brief description of its functionality, the number of methods in its LQL interface, and an overview of the tests (i.e., stimulus sheets) defined to characterize its behavior.

Note that to streamline the study design, we applied the same approach to all 20 abstractions without attempting to incorporate custom

Table 2
Search problems.

Name	Description	# Method Sigs.	Test(s)
Queue	Queue data type	4	First in, first out (FIFO)
FromJson	Parse a Map from a String representing a JSON document	1	Deserialize a Map (key–value pair)
ToJson	Generate a JSON document string from a Map	1	Serialize a Map to JSON representation
B64Encode	Encode characters to Base64 alphabet (require padding of characters)	1	Encode strings including/excluding padding
B64Decode	Decode characters from Base64 alphabet (requires padding of characters)	1	Decode strings including/excluding padding
Fraction	A fraction (math) with numerator, denominator and decimal representation. Simplification allowed.	4	Get numerator, denominator, decimal representation (simplification allowed)
Sha256	SHA256	1	Require string in hexadecimal representation
Matrix	A Matrix data structure	4	Setting/getting values, summing matrices
HtmlSanitizer	Clean untrusted HTML fragments	1	Clean JavaScript alert
MultiMap	A map data structure that holds key-list value pairs.	3	Require values represented as Collection type
Bag	A Bag data structure	5	Add elements and get counts
NGramDist	Distance measure based on ngram (bi-gram)	2	Test for bi-gram distance
NGramGen	Generate word-based bi-grams	1	Test for generating word-based bi-grams
FilenameExt	Extract extension from absolute path	1	Test for extracting extension from absolute file path
DAG	A directed acyclic graph	3	Test for outgoing edges
MinHeap	A min-heap (binary tree) data structure	4	Test for minimum value
MovingAverage	Moving (rolling) average (statistics)	3	Test for change in average when new values are added
Cosine	Cosine similarity of two vectors (from strings)	1	Test similarity measure of two strings
TreeNode	A node in a tree	4	Test for children
Stack	Stack data type (size and empty check)	5	Last in, first out (LIFO)

reuse criteria for each. In practice, however, there are numerous possible variations, such as requiring data types (e.g., `Stack`) to be subtypes of a specific interface in Java (e.g., `java.util.Collection`). Additionally, one may choose to filter textual candidates based on their size-based complexity or enforce other filtering steps during the study design process. These filters can either be added as part of the select action (pre-processing of candidates) or queried within the analysis of SRM records (post-processing of candidates).

6.1.3. Software analytics

Because of LASSO's flexibility, users do not necessarily need to specify an oracle as part of their tests (i.e., predefine values in the output column of stimulus sheets) for Java classes. Instead, they can establish functional equivalence relative to a reference implementation later on. In our case, to realize R-TDS, we used the external analytics tool R to load the SRM records containing the behaviors exhibited by class/adapter combinations from LASSO's database and calculate descriptive statistics about their complexity obtained using JaCoCo. More specifically, we clustered the actuation sheets' responses from class candidates for each functional abstraction's stimulus sheets. By manually inspecting these response clusters, we relaxed our matching criteria and identified one or more response clusters that matched the desired behavior in terms of the equivalence of the returned outputs.

6.1.4. Relative improvements

For the reasons mentioned above, it was not possible to calculate absolute precision and recall values for the search results due to the intractability of establishing a large, comprehensive "ground truth" dataset of executable software components.

To address this challenge, we instead estimated the precision and recall of the S-TDS and R-TDS implementations relative to each other and the first IDS step. Firstly, in order to obtain useful insights into precision, we calculated the relative improvements in the precision of S-TDS over the initial clone-free, textual result set obtained by IDS. Here we assume the clone-free set obtained through IDS contains true as well as false positives. Further, since S-TDS applies strict test filtering criteria, we assume that the set of true positives is approximated by the result set obtained through S-TDS. Of course, the result set of S-TDS may not match the real "ground truth" of true positives, but it is arguably a good estimator of how many candidates actually exhibit the desired behavior.

When we consider the equation for precision (where TP = true positives and FP = false positives) —

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

for both search approaches and remove common terms, we arrive at the following equation to indicate the *relative improvement in precision* (RIP) of S-TDS over IDS —

$$RIP = \frac{C_{IDS}}{C_{S-TDS}} \tag{2}$$

where C_{IDS} denotes the set of clone-free candidates matched by IDS, and C_{S-TDS} the set of candidates matched by S-TDS. In simpler terms, RIP quantifies how often S-TDS outperforms IDS in terms of precision.

Note that R-TDS cannot enhance precision beyond what S-TDS achieves, because both of them filter out the same false positives. Consequently, comparing the relative precision improvements between S-TDS and R-TDS would not be meaningful, as the assumed set of true positives would differ in our approximations of relative improvements in precision. On the other hand, R-TDS can exclusively enhance recall compared to S-TDS, because it has the capability to accept candidates that were initially rejected or classified as false negatives (FN) by S-TDS. Put simply, R-TDS can always be viewed as an enhancement of S-TDS, either by matching the same candidate set or an expanded one.

Similar to RIP, the *relative improvement in recall* (RIR) of C-TDS relative S-TDS can be estimated as follows —

$$RIR = \frac{C_{R-TDS}}{C_{S-TDS}} \tag{3}$$

where C_{S-TDS} denotes the set of candidates matched by S-TDS and C_{R-TDS} the set of candidates matched by R-TDS. In simpler terms, RIR quantifies how often R-TDS outperforms S-TDS in terms of recall.

To further strengthen the validity of our approximations and results, we randomly sampled candidates from the result sets matched by the three CSE strategies and manually examined their behavior.

6.2. Results

The findings of the study were obtained using a grid of 10 nodes, each equipped with Ubuntu Linux 22.04 LTS and uniform processing capabilities, including a 12th Gen Intel(R) Core(TM) i9-12900 processor and a maximum memory capacity of 12 GB RAM. One of these nodes served as the manager node, responsible for executing the pipeline script and allocating tasks, while the remaining nine nodes

Table 3
Results for 20 search problems (with max. textual = 1000, max. adapters = 100).

	Queue	*FromJson*	*ToJson*	*B64Encode*	*B64Decode*	*Fraction*	*Sha256*	*Matrix*	*HtmlSanitizer*	*MultiMap*	*Bag*	*NGramDist*	*NGramGen*	*FilenameExt*	*DAG*	*MinHeap*	*MovingAverage*	*Cosine*	*TreeNode*	*Stack*
Duration																				
Search time (min)	17:42	01:33	01:37	01:12	01:06	03:21	01:27	01:47	01:47	01:55	03:10	03:06	05:30	01:43	01:49	11:12	01:59	07:58	05:15	04:30
Textual (IDS)																				
#classes	1 000.00	1000.00	1000.00	1000.00	1000.00	1000.00	1000.00	1000.00	1000.00	1 000.00	1 000.00	1000.00	1000.00	1000.00	1 000.00	1 000.00	1000.00	1000.00	1 000.00	1 000.00
#clones	525.00	239.00	292.00	556.00	497.00	540.00	424.00	415.00	393.00	558.00	553.00	386.00	365.00	469.00	475.00	n/a	400.00	332.00	219.00	517.00
#clone-free	475.00	761.00	708.00	444.00	503.00	460.00	576.00	585.00	607.00	442.00	447.00	614.00	635.00	531.00	525.00	n/a	600.00	668.00	781.00	483.00
Adaptable																				
#classes	192.00	349.00	498.00	438.00	485.00	445.00	266.00	258.00	421.00	392.00	288.00	50.00	95.00	79.00	229.00	760.00	548.00	381.00	719.00	372.00
#adapters	19 200.00	564.00	716.00	1801.00	2029.00	44 500.00	499.00	18 905.00	1021.00	16 410.00	27 010.00	191.00	166.00	127.00	12 337.00	66 075.00	8490.00	1312.00	60 300.00	37 200.00
Testable																				
#classes	192.00	275.00	485.00	438.00	485.00	121.00	240.00	28.00	398.00	88.00	277.00	28.00	70.00	61.00	139.00	690.00	388.00	301.00	428.00	369.00
#adapters	11 495.00	401.00	671.00	1573.00	1926.00	6161.00	445.00	1373.00	930.00	1 699.00	19 068.00	127.00	116.00	97.00	3 811.00	40 366.00	4768.00	989.00	10 994.00	23 506.00
Behavioral clusters																				
#response clusters	193.00	12.00	50.00	70.00	23.00	131.00	56.00	90.00	131.00	77.00	1 658.00	27.00	25.00	15.00	67.00	1 432.00	135.00	103.00	762.00	797.00
⌀ cluster size	59.56	33.42	13.42	22.47	83.74	47.03	7.95	15.26	7.10	22.06	11.50	4.70	4.64	6.47	56.88	28.19	35.32	9.60	14.43	29.49
Matches (S-TDS)																				
#matched classes	5.00	214.00	335.00	383.00	462.00	20.00	55.00	24.00	4.00	27.00	1.00	5.00	1.00	5.00	14.00	8.00	43.00	4.00	35.00	65.00
#matched adapters	5.00	271.00	418.00	747.00	1129.00	24.00	64.00	116.00	5.00	58.00	1.00	10.00	1.00	5.00	32.00	8.00	53.00	8.00	74.00	161.00
#failed classes	187.00	61.00	150.00	55.00	23.00	101.00	185.00	4.00	394.00	61.00	276.00	23.00	69.00	56.00	125.00	682.00	345.00	297.00	393.00	304.00
#failed adapters	11 196.00	91.00	206.00	193.00	71.00	5255.00	321.00	153.00	920.00	1 163.00	18 982.00	117.00	115.00	83.00	3 633.00	39 819.00	4131.00	981.00	9 819.00	19 304.00
Matches (R-TDS)																				
#matched classes	79.00	214.00	383.00	383.00	465.00	64.00	62.00	24.00	62.00	28.00	16.00	5.00	1.00	33.00	45.00	8.00	62.00	4.00	35.00	198.00
#matched adapters	187.00	271.00	492.00	747.00	1134.00	117.00	82.00	116.00	95.00	60.00	54.00	10.00	1.00	34.00	224.00	8.00	83.00	8.00	74.00	423.00
#failed classes	113.00	61.00	102.00	55.00	20.00	57.00	178.00	4.00	336.00	60.00	261.00	23.00	69.00	28.00	94.00	682.00	326.00	297.00	393.00	171.00
#failed adapters	6802.00	91.00	140.00	193.00	64.00	2044.00	304.00	153.00	722.00	1 147.00	17 720.00	117.00	115.00	32.00	1 835.00	39 819.00	3816.00	981.00	9 819.00	10 718.00
Rel. improvements																				
in precision	95.00	3.56	2.11	1.16	1.09	23.00	10.47	24.38	151.75	16.37	447.00	122.80	635.00	106.20	37.50	0.00	13.95	167.00	22.31	7.43
in recall	15.80	1.00	1.14	1.00	1.01	3.20	1.13	1.00	15.50	1.04	16.00	1.00	1.00	6.60	3.21	1.00	1.44	1.00	1.00	3.05
Characterization																				
Lines																				
min	8.00	1.00	1.00	1.00	1.00	4.00	1.00	15.00	1.00	1.00	1.00	40.00	14.00	2.00	2.00	19.00	1.00	3.00	4.00	2.00
q25	22.00	3.00	3.00	5.00	17.50	10.00	3.00	22.00	3.00	12.00	15.00	40.00	14.00	7.00	8.00	19.00	8.00	3.00	9.00	10.00
median	41.00	6.00	6.00	29.00	31.00	13.00	6.00	23.00	11.00	18.00	20.00	40.00	14.00	7.00	12.00	19.00	10.00	3.00	10.00	11.00
q75	60.00	11.00	11.00	52.00	52.00	22.00	10.75	28.25	17.75	18.00	22.00	40.00	14.00	8.75	19.00	19.75	14.50	3.00	14.75	14.00
max	78.00	97.00	55.00	106.00	200.00	35.00	18.00	46.00	140.00	18.00	33.00	40.00	14.00	22.00	48.00	41.00	48.00	3.00	22.00	28.00
mean	39.65	11.73	9.49	32.99	34.53	15.88	7.07	25.75	19.08	14.50	19.33	40.00	14.00	8.74	14.76	22.12	12.05	3.00	11.88	11.60
sd	19.52	15.49	10.20	27.37	25.51	7.26	4.74	8.67	29.25	4.94	8.32	0.00		5.11	9.54	7.70	8.34	0.00	5.09	4.66
Cycl. complexity																				
min	4.00	1.00	1.00	1.00	1.00	4.00	1.00	7.00	1.00	1.00	1.00	13.00	5.00	1.00	1.00	5.00	1.00	2.00	3.00	1.00
q25	7.00	2.00	2.00	3.00	5.00	4.00	2.00	8.00	2.00	5.00	6.00	13.00	5.00	2.00	3.00	5.00	3.00	2.00	5.00	4.00
median	9.00	3.00	3.00	7.00	8.00	4.00	3.00	9.00	2.00	6.00	6.00	13.00	5.00	2.00	4.00	5.00	4.00	2.00	6.00	6.00
q75	10.00	4.00	4.00	11.00	10.00	5.00	4.00	10.25	4.00	6.00	9.00	13.00	5.00	3.00	6.00	7.00	4.75	2.00	6.00	8.00
max	13.00	34.00	15.00	29.00	25.00	13.00	7.00	17.00	19.00	7.00	10.00	13.00	5.00	6.00	10.00	12.00	9.00	2.00	8.00	12.00
mean	8.46	4.13	3.28	8.20	8.67	4.78	3.01	9.79	3.84	5.40	6.74	13.00	5.00	2.74	4.37	6.38	3.85	2.00	5.76	6.06
sd	1.85	4.04	1.90	6.21	5.66	1.25	1.43	2.85	3.76	1.29	2.11	0.00		1.42	2.02	2.45	1.47	0.00	1.24	2.36

functioned as worker nodes, dedicated to running the candidate implementations (i.e., were responsible for the arena test driver execution). The manager node assigned the arena workload evenly among the worker nodes by partitioning the candidate implementations into nine blocks and distributing them in a round-robin manner. Table 3 provides a summary of the search time for each problem studied.

The average time taken for the search was 3 min and 59 s. Although most searches were completed in less than 5 min and 30 s, certain exceptions were observed, with the *Queue* search problem taking the longest time, followed by the $MinHeap$ search problem. Three basic factors contribute to extended search times: (1) the presence or absence of a candidate's artifacts on the worker node, necessitating downloading from the corpus,[4] (2) running into timeouts set by the platform/arena during execution, which may arise from undesirable run-time behavior due to the inputs and/or the tried adapter, and (3) the size of an abstraction's interface, as determined by its method signatures and the number of parameters, leading to a greater number of combinations (i.e., adapters) that need to be prioritized, and a possibly longer execution time due to longer test sequences. Note that the timeouts giving rise to the second factor can also occur as part of the expected behavior. For example, a *Queue* implementation that employs blocking may intentionally halt the execution of the ENQUEUE method until it is feasible to add another element, based on the capacity of the queue (i.e., a "blocking" queue).

As well as giving the search times, Table 3 summarizes the outcomes of the matching processes for the 20 search problems. This includes the results for each functional abstraction and three corresponding analysis findings. Each column within the summary table represents a functional abstraction and the corresponding classes processed by the pipeline, while the rows in the top third of the table provide information about the number of processed and matched classes by IDS as well as adapters. The rows in the middle third of the table provide information about the behavioral clusters and matches by S-TDS and R-TDS. Finally, the rows in the bottom third of the table characterize the final classes (and adapters) matched by R-TDS based on JaCoCo's line- and cyclomatic complexity measures for each functional abstraction.

To provide further insights into the kinds of software components retrieved, we have also included a table displaying 25 randomly sampled members of the set of class implementations retrieved for the queue abstraction in Table 4. These results were taken from the overall set of retrieved *Queue* implementations.

6.2.1. Textual selection and code clones

For each functional abstraction under study, the prototype platform successfully textually matched $N = 1000$ class candidates using IDS. From these matches, Nicad identified an average of approximately 430 class clones across all abstractions, which were removed from further processing in the pipeline. This left clone-free sets of classes containing

[4] Occasionally, transitive artifact dependencies present in third-party repositories may be inaccessible, potentially leading to network timeouts.

Table 4
25 randomly sampled implementations retrieved for *Queue* (excerpt out of 79).

	Class	Package	Project (Maven URI)
1	SinglyLinkedList	edu.columbia.cs.psl.phosphor.struct	io.rivulet:embedded-server:1.0.0
2	Queue	org.eclipse.core.internal.jobs	org.jibx.config.3rdparty.org.eclipse:org.eclipse.core.jobs:3.5.100
3	ArrayBlockingQueue	jadex.commons.collection	org.activecomponents.jadex:jadex-commons:3.0.117
4	Queue	edu.princeton.cs.introcs	com.github.fracpete:princeton-java-examples:1.0.2
5	ResizableCapabilityLinkedBlockingQueue	com.github.seaframework.core.queue	com.github.seaframework:sea-core-basic:1.0.0
6	GrowableArrayBlockingQueue	org.apache.pulsar...util.collections	io.streamnative.connectors:managed-ledger-shaded:2.7.6
7	ObjectHeapPriorityQueue	it.unimi.dsi.fastutil.objects	co.datadome:fastutil-core:8.5.11.1
8	BlockingQueueImpl	org.kaazing.net.impl.util	org.kaazing:gateway.client.java.internal:5.0.0.17
9	FastQueue	org.apache.phoenix.shaded.org.antlr.runtime.misc	org.apache.phoenix:phoenix-client-hbase-2.4:5.1.3
10	ClosableBlockingQueue	org.apache.flink.streaming.connectors.dis.internals	com.huaweicloud.dis:huaweicloud-dis-flink-connector_2.11:2.0.1
11	GrowableArrayBlockingQueue	org.apache.pulsar.common.util.collections	org.apache.pulsar:pulsar-common:2.11.0
12	ObjectHeapPriorityQueue	it.unimi.dsi.fastutil.objects	org.apache.hivemall:hivemall-all:0.6.0-incubating
13	ArrayFIFO	toools.collections	io.github.lhogie:toools:0.0.7
14	HashedQueue	io.pivotal.arca.threading	io.pivotal:arca-threading:1.0-beta.3
15	ObjectArrayPriorityQueue	it.unimi.dsi.fastutil.objects	co.datadome:fastutil-core:8.5.11.1
16	PriorityQueueImpl	org.tinygroup.queue.impl	org.tinygroup:org.tinygroup.queue:3.4.9
17	FastQueue	infinispan.org.antlr.runtime.misc	org.infinispan:infinispan-embedded-query:9.1.7.Final
18	RingArrayBlockingQueue	com.serialpundit.core.util	org.bidib.com.serialpundit:sp-core:1.0.4
19	ObjectQueue	net.oschina.dajiangnan.cmppclient.util.waitqueue	net.oschina.dajiangnan:cmppclient:1.0.0
20	ArrayQueue	org.eclipse.jetty.util	org.testatoo.container:testatoo-container-jetty-full:1.0-rc5
21	GrowableArrayBlockingQueue	com.yahoo.pulsar.common.util.collections	com.yahoo.pulsar:pulsar-common:1.18
22	Queue	org.apache.tomcat.util.collections	tomcat:tomcat-util:5.5.4
23	BoundedPriorityQueue	com.gemstone.org.jgroups.oswego.concurrent	io.snappydata:gemfire-jgroups:2.0-BETA
24	Queue	org.apache.phoenix.shaded.org.apache.jasper.util	org.apache.phoenix:presto-phoenix-client-shaded:4.14.1
25	LinkedTransferQueue	com.google.code.yanf4j.util	com.googlecode.xmemcached:xmemcached:2.4.7

an average of 570 class candidates for each functional abstraction, excluding the `MinHeap` abstraction since Nicad encountered parsing errors when processing a subset of the class candidates retrieved.[5]

6.2.2. Adaptation and matches

During the arena analysis step, the adaptation engine managed to compute adapters for an average of 363 classes, with an average of approximately 15,942 adapters. For each class, the number of testable classes (i.e., those that could be executed using the given stimulus sheets) varied significantly, averaging around 275 testable classes (or roughly 6525 testable adapters on average).

In addition to these findings, for S-TDS we identified an average of approximately 89 functionally-equivalent class implementations across all abstractions (averaging about 167 matching adapters). In contrast to S-TDS, R-TDS obtained roughly 113 functionally-equivalent class implementations across all abstractions (totaling about 221 matching adapters on average).

These results for both variants of TDS indicate that a substantial number of classes were able to provide the desired functionality based on our interface specifications and stimulus sheets.

For each functional abstraction, we successfully matched a non-trivial number of class implementations, except for the *NGramGen* abstraction where we only found one matching implementation in the corpus. Upon manual inspection of the SRM records, we discovered that bi-gram functionality existed in more classes, but not in the way we had specified (i.e., as word-level bi-grams while other classes delivered token-level bi-grams). The number of response clusters varied significantly among functional abstractions and testable classes. Even when dealing with large numbers of response clusters, identifying matching clusters was relatively straightforward in R-TDS since we could first match a subsequence of returned outputs to narrow down the set of relevant clusters.

[5] Presumably, the most recent Nicad version that we used had issues with the most recent Java language syntax.

6.3. Relative improvement in precision and recall

To evaluate whether incorporating a test-driven filtering step enhances the precision of IDS, we first compared the class matches of S-TDS and R-TDS with the textual ranks as determined by IDS's text-based search. The ranking is based on scores returned as part of IDS. To exclude code duplicates, we recalculated the ranks using the scores of the clone-free matches.

Fig. 6 illustrates how many classes where matched based on the clone-free IDS set by either of the studied TDS approaches. While the x-axis represents the textual ranks determined by IDS's text-based search, the red line signifies the highest number of candidates remaining in IDS's result set after clone removal. On the y-axis, we have the cumulative sum of matches up to a specific rank for both S-TDS (shown as the orange line) and R-TDS (represented by the green line). It is evident that for a considerable number of functional abstractions, R-TDS consistently outperformed S-TDS in terms of candidate matches, demonstrating its superior recall capabilities.

An analysis of the overall trend of the cumulative sums of behavioral matches for most abstractions shows that growth tends to plateau at a certain point, resulting in an asymptotic curve shape. Although there are some abstractions with only a small set of matched implementations, they also seem to follow the same growth trend.

To assess improvements in precision and recall, we utilized the previously introduced metrics of RIP (Relative Improvement in Precision) and RIR (Relative Improvement in Recall). The improvements for each functional abstraction are shown in the "Rel. Improvements" row of Table 3. On average, across all functional abstractions, both S-TDS and R-TDS surpassed IDS in terms of precision by a factor of approximately

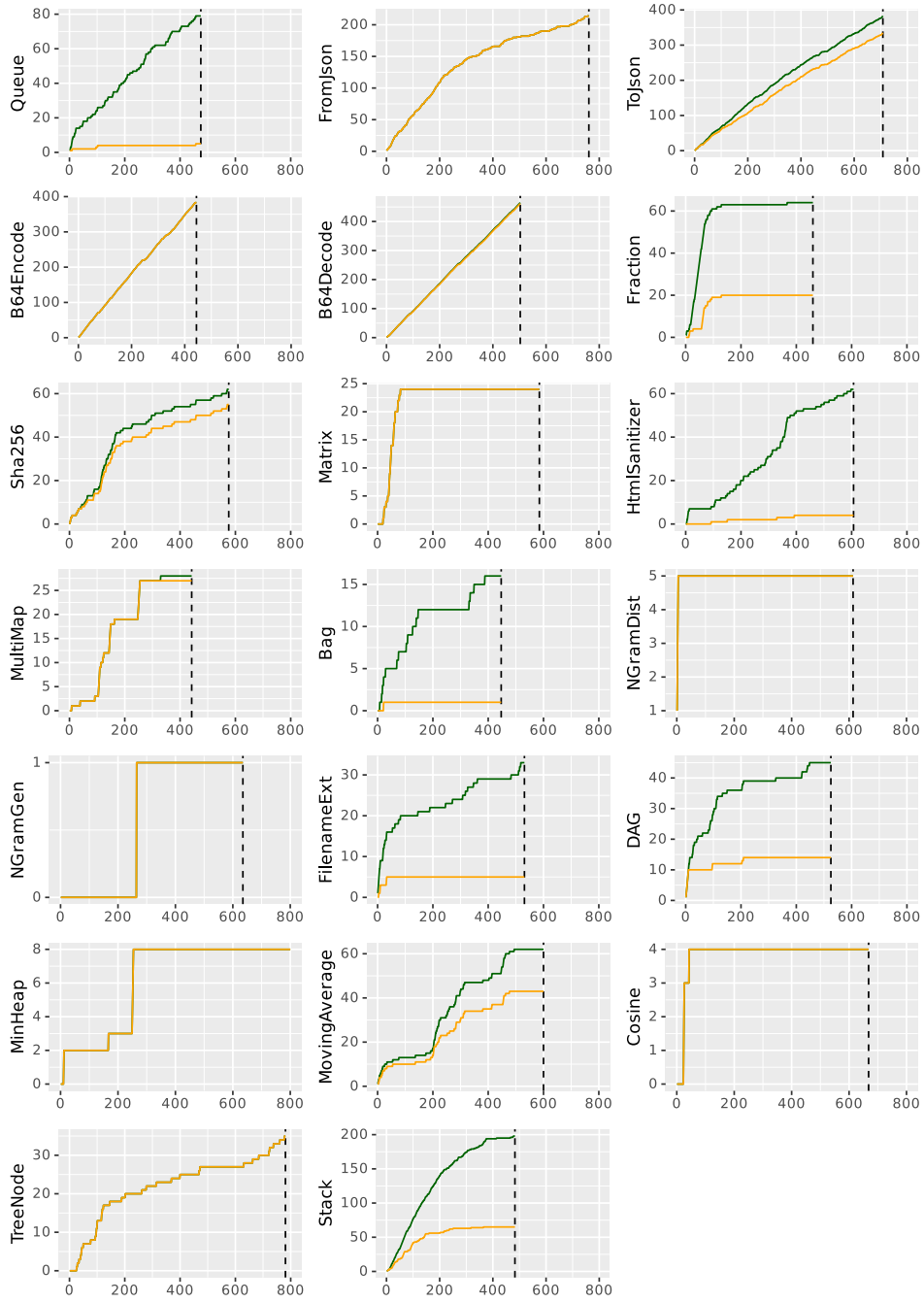


Fig. 6. Trend of behavioral matches — rank of textual matches (x-axis) (as returned by IDS after code clone removal) vs. class implementation matches (y-axis) - S-TDS (orange) and R-TDS (green). The dashed black line indicates the number present after clone removal.

99.37. On the other hand, R-TDS showed a recall improvement factor relative to S-TDS of approximately 4.

In summary, while S-TDS exhibited a substantial relative improvement in precision compared to IDS, R-TDS demonstrated a significant boost in recall over S-TDS by accepting candidates that S-TDS had rejected.

6.3.1. Characterization of software components

The third part of Table 3 presents an overview of the behavioral matches obtained by R-TDS over all matching class implementations in terms of their implementation size (measured by line-count and complexity at run-time when stimulus sheets were executed). The variability in size-based measures is considerable across different functional abstractions. Typically, a line count of 1 represents a delegate method invocation.

In conjunction with Table 4, which contains a random sample of queue class implementations, we can appreciate the diversity of results obtained from our analysis. This suggests that LSL is a powerful dynamic query language for achieving component diversity, allowing users to choose from a wealth of diverse candidates.

7. Discussion

In this section, we discuss the overall lessons that can be learned from the presented study and consider the validity of the results, including the approach's limitations. We then look at the extent to which the presented platform addresses the weaknesses of the first generation of CSEs identified in Section 1, before discussing the technology's potential impact on the quality and usability of code search results.

7.1. Lessons learned

Our study shows that LASSO TDS is indeed capable of retrieving a large number of diverse (adapted) implementations from LASSO's code repository for a set of 20 functional abstractions. During the course of our analysis, we also made several intriguing observations that could potentially help enhance the technology by making it more tolerant of varying responses and able to accommodate potential differences in interface signatures.

For instance, when examining functional abstractions such as `Queue` (along with related data structures like `Stack`), we discovered implementations exhibiting a wide range of behavior despite having high levels of similarity. In the case of the `enqueue` method for the `Queue` abstraction, we identified three distinct response clusters that matched the desired functionality, even though the method signature and its return type varied significantly. Similarly, when analyzing the `ToJson` abstraction, multiple relevant response clusters were discovered, with differences in how the final JSON representation was represented as a string (e.g., some returned pretty-printed strings while others produced single-line strings). LASSO TDS's strict variant simply rejects all these alternatives, whereas its relaxed variant has the potential to recognize and accommodate them.

7.2. Limitations and threats

Although all first generation TDS engines have been deprecated and are no longer available as benchmarks for comparison, we were nevertheless able to show that LASSO TDS is able to improve on the precision and recall of "classic" CSEs based on their well known implementation strategies. As well as providing evidence for the feasibility of the approach presented in this paper, the study also shows the power and flexibility of LASSO's dynamic scripting language, LSL, for defining customized search pipelines and services.

Even though the study was performed using a large dataset of open source Java software components sourced from Maven Central, which is widely utilized in both open source and industrial development, the generalizability of our findings are limited by the small scale of our study. Conducting a larger-scale study is essential to attain more reliable results. Furthermore, it is important to note that LASSO currently focuses exclusively on Java software components, and therefore, we cannot make any assertions about its applicability to other programming languages like Python. The asymptotic curves identified in our study also warrant further investigation. To pinpoint the true plateaus where behavioral matches reach saturation, an analysis of larger result sets obtained through IDS is necessary.

There are several potential threats to the design of our study. Firstly, the functional equivalence of the retrieved implementations was primarily achieved through the comparison of quantitative values (i.e., output values). To increase our confidence that this worked as expected, we randomly selected a subset of implementation matches from the search problems and manually reviewed their code. Secondly, the sequence sheets used to characterize the desired functionality may not be entirely representative. Given that exhaustive software testing is rarely practical (Ammann and Offutt, 2016), a limited set of tests may not fully capture the actual behavior of candidates during test-driven filterings in LASSO TDS. To address this concern, we endeavored to create tests that accurately reflect the fundamental concept of the functionality being evaluated.

A related concern stems from the measurement of relative improvements in recall for R-TDS compared to S-TDS. This requires a ground truth to evaluate alternative, acceptable outputs for R-TDS that are deemed equivalent. In order to mitigate this risk, we manually examined all the alternative outputs for the functionality in question and carefully assessed whether the implementations effectively deliver the desired functionality. However, it is important to acknowledge that other human examiners may form different judgments based on their individual experiences or varying interpretations of the functionality.

Our approach has other inherent limitations that require further exploration and research. Firstly, software adaptation is still a significant challenge, particularly when dealing with class implementations that rely on, or involve complex, custom (i.e., developer-defined) types. The problem is that potential class implementations may be missed by the approach, even though a potential adapter could be developed (i.e., from a developer's perspective). This has a negative impact on the recall of the approach.

Secondly, even though the process of establishing functional equivalence of class implementations in an offline, data-driven manner disconnects the matching process from run-time observations and comparisons, our current serialization technique for matching outputs represented as strings of Java objects is limited. This could represent a potential risk in cases where string-based equivalence varies from object-based equivalence. To mitigate this risk, we employ two strategies: (1) utilizing established methods for object serialization via JSON, and (2) making it possible for users to examine SRM records more closely (e.g., examining behavioral clusters for spotting discrepancies).

Finally, executability, and thus the ability to actually test and uncover the behavior of software components, is still an issue that needs further research. Although components may compile without issues, their execution is not guaranteed. This is partly due to potential late-time binding problems with class dependencies or dependencies on external services. In the worst-case scenario, a component's behavior may not be observable (e.g., visibility issues in Java) or even testable at all (cf. testing limitations Ammann and Offutt, 2016).

The potential impact of the improvements outlined above is significant, especially in terms of enhancing recall by exploring alternative potential outputs to those envisaged by the developer. This insight underscores the need for the further development of adaptation and similarity techniques for matching outputs.

7.3. Addressing weaknesses of CSEs

In this subsection, to complement our quantitative results, we discuss how well our proposed platform resolves the shortcomings of existing CSEs outlined in Section 1.

1. *Small corpora*: As explained in Section 3, the LASSO platform has a large index of executable software components extracted primarily from Maven Central, augmented by a steadily growing corpus of "mavenized" code harvested from other platforms. Once the "mavenization" mapping has been defined for a particular repository's build information format, the process of harvesting new code is automatic and performed on an ongoing basis. To the best of our knowledge, at the time of writing, the LASSO corpus is the largest repository of *automatically* executable software available. In order to compare the magnitude of our corpus with that of other corpora, we use "artifacts" (cf. Table 1) as our fundamental unit of measurement. Given that an executable artifact is typically generated from a software project (i.e., its source code in terms of compilation units), which usually encompasses an executable Java program (i.e., compiled code), we can compare the number of executable artifacts in LASSO's corpus to the number of executable Java programs reported for other corpora published in the literature. Our corpus contains 184,464 executable artifacts sourced from Maven Central, which is equivalent to approximately 85k (i.e., 85%) more executable Java programs than the next biggest corpus we are aware of, NJR corpus (Palsberg and Lopes, 2018), and roughly 135k (i.e., 270%) more than the 50k-C (Martins et al., 2018b), which is the third biggest. Other popular corpora primarily used for benchmarking tasks are much smaller (e.g., $SF110$ contains 110 executable Java programs (Fraser and Arcuri, 2014), while XCorpus contains 76 (Dietrich et al., 2017)).
2. *Poor scalability and performance*: As explained in Section 4, LASSO's distributed architecture and its execution environment is powered by Apache Ignite (The Apache Software Foundation, 2022a) which is a fully-blown clustering technology supporting vertical and horizontal scaling of workloads, including both the building and execution of software components as well as the collection and storing of observational records. It supports scalable build automation and build script synthesis using Maven as the build tool ecosystem. This not only dramatically increases the speed at which search results can be processed, it also allows the platform to be easily scaled up with more computing resources. The findings of our study suggest that the search time, which is approximately 4 min on average for 1000 candidates, underscores the platform's practical utility. This is particularly relevant when taking advantage of pre-existing computing resources, such as continuous integration systems. Since the secure sandbox environment for executing actions and components is based on docker containerization at the operating system level, it supports the specification of controllable, reusable and isolated run-time environments (e.g., in terms the operating system, Java version etc.) and enables fine-grained control over resource allocation and permissions (Boettiger, 2015).
3. *Declarative query languages*: As mentioned by Grazia and Pradel (2022), existing CSEs exclusively support declarative query languages that basically allow users to describe the properties they want the desired code to have. However, this approach is problematic for test-driven search engines because (a) the desired properties include the specification of desired semantics expressed as tests, and (b) in practice, the search process is at least a dual-phase process in which a dynamic-observation step is built on top of a traditional text-based search step. Expressing this combination in a declarative query language would be complex and counterintuitive. LASSO's rich, but nevertheless abstract, scripting language allows static and dynamic analysis steps (including search steps) to be combined into imperative process pipelines. These scripts are not incompatible with, or intended to completely replace, traditional code search query languages, but can be used to realize them as shown by LSL. This approach of complementing traditional, declarative query languages with an imperative, but abstract, scripting language provides users with a range of ways to use the platform for code searches and analyses.
4. *Platform-specific test description language*: In first generation TDS engines, users could only define and input the tests describing their sought-after functionality using mainstream programming languages such as JUnit test classes/methods. This not only meant that users had to expend significant effort in coding up a compilable and executable test, they had to use fully-blown programming languages which obscure the core input/output mappings characterizing the desired functionality (e.g., within the assertion statements). The sequence sheet notion offered by LASSO, supported in LSL, provides a much more compact (yet executable) representation of tests that can be more easily incorporated into declarative queries.
5. *Limited adaptation technology*: A key feature of LASSO's automated execution technology is the inclusion of a range of adaptation heuristics that attempt to adapt the interfaces of candidate code modules to the interfaces assumed in the sequence sheet test specifications. This can significantly increase the recall of the search process by allowing candidates to be assessed that otherwise would not match the interface.
6. *Simplistic ranking approaches*: The current generation of CSEs use statically-derived metrics to rank the members of a result set since they are unaware of their true run-time behavior. Test-driven search engines, which execute the candidate code modules, can supplement the static metrics with easily measured dynamic properties such as execution time and resource usage. If all candidate code components are executed under the same conditions, such metrics are meaningful because by definition they involve the same tests. However, defining ranking algorithms that take the dynamic behavior of the components into account is much more challenging because it requires similarity metrics based on a full record of each component's responses to each test. For example, software components could be ranked according to how many of the tests they pass. As explained in Section 3, LASSO supports such complex behavior comparisons by storing all execution information (i.e., observational records) in SRMs and allowing them to be exported for analysis using state-of-the-art data analysis tools.
7. *Lack of dynamic metrics about the result set*: Finally, a natural consequence of the previously discussed feature is that LASSO can significantly enhance the value of the returned component descriptions by augmenting traditional static metrics (e.g., LOC, cyclomatic complexity) with dynamic metrics (e.g., execution speed, resource usage, functional similarity etc.).

7.4. Additional benefits and impact

Overall, the presented search scripts indicate that only a small set of straightforward, recurring actions are necessary to realize customized, behavior-aware search processes. Despite being formatted in an easily readable manner, the size of the LSL scripts used to implement LASSO TDS is relatively small, demonstrating the power and efficiency of LSL as a dynamic querying language. This compactness gives users more flexibility to develop new search strategies by either modifying existing actions or adding additional ones that incorporate custom reuse criteria or unique search techniques.

We believe that even novice users can quickly grasp the supplied search scripts and adapt them to their unique functional/non-functional

requirements in a particular reuse scenario. As with any other programming language, LSL scripts can be decomposed into smaller subscripts (e.g., divide-and-conquer) thanks to the flexibility of Groovy/Java. Even if users prefer not to use LSL as their query language, they still have access to simpler, classic CSE's through the LASSO TDS service, which can be utilized via a web front end or through plug-in integration.

To the best of our knowledge, no existing CSE provides a dynamic query language that integrates observational services with cutting-edge analysis tools. Even though Boa (Dyer et al., 2013) is not a CSE, it does provide a DSL to answer questions at the code repository level. The drawback of Boa, however, is that it does not combine its static analysis capabilities with dynamic analysis like our platform. It therefore lacks behavior-aware selection capabilities (i.e., behavior sampling).

7.4.1. Software measurement

Since LASSO supports the combination of static and dynamic code analysis, SRMs can store virtually any kind of measurable data about software components (even non-numerical data). Users can therefore formulate and combine a large variety of analysis criteria based on a wide range of measurable engineering goals. The collection of measurements in SRMs serves multiple purposes —

- *Quality-based reuse selection criteria*: Measurements can be used to identify the most suitable implementations for reuse based on predefined quality attributes (Long, 2001; Bauer and Vetro', 2016).
- *Differences and comparisons*: Measurements reveal discrepancies between, and interesting observations about, different components involved in the analysis process, helping users make informed decisions about whether a particular candidate is suitable for their purposes (e.g., estimating the potential reuse cost Barns and Bollinger, 1991).
- *Increased transparency*: By gathering more information about tested components, users can identify code clones or undesired structural anti-patterns, leading to better decision-making and increased code understanding.
- *Dynamic metrics guidance*: In cases where no true positives are found, the gathered properties about components can be used to redesign test sequences in TDS.

7.4.2. Fine-grained filtering and sorting options

LASSO TDS offers multiple ways to filter candidates based on various properties including (1) text-based filters (applied at the level of the executable corpus), (2) action filters and (3) behavior filters imperatively encoded in LSL actions (inside the `whenAbstractionsReady` block). These filtering options can be used independently or combined into a hybrid strategy for improving precision in the search process.

In addition, type-driven (or structural) queries can be used to retrieve classes based on their specific types (e.g., extending a certain super class or implementing an interface). This feature is particularly useful for targeted searches within specific application domains (e.g., requiring queues to be subtypes of `java.util.Collection`). LSL pipelines also provide flexibility in reformulating filtering-based reuse criteria into ranking-based reuse criteria. Users have the option to encode custom, relevance-based scoring methods within LASSO TDS, allowing best matches to be identified and ranked based on their relevance to specific reuse objectives (in contrast to the binary choice of rejecting them).

7.4.3. Executability, testability and environments

While our "mavenization-based" corpus unification and curation approach aims to improve the rate of success in obtaining executable components, the automated build script synthesis capability provided by Maven simplifies the incorporation of reusable components into Java projects. Since users often need to make sensitive choices about actual target execution environments in which potential reuse candidates may be executed, LASSO lets them leverage state-of-the-art containerization technology to enable the automatic selection and creation of execution environments on-the-fly. This innovative approach streamlines the process of defining and managing desired test execution environments.

As demonstrated in Listing 2 for LASSO TDS, users have the flexibility to specify their target execution environments (here Java version 17). This allows them to assess whether potential reuse candidates meet their requirements. Additionally, TDS pipelines can easily be modified or extended to verify that systems execute consistently across multiple environments (e.g., by comparing SRMs).

By offering these advanced features and maintaining a high degree of flexibility, LASSO empowers users to more effectively leverage existing software components in their projects, ultimately enhancing the overall quality and efficiency of their reuse and development efforts.

8. Conclusion

A significant weakness of the current generation of CSEs, as they attempt to cope with the huge volumes of code being added to repositories on a daily basis, is their almost exclusive reliance on statically-derived properties of software to match code to queries. This means that they can only make judgments about components' semantics based on unreliable inferences from the identifiers selected by the author in the source code, which has direct negative consequences on their precision and recall. Since so-called "symbolic execution" approaches (Baldoni et al., 2018) are still only applicable to small snippets of code, the only practical way of gaining information about the "'true" run-time behavior of code modules is to execute them. However, scaling such "test-driven search" approaches to significant levels, and making them usable in practice, presents several challenges.

In this paper we have presented the LASSO platform,[6] and LASSO TDS, a test-driven code search service built on top of that platform, that addresses these challenges by automating the execution and run-time comparison of large numbers of software components. Moreover, based on a study of 20 search problems we gained preliminary evidence that these capabilities can enhance several aspects of the classic TDS reuse scenario. We demonstrated LASSO TDS's performance in terms of (a) relative precision improvements over IDS-based textual search as well as in terms of relative recall improvements over traditional, strict TDS approaches. In particular, LASSO's test-driven search capabilities are able to establish the functional relevance of candidate implementations more scalably and reliably than traditional static (i.e., text-based) algorithms, thereby increasing search precision. Moreover, LASSO's interface adaptation and matching techniques can increase the number of candidate software implementations that can be subject to testing, thereby increasing recall. Furthermore, we demonstrated that relaxed TDS implementations can further improve recall, relative to strict TDS implementations, by generalizing the matching criteria (i.e., explicitly allowing different, but semantically equivalent, outputs), without compromising precision. Currently, the LASSO platform only supports Java, but we are working on adding other languages such as Python.

Of course, the precision and recall of search algorithms can always be optimized individually, so the challenge is to find techniques that can raise one, or both, without affecting the other. LASSO's dynamic analysis services can be used to generate deeper insights into the clusterings of behaviorally-similar implementations in the search results, potentially leading to even further precision, and providing a range of useful dynamic metrics that better characterize the relevance of candidate implementations. All this additional information can be used to increase the quality of the final ranking algorithm, and thus

[6] Freely available at GitHub: https://github.com/SoftwareObservatorium/lasso.

maximize the likelihood that the most suitable implementations will be offered to the users first. Many of the example measures and services are unique to LASSO, or are not available at the large scale offered by LASSO.

We believe that LASSO has the potential to further enhance software reuse. As well as encoding custom search processes and rich reuse criteria, the dynamic LSL query language can be used to support "continuous reuse" (Kessel and Atkinson, 2018). Since LASSO is compatible with modern continuous integration platforms, its dynamic analysis capabilities can be exploited at opportune moments for continuous code recommendation. For this, we envision integrating LASSO into continuous integration (CI) pipelines (Fowler, 2006) to allow developers to receive recommendations in an unobtrusive manner based on text-based TDS queries that are generated and submitted automatically. This service can be used to identify reusable candidates when the CI system is idling, and to provide fast feedback to developers if existing systems already exist. This opens up the possibility for companies to discover existing functionality they were unaware of in early phases of their projects, thus making it possible to reduce overall development costs.

LASSO's flexible pipeline design can also enable the benchmarking of new and existing search strategies, or even the comparison to code generation strategies such as offered by generative AI (Chen et al., 2021). Since search strategies are encoded in LSL scripts, LASSO allows them to be assessed with respect to important evaluation criteria (i.e., precision and recall). In particular, users can compare the SRMs produced by each search script using LASSO's data analytics layer. Benchmarking can also be used to improve existing search strategies and to assess new ones. For example, the precision of interface-driven search can be assessed by looking at the number of false positives when evaluated in the arena based on a set of test sequences.

Although this paper has focused on the dynamic analysis and search services of LASSO, since these are the most innovative and represent the main contribution of the platform, these are not intended to compete with, or replace the static analysis techniques employed by other platforms. On the contrary, the long term goal is to seamlessly integrate LASSO's dynamic analysis capabilities with the large-scale syntactic analysis and machine learning capabilities of existing platforms.

CRediT authorship contribution statement

Marcus Kessel: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Colin Atkinson:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Link is in article.

Appendix. LASSO TDS – LSL pipeline script

```
dataSource 'mavenCentral2023'
def interfaceSpec = '''Queue {
                        enqueue(Object)->Object
                        enqueue()->Object
                        peek()->Object
                        size()->int
                    }'''
study(name:'Queue-TDS') {
    action(name:'select', type:'Select') {
        abstraction('Queue') { // interface-driven code search
            queryForClasses interfaceSpec
            rows = 10
            excludeClassesByKeywords(['private', 'abstract'])
            excludeTestClasses()
            excludeInternalPkgs()
            filter 'complexity:[2 TO *]'
        }
    }
    action(name: 'rejectClones', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true

        dependsOn 'select'
        includeAbstractions 'Queue'
    }
    action(name:'filter',type:'ArenaExecute') { // test filter
        sequences = [
                'testQueueElements': sheet(p1: 'Queue', p2: 1, p3: 2) {
                    row '',    'create', '?p1'
                    row true,    'enqueue', 'A1', '?p2'
                    row true,    'enqueue', 'A1', '?p3'
                    row '?p2', 'peek',    'A1'
                    row 2,     'size',    'A1'
                    row '?p2', 'dequeue', 'A1'
                    row 1,     'size',    'A1'
                }
        ]
        maxAdaptations = 1 // how many adaptations to try
        features = ['cc'] // code coverage measurement

        dependsOn 'rejectClones'
        includeAbstractions 'Queue'
        profile('myTdsProfile') {
            scope('class') { type = 'class' }
            environment('jdk17') { image = 'maven:3.6.3-openjdk-17' }
        }

        whenAbstractionsReady() {
            def queue = abstractions['Queue']
            def expectedBehaviour = toOracle(srm(abstraction: queue).sequences)
            // returns a filtered SRM
            def matchesSrm = srm(abstraction: queue)
                    .systems // select all systems
                    .equalTo(expectedBehaviour) // functionally equivalent
            // continue pipeline with matched systems only
            queue.systems = matchesSrm.systems
        }
    }
    action(name:'rank', type:'Rank') { // rank based on two criteria
        strategy = 'HDS_SMOOP' // SOCORA ranking strategy
        criteria = ['IndexMeasurements.m_static_loc_td:MIN:1',
                    'cc.branch.total:MIN:2']
        dependsOn 'filter'
        includeAbstractions '*'
    }
}
```

Listing 2: LASSO TDS - Strict test-driven search pipeline in LSL including ranking preferences (last action)

References

Abdalkareem, R., Shihab, E., Rilling, J., 2017. On code reuse from StackOverflow: An exploratory study on android apps. Inf. Softw. Technol. 88, 148–158. http://dx.doi.org/10.1016/j.infsof.2017.04.005, URL: http://www.sciencedirect.com/science/article/pii/S0950584917303610.

Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C., 2018. A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51 (4), http://dx.doi.org/10.1145/3212695.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), http://dx.doi.org/10.1145/3290353.

Ammann, P., Offutt, J., 2016. Introduction to Software Testing. Cambridge University Press.

Bajracharya, S., Ossher, J., Lopes, C., 2014. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. Sci. Comput. Program. 79, 241–259. http://dx.doi.org/10.1016/j.scico.2012.04.008, URL: https://www.sciencedirect.com/science/article/pii/S016764231200072X. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 50:1–50:39. http://dx.doi.org/10.1145/3182657, URL: http://doi.acm.org/10.1145/3182657.

Barns, B., Bollinger, T., 1991. Making reuse cost-effective. IEEE Softw. 8 (1), 13–24. http://dx.doi.org/10.1109/52.62928.

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S., 2015. The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41 (5), 507–525. http://dx.doi.org/10.1109/TSE.2014.2372785.

Bauer, V., Vetro', A., 2016. Comparing reuse practices in two large software-producing companies. J. Syst. Softw. 117, 545–582. http://dx.doi.org/10.1016/j.jss.2016.03.067, URL: http://www.sciencedirect.com/science/article/pii/S0164121216300176.

Boettiger, C., 2015. An introduction to docker for reproducible research. SIGOPS Oper. Syst. Rev. 49 (1), 71–79. http://dx.doi.org/10.1145/2723872.2723882, URL: http://doi.acm.org/10.1145/2723872.2723882.

Brin, S., Page, L., 1998. The anatomy of a large-scale hypertextual web search engine. Comput. Netw. ISDN Syst. 30 (1), 107–117. http://dx.doi.org/10.1016/S0169-7552(98)00110-X, URL: https://www.sciencedirect.com/science/article/pii/S016975529800110X. Proceedings of the Seventh International World Wide Web Conference.

Carpineto, C., Romano, G., 2012. A survey of automatic query expansion in information retrieval. ACM Comput. Surv. 44 (1), http://dx.doi.org/10.1145/2071389.2071390.

Chen, M., Tworek, J., Jun, H., et al., 2021. Evaluating large language models trained on code. arXiv:2107.03374.

Cordy, J.R., Roy, C.K., 2011. The NiCad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension. pp. 219–220. http://dx.doi.org/10.1109/ICPC.2011.26.

De Paula, A.C., Guerra, E., Lopes, C.V., Sajnani, H., Lazzarini Lemos, O.A., 2016. An exploratory study of interface redundancy in code repositories. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 107–116. http://dx.doi.org/10.1109/SCAM.2016.31.

Diamantopoulos, T., Thomopoulos, K., Symeonidis, A., 2016. QualBoa: Reusability-aware recommendations of source code components. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories. MSR, pp. 488–491.

Dietrich, J., Schole, H., Sui, L., Tempero, E., 2017. Xcorpus – an executable corpus of java programs. J. Object Technol. 16 (4), 1:1–24. http://dx.doi.org/10.5381/jot.2017.16.4.a1.

Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435. http://dx.doi.org/10.1007/s10664-005-3861-2.

Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N., 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 2013 35th International Conference on Software Engineering. ICSE, pp. 422–431. http://dx.doi.org/10.1109/ICSE.2013.6606588.

Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N., 2015. Boa: Ultra-large-scale software repository and source-code mining. ACM Trans. Softw. Eng. Methodol. 25 (1), http://dx.doi.org/10.1145/2803171.

Fowler, M., 2006. Continuous integration. URL: https://www.martinfowler.com/articles/continuousIntegration.html.

Frakes, W.B., Kyo Kang, 2005. Software reuse research: status and future. IEEE Trans. Softw. Eng. 31 (7), 529–536. http://dx.doi.org/10.1109/TSE.2005.85.

Fraser, G., Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using EvoSuite. ACM Trans. Softw. Eng. Methodol. 24 (2), http://dx.doi.org/10.1145/2685612.

Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T., 1987. The vocabulary problem in human-system communication. Commun. ACM 30 (11), 964–971. http://dx.doi.org/10.1145/32206.32212.

GitHub, 2022. Octoverse 2022. URL: https://octoverse.github.com/. (Accessed 01 December 2022).

Grazia, L.D., Pradel, M., 2022. Code search: A survey of techniques for finding code. ACM Comput. Surv. http://dx.doi.org/10.1145/3565971.

Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering. ICSE, pp. 933–944. http://dx.doi.org/10.1145/3180155.3180167.

Hummel, O., 2008. Semantic Component Retrieval in Software Engineering (Ph.D. thesis). Universität Mannheim.

Hummel, O., Atkinson, C., 2006. Using the web as a reuse repository. In: Morisio, M. (Ed.), Reuse of Off-the-Shelf Components. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 298–311.

Hummel, O., Janjic, W., Atkinson, C., 2008. Code conjurer: Pulling reusable software out of thin air. IEEE Softw. 25 (5), 45–52. http://dx.doi.org/10.1109/MS.2008.110.

Inoue, K., Miyamoto, Y., German, D.M., Ishio, T., 2020. Code clone matching: A practical and effective approach to find code snippets. http://dx.doi.org/10.48550/ARXIV.2003.05615, URL: https://arxiv.org/abs/2003.05615.

Joachims, T., Granka, L., Pan, B., Hembrooke, H., Gay, G., 2017. Accurately interpreting clickthrough data as implicit feedback. SIGIR Forum 51 (1), 4–11. http://dx.doi.org/10.1145/3130332.3130334.

Johnston, W.M., Hanna, J.R.P., Millar, R.J., 2004. Advances in dataflow programming languages. ACM Comput. Surv. 36 (1), 1–34. http://dx.doi.org/10.1145/1013208.1013209.

Josefsson, S., 2006. The base16, base32, and base64 Data Encodings. RFC 4648, RFC Editor.

JUnit, 2022. JUnit. URL: https://junit.org/.

Kessel, M., 2023. LASSO - an Observatorium for the Dynamic Selection, Analysis and Comparison of Software (Ph.D. thesis). Mannheim, URL: https://madoc.bib.uni-mannheim.de/64107/.

Kessel, M., Atkinson, C., 2016. Ranking software components for reuse based on non-functional properties. Inf. Syst. Front. 18 (5), 825–853. http://dx.doi.org/10.1007/s10796-016-9685-3.

Kessel, M., Atkinson, C., 2018. Integrating reuse into the rapid, continuous software engineering cycle through test-driven search. In: 2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering. pp. 8–11.

Kessel, M., Atkinson, C., 2019. A platform for diversity-driven test amplification. In: Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. In: A-TEST 2019, Association for Computing Machinery, New York, NY, USA, pp. 35–41. http://dx.doi.org/10.1145/3340433.3342825.

Kessel, M., Atkinson, C., 2019a. Automatically curated data sets. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 56–61. http://dx.doi.org/10.1109/SCAM.2019.00015.

Kessel, M., Atkinson, C., 2019b. On the efficacy of dynamic behavior comparison for judging functional equivalence. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 193–203. http://dx.doi.org/10.1109/SCAM.2019.00030.

Kessel, M., Atkinson, C., 2022. Diversity-driven unit test generation. J. Syst. Softw. 193, http://dx.doi.org/10.1016/j.jss.2022.111442, URL: https://www.sciencedirect.com/science/article/pii/S0164121222001406.

Kessel, M., Atkinson, C., 2023. Data set: Code search engines for the next generation. http://dx.doi.org/10.5281/zenodo.8398748.

Kessel, M., Atkinson, C., 2024. Promoting open science in test-driven software experiments. J. Syst. Softw. 212, 111971. http://dx.doi.org/10.1016/j.jss.2024.111971, URL: https://www.sciencedirect.com/science/article/pii/S0164121224000141.

Kim, K., Kim, D., Bissyandé, T.F., Choi, E., Li, L., Klein, J., Traon, Y.L., 2018. Facoy: A code-to-code search engine. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, ACM, New York, NY, USA, pp. 946–957. http://dx.doi.org/10.1145/3180155.3180187, URL: http://doi.acm.org/10.1145/3180155.3180187.

Koschke, R., 2007. Survey of research on software clones. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Krueger, C.W., 1992. Software reuse. ACM Comput. Surv. 24 (2), 131–183. http://dx.doi.org/10.1145/130844.130856.

Lazzarini Lemos, O.A., Bajracharya, S.K., Ossher, J., 2007. CodeGenie: A tool for test-driven source code search. In: Companion To the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. OOPSLA '07, Association for Computing Machinery, New York, NY, USA, pp. 917–918. http://dx.doi.org/10.1145/1297846.1297944.

Lazzarini Lemos, O.A., Bajracharya, S., Ossher, J., Masiero, P.C., Lopes, C., 2009. Applying test-driven code search to the reuse of auxiliary functionality. In: Proceedings of the 2009 ACM Symposium on Applied Computing. SAC '09, Association for Computing Machinery, New York, NY, USA, pp. 476–482. http://dx.doi.org/10.1145/1529282.1529384.

Lemos, O.A.L., Bajracharya, S.K., Ossher, J., Morla, R.S., Masiero, P.C., Baldi, P., Lopes, C.V., 2007. CodeGenie: Using test-cases to search and reuse source code. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. ASE '07, ACM, New York, NY, USA, pp. 525–526. http://dx.doi.org/10.1145/1321631.1321726, URL: http://doi.acm.org/10.1145/1321631.1321726.

Lerner, J., Tirole, J., 2001. The open source movement: Key research questions. Eur. Econ. Rev. 45 (4), 819–826. http://dx.doi.org/10.1016/S0014-2921(01)00124-6, URL: https://www.sciencedirect.com/science/article/pii/S0014292101001246. 15th Annual Congress of the European Economic Association.

Li, Y., Tan, T., Xue, J., 2019. Understanding and analyzing java reflection. ACM Trans. Softw. Eng. Methodol. 28 (2), http://dx.doi.org/10.1145/3295739.

Lilis, Y., Savidis, A., 2019. A survey of metaprogramming languages. ACM Comput. Surv. 52 (6), http://dx.doi.org/10.1145/3354584.

Long, J., 2001. Software reuse antipatterns. SIGSOFT Softw. Eng. Not. 26 (4), 68–76. http://dx.doi.org/10.1145/505482.505492.

Lopes, C.V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., Vitek, J., 2017. Déjàvu: A map of code duplicates on GitHub. Proc. ACM Program. Lang. 1 (OOPSLA), http://dx.doi.org/10.1145/3133908.

Martins, P., Achar, R., Lopes, C.V., 2018a. 50K-C: A dataset of compilable, and compiled, java projects. In: Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18, ACM, New York, NY, USA, pp. 1–5. http://dx.doi.org/10.1145/3196398.3196450, URL: http://doi.acm.org/10.1145/3196398.3196450.

Martins, P., Achar, R., Lopes, C.V., 2018b. 50K-C: A dataset of compilable, and compiled, java projects. In: Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18, Association for Computing Machinery, New York, NY, USA, pp. 1–5. http://dx.doi.org/10.1145/3196398.3196450.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

Mili, H., Mili, F., Mili, A., 1995. Reusing software: issues and research directions. IEEE Trans. Softw. Eng. 21 (6), 528–562. http://dx.doi.org/10.1109/32.391379.

Mili, A., Mili, R., Mittermeir, R., 1998. A survey of software reuse libraries. Ann. Softw. Eng. 5 (1), 349–414. http://dx.doi.org/10.1023/A:1018964121953.

Nie, L., Jiang, H., Ren, Z., Sun, Z., Li, X., 2016. Query expansion based on crowd knowledge for code search. IEEE Trans. Serv. Comput. 9 (5), 771–783. http://dx.doi.org/10.1109/TSC.2016.2560165.

Palsberg, J., Lopes, C.V., 2018. NJR: A normalized java resource. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. ISSTA '18, Association for Computing Machinery, New York, NY, USA, pp. 100–106. http://dx.doi.org/10.1145/3236454.3236501.

Podgurski, A., Pierce, L., 1992. Behavior sampling: A technique for automated retrieval of reusable components. In: Proceedings of the 14th International Conference on Software Engineering. ICSE '92, ACM, New York, NY, USA, pp. 349–361. http://dx.doi.org/10.1145/143062.143152, URL: http://doi.acm.org/10.1145/143062.143152.

Podgurski, A., Pierce, L., 1993. Retrieving reusable software by sampling behavior. ACM Trans. Softw. Eng. Methodol. 2 (3), 286–303. http://dx.doi.org/10.1145/152388.152392, URL: http://doi.acm.org/10.1145/152388.152392.

Rahman, M.M., Barson, J., Paul, S., Kayani, J., Lois, F.A., Quezada, S.F., Parnin, C., Stolee, K.T., Ray, B., 2018. Evaluating how developers use general-purpose web-search for code retrieval. In: Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18, Association for Computing Machinery, New York, NY, USA, pp. 465–475. http://dx.doi.org/10.1145/3196398.3196425.

Reiss, S.P., 2009. Semantics-based code search. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 243–253. http://dx.doi.org/10.1109/ICSE.2009.5070525.

Rice, H.G., 1953. Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. 74 (2), 358–366, URL: http://www.jstor.org/stable/1990888.

Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T., 2014. Recommendation Systems in Software Engineering. Springer.

Roy, C.K., Cordy, J.R., Koschke, R., 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. 74 (7), 470–495. http://dx.doi.org/10.1016/j.scico.2009.02.007, URL: https://www.sciencedirect.com/science/article/pii/S0167642309000367.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V., 2016. Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1157–1168.

Sim, S.E., Gallardo-Valencia, R.E., 2015. Finding Source Code on the Web for Remix and Reuse. Springer Publishing Company, Incorporated.

Sonatype, 2022. Maven Central. URL: http://search.maven.org.

Stolee, K.T., Elbaum, S., Dobos, D., 2014. Solving the search for source code. ACM Trans. Softw. Eng. Methodol. 23 (3), 26:1–26:45. http://dx.doi.org/10.1145/2581377, URL: http://doi.acm.org/10.1145/2581377.

Svajlenko, J., Roy, C.K., 2016. BigCloneEval: A clone detection tool evaluation framework with BigCloneBench. In: 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 596–600. http://dx.doi.org/10.1109/ICSME.2016.62.

Terra, R., Miranda, L.F., Valente, M.T., Bigonha, R.S., 2013. Qualitas.class corpus: A compiled version of the qualitas corpus. SIGSOFT Softw. Eng. Not. 38 (5), 1–4. http://dx.doi.org/10.1145/2507288.2507314, URL: http://doi.acm.org/10.1145/2507288.2507314.

The Apache Software Foundation, 2022a. Apache ignite. URL: https://ignite.apache.org/.

The Apache Software Foundation, 2022b. Apache maven project. URL: https://maven.apache.org.

The Apache Software Foundation, 2022c. Apache solr project. URL: https://solr.apache.org/.

The Apache Software Foundation, 2022d. Groovy - domain-specific languages. URL: http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html.

Wang, Y., Feng, Y., Martins, R., Kaushik, A., Dillig, I., Reiss, S.P., 2016. Hunter: Next-generation code reuse for Java. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, ACM, New York, NY, USA, pp. 1028–1032. http://dx.doi.org/10.1145/2950290.2983934, URL: http://doi.acm.org/10.1145/2950290.2983934.

Wang, K., Walker, T., Zheng, Z., 2009. PSkip: Estimating relevance ranking quality from web search clickthrough data. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '09, Association for Computing Machinery, New York, NY, USA, pp. 1355–1364. http://dx.doi.org/10.1145/1557019.1557164.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.

Yang, F., Mei, H., Li, K., 1999. Software reuse software component technology. Acta Electron. Sin. 27, 68–75.

Zaremski, A.M., Wing, J.M., 1995. Signature matching: A tool for using software libraries. ACM Trans. Softw. Eng. Methodol. 4 (2).