The Journal of Systems and Software 212 (2024) 111971

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Promoting open science in test-driven software experiments☆

Marcus Kessel *, Colin Atkinson

University of Mannheim, 68159 Mannheim, Germany

ARTICLE INFO

Keywords:
Software
Engineering
Empirical
Experimentation
Observation
Behavior
Reproducibility
Replication
Data structures
Open science
Large language models
Machine learning
Generative artificial intelligence
Benchmark
Language-to-code
HumanEval
Automation
Measurement

ABSTRACT

A core principle of open science is the clear, concise and accessible publication of empirical data, including "raw" observational data as well as processed results. However, in empirical software engineering there are no established standards (de jure or de facto) for representing and "opening" observations collected in test-driven software experiments — that is, experiments involving the execution of software subjects in controlled scenarios. Execution data is therefore usually represented in ad hoc ways, often making it abstruse and difficult to access without significant manual effort. In this paper we present new data structures designed to address this problem by clearly defining, correlating and representing the stimuli and responses used to execute software subjects in test-driven experiments. To demonstrate their utility, we show how they can be used to promote the repetition, replication and reproduction of experimental evaluations of AI-based code completion tools. We also show how the proposed data structures facilitate the incremental expansion of execution data sets, and thus promote their repurposing for new experiments addressing new research questions.

1. Introduction

Empirical studies have grown in importance in software engineering research over the last few years as leading software engineering conferences and journals have placed increased emphasis on the need for research claims to be supported by empirical results (Siegmund et al., 2015; ESE, 2023). Most recently, this trend has evolved to encompass the principles of "Open Science" which aims to facilitate transparency, collaboration, and accessibility in software engineering research by making research data, methods and findings publicly available (Méndez Fernández et al., 2019). Today, open science is actively promoted by most leading software engineering conferences and journals such as EMSE[1], and has spawned new research communities such as the "Empirical Software Engineering and Measurement" (ESEM) symposium (ESEM, 2023).

The basic motivation for this trend is broadly the same across all scientific and engineering disciplines — namely, making all elements of empirical studies available for other researchers to examine, use and build upon, so that research communities can reach consensus on their conclusions and more easily repeat, replicate and reproduce them. According to the ACM (ACM, 2023), repetition involves the original team reperforming an empirical study with the original setup, replication involves a different team reperforming an empirical study with the original setup, and reproduction involves a different team reperforming the empirical study with a different design and/or setup.

The fundamental principles and ingredients of open science are also basically the same across scientific disciplines, even though the terminology and the detailed breakdown of steps may differ. In the social science domain, Minocher et al. (2020) define four fundamental prerequisites for an empirical study to be fully reproducible – "(1) data recoverability, the availability of the involved data and other materials to attempt reproduction of analyses, (2) data usability, the completeness and clarity of data, when available, (3) analytical clarity, the adequacy of published reports for repeating analyses, and (4) agreement of reproduced results with published results". In their overview of open science in software engineering, Mendez et al. (2020) point out that the first of these, data recoverability, involves the dissemination

☆ Editor: Alexander Serebrenik.

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

[1] with its Open Science Initiative (https://emsejournal.github.io/open_science)

https://doi.org/10.1016/j.jss.2024.111971
Received 2 August 2023; Received in revised form 8 December 2023; Accepted 15 January 2024
Available online 12 March 2024

of manifold artifacts including (a) the data sets and observed results (*i.e.*, open data), (b) the artifacts used to support empirical studies such as infrastructure support and analysis scripts (*i.e.*, open materials), (c) the (prototype) realizations of the studied objects (*i.e.*, open source), and (d) the final descriptions of the findings (*i.e.*, open access).

Although the software engineering community has made progress in addressing many of these prerequisites for open science, Mendez et al. (2020) note that the community is "struggling in adapting open science to the particularities of our discipline". The area of empirical software engineering in which the challenges are arguably the most acute is "test-driven software experimentation", where software subjects are executed in controlled settings through carefully selected sets of stimuli (*i.e.*, tests) to observe their behavior. These observations are then used to infer properties about the executed software subjects and/or the tests, often to validate hypotheses about the tools and techniques that created and/or assembled them.

Test-driven Software Experiments (TDSEs) have an important role to play in empirical software engineering because of Rice's theorem, which states that "all non-trivial properties of programs are undecidable" (Rice, 1953). This means it is impossible to create general algorithms that can analytically determine certain important properties of software, such as its run-time semantics (a.k.a. as its run-time behavior or functionality) or its execution properties (*e.g.*, performance and resource usage). Such properties can therefore only be measured or estimated "dynamically" by executing the software in question using known and carefully selected tests.

There are many reasons why supporting open science is currently challenging in the context of test-driven software experimentation. Chief among them are —

1. *Data Recoverability*: to attain generalizable results, the object of an experiment must be applied to large numbers of software subjects. However, although it is relatively easy to retrieve code from online open source repositories, a large proportion of this software is not immediately executable for technical reasons such as incomplete build information, missing parts or other incompatibilities (Palsberg and Lopes, 2018). As a result, the subjects of test-driven software experiments are still today mainly "curated" (*i.e.*, collected and managed) by hand, which takes a lot of manual effort and usually means that executable software data sets (a.k.a. executable software corpora) quickly degrade over time due to a lack of maintenance (Dietrich et al., 2017; Kessel and Atkinson, 2019a). This not only directly impacts the data recoverability prerequisite of Minocher et al. it means the data (*i.e.*, software) is often not in a form that can be readily used to repeat, replicate or reproduce an experiment.
2. *Data Usability*: even if a software subject is executable, it may not be compatible with the test sets selected to stimulate it in a software experiment. In other words, the interface by which a software subject offers its functionality might not match the interface by which a test expects to invoke it. Therefore, most TDSEs performed to date have required a lot of "set up and adaptation" code, often written by hand, to make the subjects of the experiment compatible with the tests (Just et al., 2014; Palsberg and Lopes, 2018). This set up code is usually written in mainstream programming and/or scripting language, and is often poorly structured and documented (Nong et al., 2023), defeating the data usability prerequisite of Minocher et al.
3. *Analytic Clarity*: the tests used in TDSEs are almost always described using a mainstream unit testing platform such as JUnit (JUnit, 2022) which was designed for practical software testing applications not experimentation. Such "test descriptions" are written in a mainstream programming language and usually "bury" the actual stimuli of the subject software in extensive scaffolding code to generate instances and place it in a desired state, and testing frameworks like JUnit only issue messages about whether the responses to stimuli "passed" or "failed". They do not attempt to record the actual outputs returned by the subject software (*i.e.*, the observed responses) and provide little if any tracing data (*e.g.*, stack traces) to help third parties understand the conclusions. Moreover, any behavioral observations that are recorded are usually stored in ad hoc ways that bear little or no relationship to the test descriptions. Thus, the current approaches used to define the stimuli, and record the system responses, involved in TDSEs greatly reduce the analytic clarity of the produced data.
4. *Consensual Results*: because of the ad hoc and non-standard ways in which TDSE execution outcomes are stored or reported, a lot of dedicated analysis code usually has to be written to process the data and obtain the final results. As with the "set up and adaptation" code, this is often written in a mainstream programming language and is poorly documented. Moreover, since usually only aggregated results are disseminated, and all the intermediate steps and/or data required to reproduce them are omitted, the published results of TDSEs often cannot be fully understood without analyzing the code used to generate them. This is a major obstacle to establishing consensus about results and the derivable conclusions.

Even when all the artifacts involved in TDSEs are openly disseminated, therefore (which is often not the case (Nong et al., 2023)), it is still extremely difficult for third party researchers to gain an understanding of the exact conditions under which the software subjects were executed, what stimuli were actually used to invoke them, what resulting behavior was observed, and how these observations were processed. Researchers performing TDSEs therefore face significant challenges in fulfilling the principles and expectations of open science, and these challenges increase dramatically as the scale of experiments is increased to improve statistical power and generalizability (cf. Wohlin et al., 2012).

This paper is based on the premise that to address these problems and make TDSEs more amenable to open science, the software engineering research community needs (a) new data structures for clearly and simply representing software tests and the expected and/or actual responses of software subjects, (b) new data structures for systematically storing and analyzing the many stimulus/response pairs that are generated in TDSEs, (c) new domain specific languages for applying these data structures and defining the high-level steps involved in study pipelines, and (d) a dedicated platform to execute such pipelines, automate the executions involved in TDSEs and "open" the resulting data structures according to the principles of open science.

The focus of the paper is on presenting the data structures offered by our LASSO (Large-Scale Software Observatorium) platform[2] for addressing (a) and (b). For space reasons, the paper is unable to provide much detail about (c), the so-called LASSO Scripting Language used to represent study pipelines, and since descriptions of LASSO have been published in previous papers (Kessel and Atkinson, 2019; Kessel and Atkinson, 2019b; Kessel and Atkinson, 2022), the paper provides only a very high level overview of (d). A detailed description of LASSO is available in the first author's PhD thesis (Kessel, 2023).

To demonstrate their utility for promoting open science in TDSE, we show how the data structures can be used to represent and publish the results of three TDSE-based evaluations of AI-based code generation tools powered by *Large Language Models* (Austin et al., 2021), a technology that is currently receiving a great deal of attention due to chatbots like ChatGPT that use the popular transformer architecture (Vaswani et al., 2017). More specifically, we show how the proposed data structures can be used to concisely and transparently —

[2] The platform and its project are freely available on GitHub: https://softwareobservatorium.github.io/

1. *replicate* a recent high-profile experiment evaluating the effectiveness of AI-based code completion tools using a well-known benchmarking data set (Chen et al., 2021; Cassano et al., 2023b),
2. *reproduce* the above experiment using an enhanced experimental design,
3. *repurpose* and extend the data structures from the aforementioned replication and reproduction of the experiment to answer new research questions.

The rest of the paper is structured as follows. We first present the background to test-driven software experimentation in Section 2 and introduce the terminology we use in the rest of the paper. Section 3 then presents the proposed data structures. This is followed by a description of three TDSEs used to demonstrate the benefits of these data structures, the experimental setup used to support these TDSEs, and how the concepts/subjects in the replicated experiment are adapted to LASSO's platform. The three sections that follow then present each of the TDSEs which, in turn, replicate, reproduce and repurpose the HumanEval benchmarking experiment described in Cassano et al. (2023b). Finally, Section 8 discusses the strengths and weaknesses of the approach and concludes with some closing remarks in Section 9.

2. Background and terminology

The terminology used in empirical research in general, and empirical software engineering in particular, is sometimes confusing and often used inconsistently (Wohlin, 2021). This section therefore clarifies the terminology used in the rest of the paper and positions the contributions of this paper in the field of empirical software engineering.

2.1. Test-driven software experimentation

Generally speaking, the notion of empirical software engineering covers all methods that use data-driven and statistically-based techniques to derive new knowledge about software, software engineering and related technologies. These range from methods originating in social sciences (such as systematic literature reviews, surveys, action research and questionnaires) and data mining techniques (such as software repository analysis and meta-analysis) to experimental techniques such as case studies, quasi experiments and controlled experiments (Wohlin et al., 2003, 2012).

In their paper "Empirical research in software engineering - a literature survey", Zhang et al. (2018) identify three basic purposes for empirical research in this field – "exploratory research", which aims to discover new software-related phenomena, generate hypotheses and gain a broader understanding of things of interest (*e.g.*, what are possible coverage goals in automated test generation? Fraser and Arcuri, 2012), "explanatory research", which aims to interpret such phenomena, establish causal relationships between them and provide clear explanations (*e.g.*, what is the impact of automated unit test generation on revealing real faults? Shamshiri et al., 2015), and "technical validations", which aim to validate or disprove hypotheses about methods, tools and techniques applied to software (*e.g.*, does tool X generate tests that achieve higher coverage than tool Y? Fraser and Arcuri, 2014). However, these categories have to be used with care, since particular experiments may belong to more than one category, or none at all (in which case they are classified as "other").

Experimentation is one of the most widely used methods in empirical software engineering, as it is in most scientific disciplines. The canonical type of experiment in science is the controlled experiment, where one variable, the factor, of the object under study is carefully manipulated, and other dependent variables are measured to observe the suspected effects and establish cause-and-effect relationships in a systematic manner (Basili et al., 1986; Wohlin et al., 2012). The object is applied to multiple experimental units, or subjects, through which the dependent variables are controlled. For instance, in a medical study testing the effectiveness of a new drug, the individual patients would be the experimental units, while in agricultural research studying the growth of plants under different conditions, each plant or plot of land might be considered an experimental unit.

In experimental software engineering, the experimental units involve one or more software engineering products, such as requirements, design artifacts or software (*i.e.*, code). For example, in an experiment to evaluate the effectiveness of an automated program repair tool (*i.e.*, to find solutions to software defects automatically Monperrus, 2018), the experimental units (*i.e.*, subjects) would be examples of code components with static defects (*e.g.*, classes or methods) to which the tool (*i.e.*, the object) is applied. The evaluation would then involve determining, for each experimental unit, whether the proposed software solution has (a) indeed fixed the defect, and (b) has the same semantics (*i.e.*, behavior) as the original. If the subject code components are very small (which can of course be arranged in a controlled experiment) both (a) and (b) can be established by "static" analysis techniques, such as symbolic execution. However, for non-trivial software subjects, static analysis techniques are often not applicable, so the only way to address (b) and gain confidence that the refactored code solution has the same behavior as the original is to execute it, which predicates the availability of suitable tests. Generally speaking, any software analysis technique that involves the execution of the subject software over some inputs is referred to as a dynamic technique, while any analysis technique that does not involve the execution of the subject software, but examines its code, is referred to as static (Ernst, 2003).

In this paper, therefore, we define a test-driven software experiment (TDSE) as an experiment in which the experimental unit involves software (*i.e.*, code) that is executed under controlled conditions by means of one or more tests. Although they are executed during a TDSE, the tests are usually not regarded as being part of the experimental unit, because they are only a means to an end (*i.e.*, the execution of the software in the experimental unit), not the end themselves. However, when the tool or algorithm under investigation in the experiment generates or manipulates the tests, the tests themselves become part of the experimental unit and thus become subjects in the experiment. For example, in an experiment to judge the effectiveness of a test generation tool, multiple tests generated by the tool have to be applied to multiple implementations of different software systems. In this case, both the tests, and the examples of software functionality (*e.g.*, classes or methods) are part of the experimental unit to be replicated multiple times and are thus referred to as "subject software". The focus of an experimental unit is usually not just an individual software component, therefore, but a collection of related software components which either test or implement a "functional abstraction", such as a class representing a stack, a method for sorting lists of elements or a method for calculating the greatest common denominator of two integers. For example, if the test-generation TDSE mentioned above were to use mutation score as one of the test quality metrics, the experimental unit would have to include multiple implementations of a given functional abstraction — one "correct" implementation and multiple, intentionally-incorrect implementations (*i.e.*, mutants) to establish how many are killed by the generated test set (cf. Andrews et al., 2005; Ammann and Offutt, 2016).

2.2. Ultra large-scale TDSEs

Although TDSEs have played an important role in empirical software engineering for many years, they have traditionally been performed only on a small scale, at least relative to studies and experiments that are exclusively based on static techniques. In the early days of empirical software engineering, finding sufficient numbers of software components to serve as experimental subjects was a significant problem, but since the exponential growth of large, online software repositories like GitHub, obtaining large quantities of open source software is no longer a major issue. This has spurred the growth of large, heterogeneous software data sets (sometimes called corpora)

Purpose/ Design / Team	Same Purpose (i.e., RQs)		Different Purpose (i.e., RQs)
	Same Design	Different Design	
Original Team	**Repetition:** • same purpose • original team • same (logical) design	**Reproduction:** • same purpose • original or independent team • different (logical) design	**Repurposing:** • different purpose (i.e., RQs) • original or independent team • different (logical) design • performed in a different way
Different Team	**Replication:** • same purpose • independent team • same (logical) design		

Fig. 1. The four experiment (re)use cases based on ACM's "Artifact Review and Badging" policy (ACM, 2023).

such as GHTorrent (Gousios, 2013), "Public Git Archive" (Markovtsev and Long, 2018), CodeDJ (Maj et al., 2021), World of Code (WoC) (Ma et al., 2021) and "The Stack" (Kocetkov et al., 2022). These offer researchers convenient access to version control data that link query development histories retrieved from version control systems like Git (*e.g.*, commits, tags and social data like developers etc.) to facilitate mining activities and empirical studies.

Several specialized platforms for "ultra large scale" static analysis of software have also emerged over the last decade. Perhaps the most prominent example is the Boa platform, which makes a huge repository of software components analyzable in an abstract way through a dedicated, high-level, domain specific language. Boa supports arbitrary queries over the abstract syntax (tree) of code units and allows new analysis capabilities to be added by users (*e.g.*, to power recommender services (Diamantopoulos et al., 2016). Similarly, SourcererCC (Sajnani et al., 2016), an extension to the Sourcerer platform Bajracharya et al., 2014), provides services that support syntactic code clone detection on a very large scale (*e.g.*, Lopes et al., 2017).

The limitation of these initiatives to support empirical software studies at the "ultra-large" scale is that they do not accommodate dynamic (*i.e.*, test-driven) analysis techniques. In other words, they do not address the high levels of manual effort still needed to perform TDSEs described in Section 1. The Large-Scale Software Observatorium (LASSO), which hosts the data structures proposed in this paper, was developed to fill this gap by (semi)-automating test-driven software experiments at an ultra-large scale (Kessel, 2023). The platform not only aims to support the dynamic (*i.e.*, run-time) algorithms and workflow steps involved in TDSEs, but also offers a large corpus of readily executable software. The main goal of LASSO, therefore, is to complement static analysis platforms such as Boa by dramatically reducing the manual effort involved in performing the dynamic elements of ultra-large TDSEs.

2.3. Reinforcing and reusing TDSEs

One of the key goals of open science is to allow the knowledge and insights gained from empirical studies to be reinforced and reused. The generic term "reproduction" is often used to capture this goal, where the reproduction of an experiment involves some kind of reperformance of the study. However, ACM's "Artifact Review and Badging" policy (ACM, 2023) assigns a more precise meaning to this term in the context of experimentation and distinguishes it from two other ways of reinforcing experiments. These are summarized in the gray part of Fig. 1 —

- *repetition* is where an experiment is repeated by the original team that performed it, using the same basic design. The only things that change are the details of how the experiment is performed, for example using a more powerful computing platform. More subjects may also be analyzed to increase statistical power.
- *replication* is conceptually similar to repetition except that the reperformance of the experiment is carried out by a different team, independent of the team that performed the first incarnation of the experiment. The practical difference is significant, however, since it is predicated on all the data, materials and descriptions from the first incarnation of the experiment being accessible, understandable, complete and clearly communicated.
- *reproduction* differs from repetition and replication in that the logical design of the experiment is also changed in some way. For example, the metrics or algorithms used to measure the properties of interest may differ, but the underlying research questions the new experiment is designed to answer are the same. It is conceptually unimportant whether the original team or the independent team performs the new experiment, although the aforementioned practical differences remain (see Fig. 1).

A second major goal of LASSO, and in particular the data structures presented in this paper, is to simplify and promote the repetition, replication and reproduction of TDSEs according to the ACM's definition of these terms. In addition, LASSO supports another important use case — the reuse of the observational data gathered in a TDSE to help answer new research questions. This goes beyond the first three use cases by essentially repurposing data gathered in one TDSE to amplify the results of another TDSE.

We therefore extended the ACM's set of use cases with a fourth case, shown on the right-hand side of Fig. 1, which we refer to as repurposing —

- *repurposing* differs from the original three reuse cases in that the research questions the original experiment was designed to investigate are changed or extended. In other words, when the data created in an experiment are "repurposed", they are reused in a new (usually larger) experiment to investigate new research questions, and thus essentially form part of a new experiment. This is a powerful new kind of use case, since it means that the data used in TDSEs can be "grown" incrementally and reused for new purpose over time.

To demonstrate the utility of LASSO's data structures for test-driven software experimentation, in Sections 5 to 7 we show how they can be used to support three of the four use cases described above in the evaluation of three AI-based code completion tools. More specifically, we demonstrate (a) the replication of a previous published experiment,[3] (b) the reproduction of (a) to answer the same research question but with an enhanced way of determining functional correctness of the software subjects, and (c) the repurposing of the observational

[3] We are, of course, unable to demonstrate the repetition case since we are a different, independent team.

Functional Abstraction
name: GCD (Greatest Common Divisor)
Interface: *gcd(Input1:long,Input2:long)->long*

Stimulus Sequence Sheet

Signature: *testGCD(p1:Class, p2:Input1, p3:Input2)*

Body

	A	B	C	D	E
1		create	?p1		
2		gcd	A1	?p2	?p3

Actuation Sequence Sheets

testGCD(Imp1, 3, 7)

	A	B	C	D	E
1		create	Imp1		
2	1	gcd	A1	3	7

testGCD(Imp1, 10, 15)

	A	B	C	D	E
1		create	Imp1		
2	5	gcd	A1	10	15

testGCD(Imp1, 49, 14)

	A	B	C	D	E
1		create	Imp1		
2	7	gcd	A1	49	14

testGCD(Imp1, 144, 60)

	A	B	C	D	E
1		create	Imp1		
2	12	gcd	A1	144	60

Fig. 2. Example of a stimulus sequence sheet and actuation sequence sheets for the GCD abstraction.

data from (a) and (b), along with new data, to answer new research questions about the tools. The TDSE data from (a) is thus reused and extended in (b), and the data in (b) is extended and reused in (c).

3. Data structures for test-driven software experimentation

This section presents the new LASSO data structures designed to support open science in large-scale, test-driven software experimentation. Although these are core components of the LASSO platform, they are intended to be open and independently-usable with other platforms. The relationship is similar to that between a spreadsheet tool like Microsoft Excel and the "spreadsheet" data structure itself. While Excel played an important role in the development of spreadsheets, and its functionality is intimately tied to them, today spreadsheets are supported by a wide range tools.

3.1. Sequence sheet notation

To facilitate TDSEs it is necessary to (a) create executable descriptions of how the subject software is to be stimulated (*e.g.*, tests), and (b) record how the subject software responds to these stimuli. As mentioned in the introduction, current test definition approaches and languages do a poor job of tying these two aspects together. To address this problem, LASSO therefore offers a new approach for specifying sequences of stimuli of software components and recording their responses – the **Sequence Sheet Notation** (SSN) – which is both a data structure and a language. It is a language, since it supports the creation of stimuli descriptions (*i.e.*, tests) that can be understood by LASSO and used to invoke the software subjects in TDSEs, and it is a data structure, since it supports the persistent storage of those stimuli and the responses of the software subjects.

SSN does not support the control flow constructs of fully-blown programming languages such as Java (*e.g.*, loops and if statements), and thus is not a Turing-complete language. However, this is not necessary when writing tests, since each potential path through a test written using a full programming language like Java can be written as a separate "uni-path" test in SSN.[4] This has the advantage that the invocations of the methods of a software component are known before execution so that the responses can be recorded next to the stimuli that caused them. In LASSO, such stimuli/response pairs are referred to as **actuations**.

Sequence sheets have two facets (*i.e.*, can be viewed from two perspectives). The first is the so-called **body**, or white-box facet, which shows the details of the individual invocations of the stimulated software component that occur when the sheet is executed. The second is the so-called **signature**, or black-box facet, which shows the name of the sequence sheet and any parameters that it needs or returns when executed.

There are two basic kinds of sequence sheets – **stimulus sequence sheets** and **actuation sequence sheets**. The former kind only defines the invocations to be made on the software component under test when the sheet is executed (*i.e.*, it "encodes" the sequence ready for execution), while the latter kind augments this invocation information with a record of the responses. Therefore, since sequence sheets not only drive the run-time invocations of the subject software in a TDSE (like a program) but also record the results for future analysis (like a data structure), they blur the traditional boundary between static and dynamic structures (cf. Ernst, 2003). Fig. 2 shows an example of both kinds of sheets for a functional abstraction that is an experimental unit in the example TDSE presented in Sections 5 to 7.

The top left-hand part of the figure identifies the name of the abstraction, Greatest Common Divisor (or GCD for short), as well as its signature. This shows that the GCD operation takes in two parameters of type `long`, *input1* and *input2*, and returns another `long` value that is their greatest common divisor. The bottom left-hand part of the figure shows the signature and body of a parameterized stimulus sequence sheet for exercising the GCD operation. The signature gives the name of the sheet, `testGCD`, and indicates that it receives three parameters, *p1*, which is a class (*i.e.*, the component under test), as well as *p2* and *p3*, which are of type `long`.

The "spreadsheet" below the signature defines the sequence of invocations that are executed when the sequence sheet is executed. Using a tabular spreadsheet-like notation is useful since it allows the various elements of the sequence sheet to be easily referenced. The rows of a sequence sheet all represent invocations of methods in the interface of the component under test, and are executed sequentially from top to bottom. The columns, on the other hand, play three distinct roles. One of the columns identifies the names of the operations that are called in each invocation, in the case of Fig. 2 this is column B. The columns to

[4] Note that this does not rule out parameterized tests.

Functional Abstraction
name: Queue (first in, first out)
Interface: *Queue {*
enqueue(Integer)->boolean
dequeue()->Integer
size()->Integer }

Stimulus Sequence Sheet
testQueueElements(p1:Class)

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

Actuation Sequence Sheet
testQueueElements(QueueImp)

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

Fig. 3. Example of a stimulus sequence sheet and actuation sequence sheet for the stateful *Queue* abstraction.

the right of column B (*i.e.*, C, D and E) contain the input parameters to each invocation, while the column to the left of B (*i.e.*, A) contains the output parameters or response from the stimulated component when the sheet is executed. In the case of the stimulus sheet on the left of Fig. 2, there are no output values shown.

The right-hand side of the figure shows four different actuation sequence sheets generated from the stimulus sheet on the left-hand side by executing it on a concrete implementation of the GCD functional abstraction, Imp1. The signature of an actuation sheet describes an actual invocation of the stimulus sheet on a particular GCD implementation, including the actual input parameters. As shown in Fig. 2, the actual response of the GCD implementation to each invocation is stored in the first column of the actuation sheet.

In general, sequence sheets can have an unlimited number of output parameters, but when the components under test are written in a language like Java that only allows methods to return one result, only one output column is needed.[5] If desired, actuation sheets can be written by human oracles to define the desired behavior of a software component by providing the expected values in the output column(s). The first input parameter of a sequence sheet plays a special role. It is used to input the subject component to be executed as shown by the cell reference in the spreadsheet notation (*i.e.*, C1). The *create* method is a special method that creates an instance (*i.e.*, object) of the component (here a Java class) which is stored in cell A1. This abstract approach is used to set up the subject implementation, since it is not always possible to obtain an instance of a class in a regular way (*e.g.*, by using the "new" keyword in Java) when dealing with heterogeneous components retrieved from software repositories (*i.e.*, when adapters need to be created).

Note that stimulus sheets can theoretically contain an unlimited number of operation invocations (*i.e.*, rows), so it would be possible for all four sequences in the actuation sheets in Fig. 2 to be included in a single stimulus sheet, leading to a single actuation sheet when executed. However, when recording the results of multiple invocations of multiple implementations of the functionality in question (*e.g.*, in a TDSE), it is convenient to make each logical test separate. Note that sequence sheets can also include invocations of other components (*i.e.*, classes) such as utility or helper classes that support the testing process (*e.g.*, invocations to methods of Java's JDK), and the special create method mentioned above also supports input parameters, like constructors, do for instance creation.

[5] Note that some programming languages like Python support multiple output parameters.

3.1.1. Stateful abstractions

The sequence sheets shown in Fig. 2 are rather simple because the functional abstraction being stimulated is a single, stateless operation (GCD in this case). However, sequence sheets can be used to stimulate and record the behavior of stateful abstractions as well (*i.e.*, objects). In this case a test typically includes multiple method invocations. A simple example of the use of a sequence sheet to stimulate a stateful functional abstraction is shown in Fig. 3. The example is a simple queue with first in — first out (FIFO) semantics.

Fig. 3 is arranged in a similar way to Fig. 2. The top left part shows the name and interface of the functional abstraction, while the bottom left-hand part of the figure shows the signature and body of a parameterized stimulus sequence sheet for exercising the queue functional abstraction. The right-hand side of the figure shows the resulting actuation sheet generated from the stimulus sheet on the left-hand side by executing it on a concrete implementation of the queue abstraction (here the class *QueueImp*). As before, column A stores the output values returned by the method invocations. Although the sequence sheet notation described in this section can be used on its own to describe and record tests, in TDSEs it is designed to be used in conjunction with the data structure described in the next subsection.

3.2. Stimulus response matrix

Conducting controlled TDSEs on a large scale is challenging. For example, MultiPL-E's HumanEval benchmark experiment, which we use for demonstration purposes in Sections 5 to 7, involved the execution of 94800 (= 3 × 158 × 200) component-test pairs, since there were three LLM-based, code completion tools (*i.e.*, study objects), 158 coding problems (*i.e.*, functional abstractions), and 200 code completions per tool per problem (*i.e.*, subject software components). Note that this is a theoretical maximum, since some of the software components generated may not be executable (*e.g.*, due syntax issues etc.). Each of these pairs requires the subject component (*i.e.*, code) to be (1) represented, (2) executed, and (3) enriched (or linked) with test results including behavioral observations. All observations made at run-time (*i.e.*, measurements) also need to be stored and retrievable for later analysis to answer the research questions.

Since the execution logistics involved in such a large number of component-test pairs is overwhelming for humans to manage manually, a high degree of automation is necessary. Existing state-of-the-art approaches, however, use ad hoc techniques (typically using custom scripting) to address the representation problem, the execution logistics and the recording of observations. The authors of MultiPL-E, for instance, use custom Python scripting as a solution with an ad hoc representation of the execution process (*i.e.*, compilation, testing etc.) and linking of the results. The biggest drawback is that such "custom"

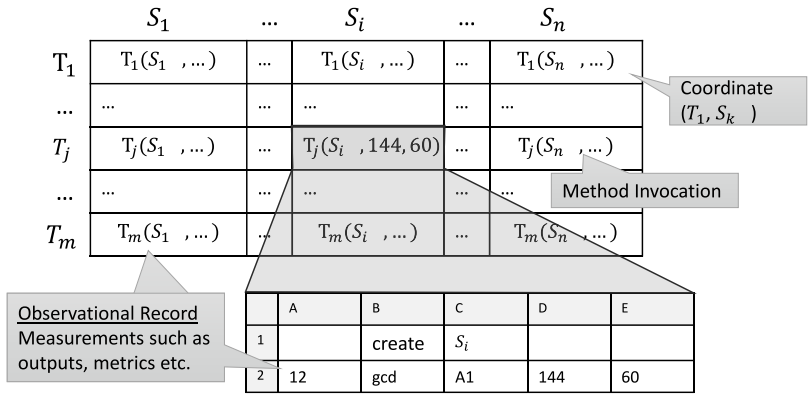

Fig. 4. SRM for the GCD abstraction.

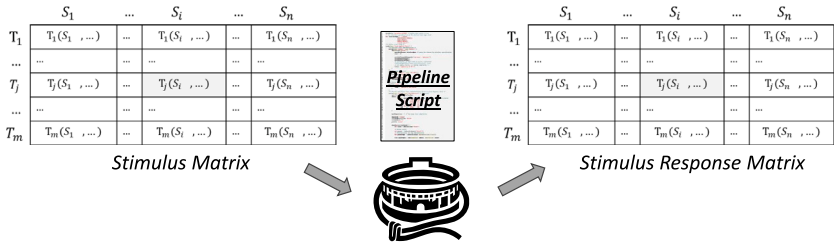

Fig. 5. Populating SRMs (recording observations).

approaches (*i.e.*, the experiment material) are not only hard to develop in the first place (*i.e.*, highly error-prone) and hard to maintain, they are also hard for third-parties to understand and set up in the same way as the original authors (thereby hindering replication).

To address these problems in a scalable manner, LASSO offers the new **Stimulus Response Matrix (SRM)** data structure. Whereas sequence sheets focus on the description of individual tests of individual software components, SRMs scale this up to the description of multiple tests on multiple components. SRMs essentially contain large numbers of actuation sheets corresponding to the invocation of multiple stimulus sheets on multiple implementations of the functional abstraction under investigation. As explained previously, it is assumed that all tests (*i.e.*, stimulus sheets) and all implementations invoke, or support, the same functional abstraction (with the same interface).

Fig. 4 illustrates an SRM for the GCD abstraction introduced in the previous subsection. The columns of this SRM correspond to the different software components under test, S_1 to S_n (*i.e.*, implementations of GCD), while the rows correspond to the tests (represented as sequence sheets), T_1 to T_m.

The figure highlights the synergy between SRMs and sequence sheets. The cell in row T_j and column S_i corresponds to the application of the stimulus sheet for T_j to the component implementation for column S_i. This application can be represented in a black box way by showing the signature of each invocation or in a white box way by showing the body of the actuation sheet as illustrated in Fig. 4. An SRM therefore provides a systematic and comprehensive data structure for storing all stimuli and responses for multiple tests and multiple implementations, thereby allowing their behavior to be easily compared. Moreover, measurements other than those related to functional behavior can also be stored and retrieved for each cell of the SRM, such as dynamic metrics.

The dichotomy between stimulus sequence sheets and actuation sequence sheets is mirrored in two basic variants of SRM. Since SRMs contain the results of the stimulations of alternative software components as well as the stimuli that lead to them, they play a similar "recording" role to actuation sheets. In fact, as shown in Fig. 4, they are essentially systematically organized ensembles of actuation sheets, and therefore could be referred to as actuation matrices. Stimulus sheets also have an analogue at the ensemble level which are referred to as **Stimulus Matrices (SMs)**. These are systematically organized ensembles of stimulus sheets which define what collections of stimuli have to be applied to what collections of software components. SMs therefore essentially represent inputs, while SRMs represent outputs. As shown in Fig. 5, we refer to the part of the LASSO platform that actually carries out the executions and observations needed to populate SRMs (*i.e.*, the test driver) as the **arena**. SMs are therefore inputs to the arena and SRMs are outputs.

SRMs can not only be serialized and stored for later data-driven analysis, they can also be re-executed under the same, or different, controlled conditions to enable repetition and replication. The important advantage of SRMs over existing approaches is that they allow the traditional "online" analysis of observational records to be shifted to classic "offline" analysis performed in a data-driven manner using sophisticated tooling (*e.g.*, as known in statistics and data science). Among other things, this means that judging functional correctness can be postponed to the analysis phase of the experimental process.

3.3. Stimulus response hypercube

SRMs provide the core data structure for storing "raw" observation data gathered when multiple software components are stimulated by multiple tests. The common thing that ties all elements in an SRM together is the functional abstraction. All the columns in an SRM are (supposed to be) implementations of that functional abstraction (*e.g.*, the GCD functional abstraction from HumanEval) and all the rows in the SRM are (supposed to be) tests of that functional abstraction. However, it is rarely the case that one SRM is sufficient to capture all the observational data needed in an experiment. Usually it is necessary to observe the behavior of multiple functional abstractions in order to be able to generalize the results (as mentioned above, HumanEval involved 158 functional abstractions), and if non-deterministic algorithms are involved such as the LLM models in the running example, multiple repeat executions are recommended to obtain adequate statistical power (Arcuri and Briand, 2014). In other words, in most TDSEs, SRMs

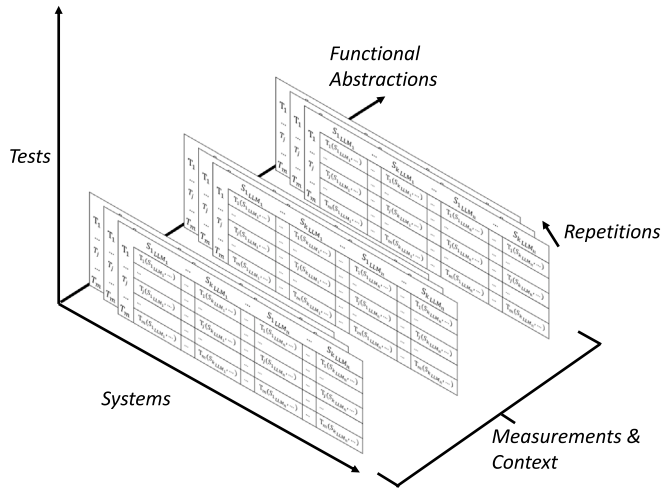

Fig. 6. Multi-dimensional arrangement of functional abstractions, systems, tests and repeat executions in an SRH.

represent individual experimental units which are repeated as many times as possible. There is therefore a need to tie the multiple related SRMs generated in a TDSE together within a higher order abstraction to allow users to understand the context in which each SRM was created and to navigate over them to extract and compare specific observations.

To fulfill this need and support the analysis phase of experiments, LASSO supports the notion of **Stimulus Response Hypercubes (SRHs)**. Like the hypercubes used in data warehousing applications (cf. online analytical processing, OLAP), SRHs are structures over multidimensional data that "bundle" the ensembles of SRMs created in a TDSE into a single logical hypercube. Each dimension provides the categorical context for navigation to, and understanding of, the individual measurements (*i.e.*, observations) stored in each cell of the underlying SRMs. Fig. 6 shows how an SRH can be used to provide a single, multidimensional view of an ensemble of related SRMs.

Each individual SRM in Fig. 6 is an ensemble of actuation sheets for a given functional abstraction. The "tests" and "systems" dimensions of the hypercube therefore correspond to the different tests and components in each SRM. Each closely grouped set of SRMs in the figure represents repeated executions of the contained tests and components in order to obtain sufficient statistical significance for non-deterministic algorithms, giving rise to a "repetition" dimension. And finally, for each functional abstraction studied in the experiment, there is a different set of SRM repetitions, giving rise to the "functional abstraction" dimension.

Fig. 6 shows an SRM ensemble organized in terms of four navigation dimensions. However, in general there can be more dimensions, depending on the number of independent variables in the study. For example, in the context of benchmarking code models (see Section 4), one variable that could be varied is the temperature. Other variables that either need to be fixed or varied in controlled ways include the run-time environment of software components (*e.g.*, the Java language version used). Another dimension arises when other kinds of measurements are made about an individual actuation recorded in a cell of an SRM, such as the execution time, the resource usage, or more complex data such as the run-time stack of called methods (*i.e.*, execution trace). These are all associated with one individual cell in an SRM, and can be accessed via an extra "measurement" or "record" dimension. The actuation sheets would therefore represent only one element within this "record" dimension. Finally, since actuation sheets are themselves two-dimensional data structures, they themselves contain further dimensions which a user could "drill down" to during data analysis.

In general, therefore, SRHs provide a multidimensional way for all the data collected in TDSEs, organized primarily through the notion of actuation sheets and SRMs, to be wrapped into a single, navigable and analyzable data structure. Moreover, SRHs allow the analysis of observation data to be conducted offline in mainstream data analytics platforms which offer rich ecosystems of tools and are supported by active user communities. Since these platforms all offer similar ways of manipulating tabular data in a cube-like way (operations like pivoting, for instance, cf. Caserta and Kimball, 2013), they all support natural ways of analyzing SRHs. In the example TDSEs presented in Sections 5 to 7, *R* (The R. Foundation, 2022) is used to analyze the SRHs.

3.4. LASSO platform

As mentioned in Section 1, the LASSO platform itself is not the focus of this paper. However, this subsection provides a brief overview of the platform's main features and the additional languages and capabilities it provides to support the use of the aforementioned data structures. Overall, LASSO offers a broad array of software code analysis services to facilitate two primary goals: (a) developing innovative or improved solutions for specific software engineering challenges (*e.g.*, diversity-driven test generation (Kessel and Atkinson, 2022) or automated curation of executable live datasets Kessel and Atkinson, 2019a), and (b) supporting test-driven software experimentation. Some of the TDSEs for which it has been used include investigations into functional equivalence in behavior sampling (*e.g.*, Kessel and Atkinson, 2019b) and test set quality assessment (*e.g.*, Kessel and Atkinson, 2022). What distinguishes LASSO's platform from related platforms for experimentation like Boa (Dyer et al., 2013, 2015) for mining software repositories, is its integration of large scale, dynamic analysis services with static, syntax-based services.

As shown in the high-level overview of the components of the platform in Fig. 7, LASSO basically has three core building blocks -

- an *executable corpus* of execution-ready software components,
- a *study pipeline engine* that executes TDSEs according to the steps described in study pipeline scripts,
- *software analytics* support for analyzing and validating the resulting data.

Executable corpus:. Curating data sets of executable software components is hard (Dietrich et al., 2017; Kessel and Atkinson, 2019a) (cf. Section 1). To support the (semi) automation of the curation process, LASSO maintains a systematically organized and extensible corpus of executable software. LASSO's corpus has been built from a vast collection of executable (Java) software components sourced from Maven Central (Sonatype, 2022) as well as other prominent software engineering repositories such as SF110 (Fraser and Arcuri, 2014). This large corpus of software components can be used to automatically provide custom data sets for specific TDSEs.

Study pipelines:. LASSO allows experimenters to write scripts to access, manipulate and analyze software from the aforementioned executable corpus based on their desired experimental process. Using this scripting language, called the *LASSO Scripting Language* (LSL), users can create multi-step pipelines to effectively analyze and retrieve software components from LASSO's software corpus. Pipelines typically include steps to —

1. retrieve software components with particular properties from the corpus (typically classes and methods),
2. define tests to stimulate the selected components using stimulation sheets,
3. load the selected components and tests into the arena as SMs and record the component's responses to the tests as SRMs,
4. export and analyze the collected data in a multidimensional way using the SRH structure to answer research questions using mainstream data analytics platforms.

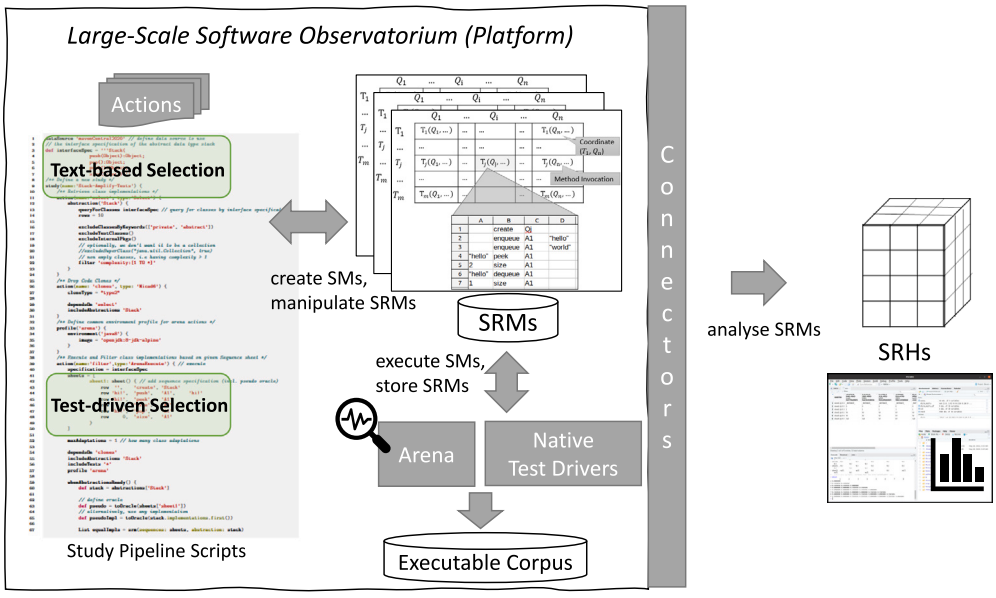

Fig. 7. LASSO platform - high-level overview of concepts.

Software analytics:. Once the experimental data has been collected, LASSO allows experimenters to handle and examine SRMs in two main ways. In the first method, observational data can be retrieved and modified during the execution of study pipelines. In the second method, experimenters can export the SRM data as SRHs into external data analytics tools that support advanced analysis options such as statistical testing.

3.4.1. Realization of SRMs and SRHs

This section discusses how the presented data structures are represented and stored, and subsequently how they can be analyzed in a classic, data-driven manner using statistical analytics.

SRMs:. SRMs and sequence sheets are tabular representations of data. Tabular data is easily shareable and can be presented in a standardized format, which in most data science technologies is referred to as the "data frame" format. This is crucial for open science, where transparency and reproducibility are key principles. Since tabular data is generally easy to interpret and understand, sharing data sets in a tabular format makes it easier for others to replicate experiments, verify results, and build upon existing research. Rows and columns provide a clear structure, and researchers can easily identify patterns, trends, and relationships within the data. This makes it straightforward for both experts and non-experts to comprehend and interpret the results.

Technically, the tabular format of SRMs provides a simple and intuitive way to work with observational data, since it allow users to perform various operations such as filtering, sorting, aggregating, and transforming data without requiring extensive programming knowledge or complex algorithms. It also supports high-dimensional analyses using abstractions such as hypercubes. Second, data frames are widespread, since they are commonly used in experimentation as well as in popular data science libraries such as Pandas (Python), Apache Spark DataFrames (Scala) and data.frame (R). Third, data frames can be used with various programming languages, making them a versatile tool for working with structured data across different platforms and applications. Finally, data frames also scale well, making them an ideal choice for representing SRMs that grow in size, and for managing and analyzing "big code" (cf. Allamanis et al., 2018) data sets. The popularity of data frames has led to increased interoperability between different tools, libraries, and platforms.

The realization approach used in LASSO was guided by the technology that powers its distributed platform, Apache Ignite (The Apache Software Foundation, 2022), which encourages data frames to be realized in a "relational" way. Ignite comes with an in-memory, key–value store that also acts as a relational database. LASSO uses the Entity-Attribute Value (EAV) data modeling technique, that allows the strict schema constraints imposed by relational databases to be relaxed, which in turn facilitates the flexible storage of observations in test-driven software experiments.

The database schema actually represents the structure of each cell in an SRM, containing its coordinates for navigability, its (measurement) type, its value and contextual information such measurement context. Each record effectively becomes a row in the table which allows SRMs to be stored in a sparse manner since only the cells that have content need to be represented. However, multiple measurements for the same coordinate can also be stored, which enables multidimensionality.

Note that since we use tabular representations, the actual storage is generally agnostic to the underlying database technology. One may just distribute SRMs as raw data frames in CSV files or in more sophisticated formats like Apache's Parquet.[6] Of course, the cells of an SRM can be represented in various ways in different database technologies. For example, they could be stored in distributed, large-scale databases like wide-column stores (*e.g.*, Apache Cassandra, Google BigTable or Amazon DynamoDB).

SRHs:. Unlike the storage options for SRMs, supporting the multidimensional analysis of SRMs through SRHs is dependent on the capabilities of the database technology and/or data analytics tool at hand. We used *R* to connect to LASSO's Ignite database and RJDBC[7] to manipulate the SRM records. *R* has a rich ecosystem of tools that offer cube operations to treat ensembles of SRMs as hypercubes. We used the tidyverse library[8] that offers cube operations (*e.g.*, pivoting) to enable rich analysis approaches such as behavior-based clustering (cf. Sections 5 to 7). Similar functionality can also be found in other popular tools such as the Python ecosystem of libraries (*e.g.*, Pandas etc.).

[6] https://parquet.apache.org/
[7] https://cran.r-project.org/web/packages/RJDBC/
[8] https://www.tidyverse.org/

Sequence sheets and object serialization:. As previously mentioned, in the LASSO platform tests are executed in the arena component, serving as the test driver for stimulus sheets. The internal implementation of the arena employs introspection (specifically, Java reflection) to run through each row within the test sequence outlined in a sheet. Using a tabular data structure allows both the inputs and outputs to be recorded systematically (based on indices that ease navigability) and stored in SRMs as actuation sheets. This makes it necessary to adopt a serialization schema that can store an object's state in a tabular way to facilitate comparisons between inputs and outputs. LASSO realizes the required serialization by representing objects as strings in the data types supported by JSON, which can then be compared for their equivalence or similarity.

4. Benchmarking large language models for code generation

As mentioned in the introduction, this paper makes two main contributions – (1) it describes three data structures designed to provide a simple and flexible basis for representing observations of run-time behavior, and (2) it showcases these data structures in the replication, reproduction and repurposing of TDSEs from the domain of AI code model benchmarking – the HumanEval benchmark for Java (Cassano et al., 2023b). The paper therefore essentially uses this benchmark as a running example to evolve a series of experiments and shows how observation data can be "opened" and reused according to open science practices. To set the scene for the detailed experiment descriptions, therefore, this section provides an overview of the running example. It also aligns the terminology used in the example benchmark TDSE with the terminology used in the LASSO platform.

4.1. Background

In order to demonstrate the utility of our proposed data structures for test-driven software experimentation, we apply them in a domain that is currently receiving a huge amount of attention due to the general excitement around Large Language Models (LLMs) in generative AI[9] – the "benchmarking" of LLM-powered programming tools (a.k.a. code models). Code models have emerged as a novel subfield of program synthesis (Gulwani et al., 2017) that capitalizes on the availability of big code and the recent advances in LLMs to potentially revolutionize the way software is developed. The technology basically aims to make software development faster and more efficient by accelerating various software engineering activities (*e.g.*, coding and test generation, documentation etc.), and making them more accessible to developers by lowering barriers (*e.g.*, by teaching coding activities through code explanations Finnie-Ansley et al., 2022).

Code models such as OpenAI's "Davinci" model (Chen et al., 2021) that powers Codex and GitHub Copilot are probabilistic LLM-based models that can create code from user specifications, typically called "prompts", which describe a coding problem in natural language. These models are trained on massive datasets of code as well as natural language text, allowing them to create new code from scratch or modify existing code to meet new requirements.

4.1.1. Code models

At the time of writing, quite a number of code models have been described in the literature. A recent overview can be found in Li et al. (2023), where code models are roughly classified by their accessibility (*i.e.*, as *open* when a model's weights are published, and *closed* when not). Three of the most well-known and widely-described code generation models are CodeGen, InCoder and Codex.

CodeGen (Nijkamp et al., 2023) is a massive parameter language model that has been extensively trained using a next-token prediction objective. The benchmark in this work focused on evaluating the multilingual version of CodeGen, which was initially trained on "The Pile" (Gao et al., 2020), a vast dataset containing mostly natural language text with approximately 8% GitHub-scraped code. To further enhance its performance and adaptability across various programming languages, the multilingual CodeGen model underwent fine-tuning on a carefully selected subset of six popular programming languages: C, C++, Go, Java, JavaScript, and Python.

InCoder (Fried et al., 2023) is a powerful parameter language model that has been extensively trained using a causal masking objective (Aghajanyan et al., 2022). The primary purpose of InCoder is to provide both code infilling and code completion services. The latter is evaluated in the running example. To achieve its high level of performance, InCoder was trained on an enormous dataset consisting of 159 GB of deduplicated (Allamanis, 2019) and filtered code from GitHub (approximately one-third of which is written in Python) and a further 57 GB of code snippets from StackOverflow.

Codex (Chen et al., 2021) is a powerful GPT-3 language model that has been specifically fine-tuned on code to generate high-quality, contextually-relevant program snippets and other software development artifacts. Until recently, it powered services like OpenAI Codex and GitHub Copilot. The benchmark focused on the more recent 175 GB parameter version of Codex (`code-davinci-002`), which has been trained on multiple languages. Unfortunately, details about the training set for this particular model are not publicly available (*i.e.*, it is a closed code model).

4.1.2. Benchmarking approaches

In order to evaluate the effectiveness of the various code models described above, and to compare them to other code delivery services such as code search engines, there is an urgent need for effective comparison platforms and approaches. At the time of writing, a variety of benchmarking approaches have been described in the literature. For the task of code generation in particular, one can roughly classify benchmarks according to whether the functional correctness of software components is established by —

- *textual similarity* metrics, originating from NLP practices, such as the *BLEU* metric (Papineni et al., 2002) where generated software components are compared to a reference component, or by
- *unit tests*, ideally hidden from the models, which determine whether software components deliver the expected behavior when executed.

Since textual similarity is only a weak indicator of functional correctness, test-driven evaluation of components based on representative tests is regarded as providing a stronger assessment of functional correctness (Chen et al., 2021; Li et al., 2022). Several benchmarks based on the test-driven evaluation of component behavior have been proposed. They all consist of diverse coding problems, of differing difficulty, mined from various internet sites that provide natural language descriptions of some desired functionality that components have to exhibit (*i.e.*, natural language-to-code). At the time of writing, such benchmarks typically contain coding problems for the Python language.

One of the highest profile benchmarks is the HumanEval benchmark for the Java language generated using MultiPL-E (Cassano et al., 2023b), and originally introduced by the authors of OpenAI's Davinci model (Chen et al., 2021) to assess the functional correctness of the Python code it generates. Apart from the HumanEval benchmark, other benchmarks of varying size and problem difficulty have been proposed, including MBPP (Austin et al., 2021), and Deepmind's CodeContests to assess AlphaCode (Li et al., 2022).

Although initiatives exist to offer systematic evaluation frameworks for code models by bundling existing benchmarks together (*e.g.*, Ben Allal et al., 2022), these do not attempt to solve the problem addressed in this paper as outlined in the introduction.

[9] especially breakthroughs stemming from the transformer architecture (Vaswani et al., 2017) that powers popular chatbots like OpenAI's ChatGPT

```java
// START PROMPT
import java.lang.reflect.*;
import org.javatuples.*;
import java.security.*;
import java.math.*;
import java.io.*;
import java.util.stream.*;
class Problem {
    // Return a greatest common divisor of two integers a and b
    // >>> greatestCommonDivisor((3l), (5l))
    // (1l)
    // >>> greatestCommonDivisor((25l), (15l))
    // (5l)
    public static long greatestCommonDivisor(long a, long b) {
        // generated by code LLM
    // END PROMPT
    }

    // START HIDDEN TESTS (hidden from LLM, standard Java using assert statement)
    public static void main(String[] args) {
        assert(greatestCommonDivisor((31), (71)) == (11));
        assert(greatestCommonDivisor((101), (151)) == (51));
        assert(greatestCommonDivisor((491), (141)) == (71));
        assert(greatestCommonDivisor((1441), (601)) == (121));
    }
    // END HIDDEN TESTS
}
```

Listing 1: Code prompt and tests of problem "HumanEval_13_greatest_common_divisor" (cf. HumanEval from MultiPL-E Cassano et al. (2023b))

4.2. HumanEval-J benchmark

This subsection provides an overview of the design of the MultiPL-E experiment that is the basis for the running example, and the HumanEval benchmark upon which it is based (further details can be found in Cassano et al., 2023b; Chen et al., 2021). Since the original HumanEval benchmark consists exclusively of Python coding problems, the MultiPL-E experiment translated them into 18 additional programming languages selected by their popularity and programming paradigms. The objective was to demonstrate the code models' performance across a range of widely used languages. Since the current version of LASSO focuses on the execution and analysis of code written in the Java programming language, we only reperformed the benchmark for the translated Java coding problems. In other words, we essentially replicated, reproduced and repurposed the experimental design used by MultiPL-E for the evaluation of the models' performance for Java code generation. To avoid confusion, in the remainder of the work, we refer to the Java version of HumanEval as HumanEval-J.

HumanEval-J contains a manually curated set of coding problems for which code generation models need to complete the code of single methods. For each coding problem, therefore, the model under evaluation is "prompted" to complete a given method body for a single coding task. The prompts for these task consists of three core ingredients (see Listing 1) —

1. a method signature including the name, input parameters and return parameter,
2. a code comment in natural language that describes the expected behavior of the method,
3. a list of example method invocations.

Based on the given set of problems, the performance of a model is then judged by the functional correctness of its code completions with respect to a supplied set of unit tests which are "hidden" from the models (*i.e.*, not the same as the set of example method invocations). Hiding the tests from the models is important in order to ensure the validity of the results, since it is important that the tests and coding problems are not "known" to the model (*i.e.*, that it was not trained on them). For this reason, the coding problems, including their tests, were manually curated from the internet, and ideally were not part of the data sets used to train the models.

The evaluation criterion used to assess whether code completions generated by the models are functionally correct is test-driven. Once a generated code completion for a problem passes *all* the provided unit tests that compare the expected behavior to the actual exhibited behavior, functional correctness is deemed to have been satisfied. Since exhaustive testing is infeasible in practice (Ammann and Offutt, 2016) and functional correctness is judged based on a small set of tests as part of the benchmark, it is assumed that these tests are "strong enough" to characterize the desired behavior for a given problem.

4.2.1. Code models (study objects)

Based on MultiPL-E's experimental design, we evaluated the performance of three state-of-the-art code generation models —

- OpenAI's Codex (`code-davinci-002`) (Chen et al., 2021),
- CodeGen (Nijkamp et al., 2023),
- InCoder (Fried et al., 2023).

For each problem, the original experiment sampled 200 code completions from each model based on a fixed "temperature" hyperparameter (0.2) which basically controls the predictability of code completions. More predictable completions correspond to lower temperature values, while more creative, less predictable completions correspond to higher temperature values. In other words, lower temperatures make code completions more deterministic, whereas higher temperatures make them less deterministic.

4.2.2. Code generation (study subjects)

MultiPL-E's attempt to translate the original 164 python coding problems into the Java programming language was successful 158 times. Note that we took the "reworded" data set variant of the problems tailored to other programming languages from (Cassano et al., 2023c). Since the (statistically sound) sampling of generated code from the models was costly and very time-consuming (cf. Cassano et al., 2023b), and the authors made the code completions accessible online (Cassano et al., 2023a), we did not resample the generated code in our running example.

4.3. Translation to LASSO platform

To replicate and reuse HumanEval-J, we translated the experimental design used in Cassano et al. (2023b) to the LASSO platform to exploit its experimentation and analysis services. LASSO is used as the platform for the running example since it realizes the data structures proposed in Section 3 and provides an execution platform to automate the operation phases of the experiments.

We basically carried out three main translation tasks to execute the experiments in the running example using LASSO —

- *Functional Abstractions*: we translated the 158 Java coding problems of HumanEval-J into LASSO's concept of functional abstractions,
- *Executable Corpus*: we migrated the code completions sampled by MultiPL-E into LASSO's single, underlying, executable corpus (for the purpose of code indexing and retrieval),
- *Study Pipelines*: we translated and encoded MultiPL-E's experimental design into study pipeline scripts (based on LASSO's scripting language) that serve as executable, reusable and shareable experimental design descriptions.

In the following subsections, we give an overview of each translation task, and streamline the terminology used in the running example.

4.3.1. Functional abstractions

To streamline the terminology used to refer to the software code units generated by the models in the experiment, we use the term **software component** to refer to the code completions of methods. Technically, these have to be encapsulated inside a Java class to make the methods invocable.[10]

As explained previously, a HumanEval-J coding problem describes the desired functionality in three basic ways (name, interface and a set of tests) without providing an executable reference component implementation. In LASSO, such "desired functionality" is referred to as a **functional abstraction**. HumanEval-J's description of each functional abstraction was therefore mapped into LASSO's format, including the translation of unit tests into stimulus sheets as explained in Section 3.

4.3.2. Executable corpus

To integrate the software components sampled from the three code models for HumanEval-J into LASSO, we set up a simple ETL process (extract, transform, load) to load them into the platform's executable code corpus. For the running example, we set up a study pipeline to extract each software component (*i.e.*, solution to a code completion task) represented as a Java class from the completions provided in Cassano et al. (2023a). Then we generated a Maven project for each class and loaded it into LASSO's executable corpus as a new data source by (1) indexing its source code (including static code measurements) for later retrieval purposes, and (2) uploading the Maven build artifact, including source and compiled code, to the underlying Maven repository for subsequent retrieval. Some software components generated by the models contained static faults such as syntax errors and were marked accordingly for later analysis.

4.3.3. Study pipelines

The last translation step was the construction of executable study scripts to conduct the various experiments contained in the running example. This was accomplished using the *LASSO Scripting Language*. To realize the running example, the experiments' processes were encoded into multiple study pipeline scripts which were used to answer the different research questions (see Appendix A). Unfortunately, for space reasons, we cannot provide further details about LASSO's scripting language and the pipelines used, so the reader is encouraged to consult (Kessel, 2023) for further details.

5. Experiment replication example

Having introduced the notion of sequence sheets, SRMs and SRHs, in the following three sections we show how they facilitate different forms of reuse of experimental data – experiment replication, reproduction and repurposing – in the context of the running example. This section focuses on the first form of reuse by presenting a *replication* of the HumanEval-J experiment. The material and data for this experiment, and the following experiments, is publicly available (Kessel and Atkinson, 2023).

The goal of the first experiment is to replicate the HumanEval-J experiment using the LASSO platform to answer the following research question (*i.e.*, the same one as the original) —

> **RQ 1.** *What is the Java code completion performance of the three code models (with temperature set to 0.2) as judged by functional correctness?*

[10] In classic object-oriented Java, a method cannot exist without a class.

To answer this research question, we first conducted the operation phase of the experiment using the study pipeline script in Listing 2 (Appendix A) in order to create all the required SMs and produce the corresponding SRMs. In the first analysis step, for each of the 158 functional abstractions of the benchmark, we retrieved all software components sampled from each model (up to 200 for each model) from LASSO's executable corpus. As part of the second analysis step (*i.e.*, running the arena test driver), we configured an SM for each functional abstraction by loading the tests of the functional abstraction as stimulation sheets. This resulted in 158 SMs being input to the arena to produce the corresponding SRMs as output. Finally, the resulting SRMs were stored for later analysis, including judging functional correctness based on the following evaluation criterion.

5.1. Judging functional correctness

In order to judge the functional correctness of the code completed by the models, we used the *pass@k* metric. This was originally proposed by Kulal et al. (2019) as part of their pseudocode-to-code approach, and was later evolved by Chen et al. (2021) to an unbiased estimator for obtaining statistically sound results. This is the metric used in the original MultiPL-E experiment being replicated.

Informally, *pass@1* represents the probability that one model-generated software component successfully passes all unit tests. Similarly, *pass@10* represents the probability that any one of 10 generated components successfully passes all unit tests. To answer the research question and compare the models' performance, *pass@1* (k=1) was calculated for each functional abstraction (*i.e.*, for each coding problem), and then averaged (using the arithmetic mean) over all problems using to the following equation —

$$pass@k(LLM_i, k) = \frac{1}{P}\sum_{p}^{P} pass@k(n, c, k) \tag{1}$$

where n is the total number of components generated by each model, c the number of correct components (according to the tests), and k is k in *pass@k* (set to 1).

5.2. Establishing functional correctness in SRMs

In order to measure *pass@k*, we first discuss how functional correctness is established through "offline" analysis of SRMs. Since SRMs store observational records as actuation sheets, it is possible to select the output columns of all executed components (*i.e.*, column A in the spreadsheet notation used by the SSN). This step alone, however, is not sufficient to perform the comparisons needed to answer the research question. What is missing is the oracle information for the functional abstraction. To provide oracle information, we introduce a "virtual" oracle component that defines the required responses of implementations of the GCD functional abstraction in the form of actuation sheets. The actual values are taken from the JUnit assertion statements in the HumanEval prompt for the GCD problem shown in Listing 1. The virtual oracle component is manifested as a new column in the SRM which contains the expected response for each stimulus. Once this oracle column has been added, the outputs can be compared to the recorded outputs (*i.e.*, actual responses) of the generated components to judge their functional correctness.

Fig. 8 demonstrates the change to the configuration of the SRM when an "oracle" O is introduced as a "virtual" component to enable comparisons. With the additional information, the responses of the components can now be pair-wise compared to the oracle responses. If there are no discrepancies between the output columns, a component is judged to be functionally correct.

Fig. 9 shows an excerpt of the distribution of responses from the SRH-based analysis of the SRM for the GCD abstraction for which four

	O	$S_{1_{LLM_1}}$	...	$S_{k_{LLM_1}}$	...	$S_{1_{LLM_n}}$	...	$S_{k_{LLM_n}}$
T_1	$T_1(O, \dots)$	$T_1(S_{1_{LLM_1}}, \dots)$	...	$T_1(S_{k_{LLM_1}}, \dots)$	...	$T_1(S_{1_{LLM_n}}, \dots)$	...	$T_1(S_{k_{LLM_n}}, \dots)$
...	...	...	...	...	...	...	...	...
T_j	$T_j(O, \dots)$	$T_j(S_{1_{LLM_1}}, \dots)$	...	$T_j(S_{k_{LLM_1}}, \dots)$	...	$T_j(S_{1_{LLM_n}}, \dots)$	...	$T_j(S_{k_{LLM_n}}, \dots)$
...	...	...	...	...	...	...	...	...
T_m	$T_m(O, \dots)$	$T_m(S_{1_{LLM_1}}, \dots)$	...	$T_m(S_{k_{LLM_1}}, \dots)$	...	$T_m(S_{1_{LLM_n}}, \dots)$	...	$T_m(S_{k_{LLM_n}}, \dots)$

Fig. 8. Adding a virtual oracle component O (first column) to the SRM to judge functional correctness.

Test@Invocation	Oracle	$CodeGen_1$	$CodeGen_2$	$CodeGen_3$	$CodeGen_4$
T_1@1	1	1	−7	1	1
T_2@1	5	5	−15	5	5
T_3@1	7	7	28	7	7
T_4@1	12	12	60	12	12

Fig. 9. Establishing functional correctness for the GCD abstraction (SRM contains 4 tests, and 5 components including the oracle and 4 components generated by the CodeGen model).

tests were defined. The filtered SRM includes the virtual oracle component (first column) and five components generated by the CodeGen model.

The oracle constructed from the abstraction is located in the first column of Fig. 9. When compared in a pair-wise manner to this first column, it is apparent that three components have result columns that are equivalent to the oracle's result column. They are therefore judged to be functionally correct. The component $CodeGen_2$, however, has different result values for all four tests and is thus judged to be functionally incorrect.

5.3. Results

Replicating the original HumanEval-J benchmark experiment using the LASSO platform allowed us to fully confirm the numbers reported by MultiPL-E's experiment in Cassano et al. (2023b). We were therefore able to successfully replicate their work, while benefiting from the effort-saving advantages of sequence sheets, SRMs and SRHs.

The SRH-based analysis of SRMs was performed using *R*, including the judgment of functional correctness and the computation of the *pass@1* metrics. Fig. 10 depicts the resulting average *pass@1* measures as a bar plot. The average *pass@1* measure was 0.222 for CodeGen, 0.416 for Davinci (Codex) and 0.0918 for InCoder. So, as reported in Cassano et al. (2023b) for the replication of Chen et al.'s results (Chen et al., 2021), we are able to confirm that Davinci is the best model for the language-to-code task for Java components.

When traditional unit testing practices are used to perform TDSEs, such as in MultiPL-E's experimental design, the aforementioned analyses can only be performed at run-time when the components are actually executed and compared using assertion statements. The obtained responses for the different code completions are therefore actually hidden from the experimenter. In contrast, as demonstrated by our replication, the use of SRMs and SRHs significantly increases actuation transparency in a TDSE, since behavioral observations are recorded and stored. This gives experimenters the ability to analyze the behavior in a purely offline, data-driven manner. SRMs, therefore, facilitate the fine-grained analysis of exhibited behavior by, for instance, identifying (a) functional similarity (subset of tests passed), (b) functional equivalence (subsets of components agreeing on certain responses) across the software components of an abstraction, and (c) the number of behavioral clusters across abstractions and models.

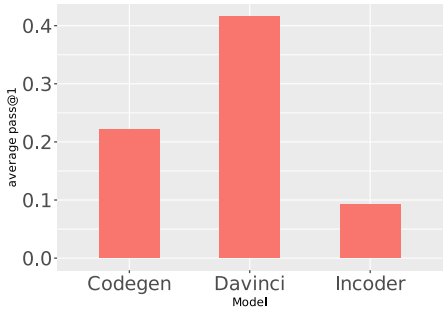

Fig. 10. Average *pass@1*: Replication results confirming results reported by MultiPL-E.

6. Experiment reproduction example

In this section, we demonstrate how LASSO's data structures allow HumanEval-J's experiment to be easily *reproduced* to address the same research question (research question 1, but using a different design to that of the original experiment in Cassano et al. (2023b).

6.1. Experimental design

The modification we apply to the design is to use an automated unit test generation tool to extend the set of tests used to judge functional correctness. In terms of the SRM, this corresponds to adding additional rows to the SRMs while keeping the columns (*i.e.*, components generated by the models) the same. The benefit of doing this is explained by the following statement made in a recent AlphaCode paper (Li et al., 2022) – "the lack of sufficient test cases in existing competitive programming datasets makes the metrics defined on them prone to high false positive rates (with 30% or more programs which pass all tests but are not actually correct), and therefore unreliable for measuring research progress". In the terminology of these benchmarks, we enhanced the design by adding extra, "hidden" tests to make the functional correctness metrics more reliable.

While the AlphaCode authors used fuzzing techniques to mutate input values of the original tests in order to generate additional tests, we used the best-in-class EvoSuite automatic unit test generator (Fraser and Arcuri, 2011; Vogl et al., 2021) to obtain additional tests to improve the

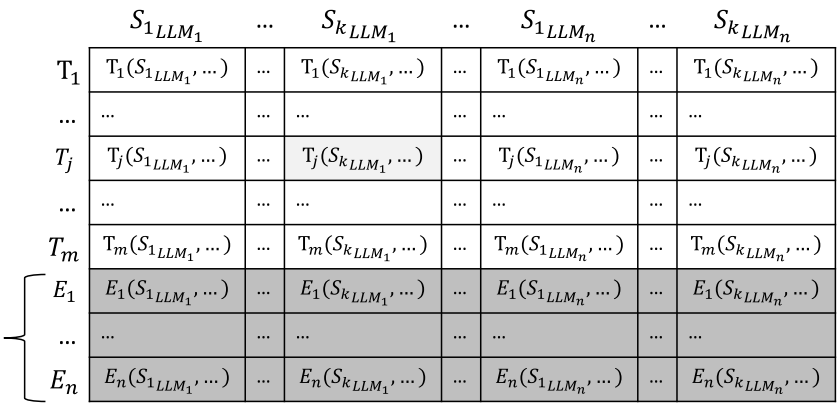

Fig. 11. Adding tests generated by EvoSuite to an SRM.

Table 1
Average *pass@1*: Results before and after adding tests generated by EvoSuite, loss in total and in % per model.

Model	Before	After	Loss	Loss in %
CodeGen	0.22	0.16	0.06	0.27
Davinci	0.42	0.32	0.10	0.23
InCoder	0.09	0.07	0.03	0.29

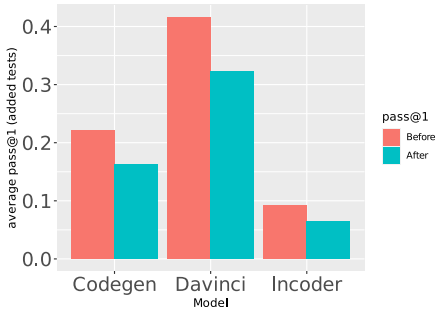

Fig. 12. Average *pass@1*: Results before and after adding tests generated by EvoSuite.

reliability of the functional correctness metric. Since EvoSuite relies on software components as its input, and since tests have to be executed on all the other components implementing a functional abstraction, we ran an additional study pipeline script that is shown in Listing 3 (Appendix A). As illustrated in Fig. 11, the original SRMs from the replication experiment described in the previous section were "grown" by adding new rows in the form of new tests generated by EvoSuite. This ultimately leads to the execution of many more component-test pairs.

The strategy we followed was inspired by the test amplification approach categorized as AMP_{add} in Danglot et al. (2019), Kessel and Atkinson (2019). For each abstraction of the benchmark, we randomly sampled 10 unique functionally correct software components from all the available functionally correct software components (as judged by the original tests from HumanEval-J). We then used EvoSuite to generate additional tests from these samples. Based on the generated tests we then executed those on all the software components in order to obtain additional actuation sheets. To obtain an oracle for the generated tests, we selected the response which the majority of the functionally correct components agreed on (based on behavioral clustering using SRH-based analysis of the data as explained in the next section). Together with the actuations obtained from the original tests, we then judged functional correctness using the original metric (*i.e.*, *pass@k*).

Note that in order to execute this experiment, we leveraged LASSO's ability to reuse past results (*i.e.*, SRMs) created by previous script executions. This capability allows experimenters to simply "resume" at a certain state in a new study pipeline script. Here we resumed the replicated experiment after the arena produced the output SRMs. The new script then analyzed the SRMs to filter the functionally correct components and used them as input for the aforementioned test amplification strategy.

6.2. Results

Overall, EvoSuite managed to generate 26.29 tests on average for each functional abstraction from at most 10 randomly sampled functionally correct components. Table 1 shows the average *pass@1* metrics before and after the new tests were added as well as the resulting changes, while Fig. 12 visualizes these changed metrics as a bar plot.

These results show that the generated tests actually changed the functional correctness judgments from those made using the original set of tests. We can observe a significant drop in the *pass@1* measures for all three models. A direct comparison to the measures obtained in the replicated experiment shows an overall decrease of roughly 25%. This indicates that the original set of tests poorly discriminated the behavior of the components generated by the model. As well as confirming the aforementioned statement made by the authors of AlphaCode (Li et al., 2022), this experiment also demonstrates the benefits of incrementally extending SRMs. Apart from the newly generated tests, no other tests had to be re-executed in order to reason over all tests involved in the analysis of this research question. The SRM data produced by the replication experiment were reused and extended with the additional records obtained by running the newly added tests generated by EvoSuite.

7. Experiment reuse example

In this section, we demonstrate how observational data contained in SRMs can be *repurposed* to help answer new research questions by incrementally "growing" them and the SRH's used to analyze them. This shows how the presented data structures can facilitate the fourth kind of reuse use case discussed in Section 2.3 – the repurposing of experimental results. To this end, we formulate a novel research question and extend the previous replication and reproduction of the HumanEval-J experiments described in Sections 5 and 6 respectively —

RQ 2. *What is the variability in observable functional behavior exhibited by the software components sampled from the code models?*

Table 2
Excerpt of 10 (out of 20) behavioral clusters identified for the GCD abstraction (randomly sampled, unique behavioral clusters, across components of all models).

Cluster ID	29	39	7	54	13	52	38	17	55	25
T_0@1	1	3	−1	7	0	5	3	0	7	1
T_1@1	5	10	−1	15	0	6	10	10	15	5
T_2@1	7	7	−1	14	0	6	49	49	196	EXCEPTION
T_3@1	12	24	−1	60	0	3	144	144	3600	EXCEPTION
#Components	643	15	10	29	41	4	17	1	5	1

Table 3
Descriptive statistics of cluster sizes over all functional abstractions (research question RQ 2).

	min	q25	median	q75	max	mean	sd
Total	1	11.00	23.50	49.75	143	32.71	29.25
Codegen	1	3.00	9.00	21.00	80	14.45	16.04
Davinci	1	2.00	4.00	7.00	55	5.83	7.39
InCoder	1	6.00	12.00	24.00	76	17.61	16.05

7.1. Behavioral cluster analysis

Answering this new research question demonstrates how the fine-grained observational records stored in SRMs (*i.e.*, the actuation sheets) can be leveraged to further analyze the behavior of the components generated by the three models. To the best of our knowledge, no work has yet shed light on the bandwidth and variability of behavior exhibited by the generated components.[11]

To explore this question, we identified unique clusters of equivalent behavior based on the responses of components for each functional abstraction in HumanEval-J. In contrast to the previous two experiments, where the results returned by each component were compared to the oracle values, in this analysis we identified all unique responses (that is technically, all unique columns of values) for each abstraction and grouped them into clusters of identical values.

To illustrate the clustering, Table 2 shows a randomly selected sample of 10 behavioral clusters out of the 20 clusters identified for the GCD functional abstraction across the components generated by the models. It is important to stress that no re-execution of the experiment was necessary to obtain these measurements, since the SRM records created in the replication process (cf. Section 5) already contain all the required data points for the clustering analysis (*i.e.*, actuation records). In other words, this analysis was conducted exclusively in the data analytics tool of choice (*i.e.*, R) using SRH-based navigation over the observational data.

7.1.1. Results

In order to obtain the results, we first identified the average number of behavioral clusters across all functional abstractions and all three code models under investigation. We then identified the behavioral clusters across all abstractions for each model. Table 3 presents the descriptive statistics from our cluster analysis averaged over all abstractions (*i.e.*, coding problems).

Overall, we identified 32.71 (median 23.5) unique behavioral clusters per coding problem (*i.e.*, functional abstraction). When we compared the variability of the exhibited behaviors across the generated components measured by the number of unique clusters, we found that the Davinci (Codex) model has the least variability with 5.83 clusters on average (median 4). The two remaining models, however, have a much larger number of unique clusters with 14.45 clusters for CodeGen (median 9), and 17.61 clusters for InCoder (median 12). Therefore, the Davinci model not only provides the best performance with respect to functional correctness (cf. Section 5), it also provides the least variability in functional behavior across the three models compared.

In terms of the underlying SRMs, this experiment "grew" the original SRMs by storing additional measurements together with their context information — namely, observational records of the data obtained from the clustering analysis. The identification of unique behavioral clusters creates new possibilities to map (or label) the clusters found for different functional abstractions, which is a worthwhile endeavor in its own right.

8. Discussion

In this section we first discuss the threats to validity of the experimental results described in the previous three sections and then discuss the benefits and drawbacks of the presented approach in the context of test-driven software experimentation and open science.

8.1. Threats to validity

In Section 5, we demonstrated the utility of SRMs and sequence sheets by successfully replicating the HumanEval-J benchmark applied to three code models for the Java programming language. Since we used the same experimental design, the replication suffers from the same threats to validity that the authors of MultiPL-E mention in their work (Cassano et al., 2023b).

8.1.1. Inherited threats

MultiPL-E uses a suite of stripped-down compilers to translate code generation benchmarks from Python into new languages in order to enable the evaluation of models for these languages. Since the original HumanEval benchmark was designed for Python coding problems, the Java benchmark we used was automatically generated. One threat to validity, therefore, is that the benchmark itself is not optimized for Java and the coding problems might not accurately reflect Java's real-world performance. The authors mitigated this threat by validating the generated prompts (including tests) in various ways including manual inspection and automatic verification through test suites. Related to this, another threat is the representation and construction of code prompts as part of the coding problems that were used to generate software components. The models under assessment are known to be sensitive to the structure of code prompts, so the threat arises from the fact that the prompts were not optimized for Java code completions (*e.g.*, Python doctest[12] is not supported in Java). To mitigate the threat, the authors kept information about all the original Python prompts for other languages.

Finally, to mitigate the threat of unsound statistical results (*i.e.*, the lack of generalizable results) due to the inherent non-determinism of probabilistic models, the authors of MultiPL-E used an acknowledged sampling technique to sample software components generated from the models in order to obtain measurements of their functional correctness. To further control non-determinism, the authors chose the commonly used temperature hyperparameter value of 0.2.

We "reused" the sampled software components selected in the original experiment, given the significant effort and time involved in obtaining them (cf. cost discussion in Cassano et al., 2023b). Since the authors used a well-acknowledged sampling technique, as explained above, repeating the sampling process (under the same controlled conditions) would almost certainly have delivered the same overall results.

[11] The bandwidth and range of behavior are shaped by the inherent structural design of the produced code. Given the probabilistic nature of the code models, the components generated for a specific coding problem are expected to exhibit variations.

[12] https://docs.python.org/3/library/doctest.html

8.1.2. Serializing objects

In Section 7, we demonstrated how the SRMs of the replicated benchmark can be reused and extended to address additional research questions. In our approach, we focused on the functional correctness, equivalence and/or similarity of software components by performing SRH-based analyses of the observation data stored in the actuation sheets. In contrast to classic unit testing, which makes these measurements at run-time, we performed the analysis "offline" on the presented data structures utilizing the data-driven analysis capabilities offered by the LASSO platform. However, a major challenge in this approach is effectively serializing the observed responses from the software components at run-time (*i.e.*, serializing Java objects), since information loss may occur as part of the process. Consequently, the comparison of serialized values in the SRH-based analysis (*i.e.*, the tabular data frame representation) may not deliver the same results as the equivalence checks performed by the assertions at execution time. To mitigate this threat, we employed a carefully designed serialization strategy that serializes the state of Java objects into a JSON representation. We manually inspected all the possible object types used in the coding problems (*i.e.*, the input parameter and output parameter types of the methods under test) and verified that these could be successfully serialized.

8.1.3. Automated test generation

Finally, a threat to the validity of the reproduction demonstrated in Section 6, which leveraged discrepancy-based testing to identify disagreements between the exhibited behavior of components, is that tests were automatically generated independently of the "actual" functional abstractions realized by the sampled software components. Since EvoSuite generates tests solely by analyzing the code provided to it, the test generator is blind to their functionality (*i.e.*, semantics). To mitigate this threat, we sampled (up to) 10 unique software components that were judged to be functionally correct by the original tests of the benchmark and checked that this was in fact the case. Finally, since EvoSuite often generates tests at the extremes or borders of the input space (*e.g.*, null values, negative values etc.), there is a chance that some of the recorded component responses are syntactically different, but semantically equivalent. For example, one method may return a "null" value to signify invalid inputs, whereas another method may return an empty string, although both indicate the same error (*i.e.*, Java exception). In our case, we take a strict view of what "same behavior" means and assume that there is only one correct response to any stimulus.

8.2. Impact on test-driven software experimentation and open science practices

As well as the advantages of the tabular representations, in this subsection we discuss the potential impact of the presented technology on test-driven software experimentation in general, and on support for open science practices in particular. In Section 5 we demonstrated how SRMs are a useful tool to facilitate the operation and analysis phases of the experimental process, from the execution of the experiment to the analysis of the results obtained. The improvements provided by SRMs not only benefit the original team, but also experimenters attempting to repeat, replicate, reproduce or repurpose previous experiments.

The three notions of reproducibility defined by the ACM, as well as our notion of repurposing presented in Section 2.3, are all supported by SRMs. Firstly, the repetition of an experiment can be supported by producing several instances of the very same SRM. Secondly, the replicability of the experiment is enabled by comparing past SRMs with the newly obtained ones. When combined with the other experimentation services offered by LASSO (*i.e.*, the executable corpus and study pipeline language), the replication process is made much simpler and (semi)-automated, since comparisons can be achieved in standardized ways. Reproducibility, when independent teams make changes to the experimental design, is supported by the inherent extensibility of SRMs, and the ability to analyze ensembles of them offline via a multidimensional SRH.

The final reuse use case, repurposing, can easily be achieved with SRMs and SRHs by adding additional rows and/or columns to existing SRMs, extending them with additional measurements stored with the appropriate contextual information, and by adding new SRMs. This additional information can be used to "tag" variations of experimental designs (make contextual information explicit) and facilitate comparisons to other measurements that used a different design (*e.g.*, experiments using different measurement facilities or measurement scopes that are otherwise identical).

SRMs provide a common format for storing vast numbers of software observations. We believe that such a scalable, standard format significantly improves the accessibility of TDSEs. Once third parties have gained a common understanding of SRMs, understanding presented results (technically) become much easier.

The tabular representation of SRMs also lowers the barriers to sharing them. To date, many (free) platforms exist that explicitly encourage the sharing of (very) large, open datasets in tabular representations and even provide some basic viewing and manipulation capabilities (*e.g.*, Hugginface[13]). If bundled with systematic study pipelines like those supported by LASSO, both open data and open material become much easier to achieve.

Despite the reproducibility concerns in TDSEs discussed in Section 7, we also demonstrated how existing experiments can be "repurposed" in a systematic way using SRMs in order to address a new research question. We showed how new measurements can be added to existing SRMs (*i.e.*, using behavioral clustering), and demonstrated that SRMs can be grown to accommodate additional tests (by adding more rows) and more software components (by adding more columns). This opens up the possibility of performing new kinds of comparisons. For example, adding components recommended by traditional search code engines opens up new experimental designs and richer comparisons between program synthesis and reuse. Other benchmarks in the field of program synthesis also offer reference component implementations which can serve as the oracles for the functionality desired from generated components (*i.e.*, often used to compute the *BLEU* metric Papineni et al., 2002). In terms of SRMs, this reference implementation can be added as a new column for comparison with the columns of other components, when judging functional correctness (recall that we used a virtual component that played the role of the oracle). When SRMs are gradually extended with components and tests, the transparency of experimental designs and results increases.

SRMs can be used to partially avoid the costly operation phase of the experimental process, since behavioral comparisons can be offloaded to data-driven analysis. Executing software components is non-trivial and requires reliable, automated execution logistics when performed on a large scale. Without SRMs and their observational records, experimenters would have to re-execute their experiments whenever they make modifications to their experimental design (*e.g.*, by adding software components, fixing tests etc.). SRMs offer an attractive alternative that can save time and resources, since experiments typically have limited time budgets (*e.g.*, access to a computing grid for the mass execution of software is typically restricted for academia). Moreover, oracle values can simply be added to SRMs as "virtual" components without the need to re-execute any component implementations.

Finally, the size of SRMs is practically unlimited. By collecting the SRMs of many experiments over time, an "observation warehouse" (*i.e.*, an instance of a data warehouse) can be assembled that not only enables meta-experimentation, but also offers the opportunity to use machine learning techniques to derive new insights from software observations, and eventually feed machine learning models to create new services (*e.g.*, oracle recommendation Langdon et al., 2017).

[13] https://huggingface.co/

8.3. Limitations

In this work, we demonstrated SRMs and sequence sheets based on Java software components. As already explained in the threats to validity subsection, one aspect of SRMs that needs further investigation, especially with respect to the storage of actuation sheets, is serializing observed object states at run-time and ensuring that this serialization results in the same comparison results as assertion statements at run-time. Apart from that, SRMs and sequence sheets are essentially independent of the programming language and paradigm used to create and analyze them. Although we are confident that the serialization of objects (or program state) is feasible in all relevant programming languages used today, this issue also requires further investigations.

By their very nature, software observations may cover a wide range of measurements, from atomic facts like numerical measures to large execution traces (*i.e.*, dynamic call graphs). Due to their tabular form, SRMs obviously cannot be used to store such large, monolithic records. We addressed this limitation by "linking" cells that represent measurements in an SRM to other records like large execution traces. Third-parties can then resolve those references to the actual records (*e.g.*, raw files). Based on our experience and the nature of experiments, however, raw (unstructured) files are typically processed and reduced[14] to structured data that can be stored directly in tabular representations.

The running example (*i.e.*, experiments) we used to showcase the benefits of SRMs, and the other data structures presented in this paper, were fully automated and executed as study pipeline scripts on the LASSO platform. However, a qualitative assessment is needed to reinforce the validity of the obtained results. Qualitative data, even though collected manually, can indeed be incorporated into SRMs. Although we did not demonstrate this capability, in practice, we expect that general data analytics tools can be used to merge manual data with SRMs in an automated manner (*e.g.*, by adding manual, virtual oracle components).

8.4. Continual observation

As explained in Section 3, by providing systematic ways of recording stimuli and corresponding system behavior, and allowing them to be viewed and analyzed in a multidimensional way, the three data structures presented in this paper provide a solid foundation for supporting the different forms of experiment reuse described above. For example, the basic goal of supporting repetition and replication is enabled by the clear, concise and systematic recording of **all** execution data (*i.e.*, all responses as well as stimuli) in clearly understandable and navigable data structures. This is essential for independent teams to be able to understand how to replicate a TDSE. The statistical power of repetitions and replications can also be easily enhanced by adding more SRMs for more subjects in the "functional abstractions" dimension, or by adding more SRMs in the "repetitions" dimension.

In addition to these options, the reproduction of experimental data can further be achieved by either using different analysis techniques to analyze the existing data, or by extending the existing SRMs with additional tests or additional implementations of functional abstractions. These can provide the information needed to address the existing research questions in a deeper or alternative way. Finally, in addition to all these previous techniques, the repurposing of experimental execution data can be achieved by adding additional dimensions in which independent variables that were previously kept constant are varied in controlled ways.

The options to incrementally extend and re-analyze the observation data contained in SRMs, and incrementally grown SRHs, opens up the possibility of "continual" test-driven software experimentation where the existing base of observation data is systematically and incrementally expanded to make existing studies more statistically powerful and to answer new research questions.

[14] Assuming that these are not needed for replication purposes

9. Conclusion

TDSEs revolves around the execution of software components to investigate their functionality (*i.e.*, semantics) and dynamic non-functional properties (*e.g.*, performance). In this paper, we introduced three new data structures designed to simplify test-driven software experimentation and promote the repetition, replication, reproduction and repurposing of TDSEs using open science principles.

The first of the three data structures presented was the **sequence sheet** approach for representing stimuli and responses (*i.e.*, actuations in terms of inputs and outputs) of software components, the second was the **Stimulus Response Matrix (SRM)** data structure for storing actuations of implementations of a given functional abstraction so that they can be readily compared, and the third was the **Stimulus Response Hypercube (SRH)** data structure, which allows ensembles of SRMs to be analyzed offline using a multidimensional hypercube model similar to that supported by data warehouses.

The paper showed how the LASSO platform, which supports these data structures, can be used to reproduce and repurpose a recent benchmark (*i.e.*, the HumanEval benchmark for Java Chen et al., 2021; Cassano et al., 2023b) for assessing the performance of large language models for code generation (*i.e.*, generative AI-based program synthesis). This demonstrated the utility of SRMs as a common representation for storing observational records, and their ability to foster the reproducibility of TDSEs. Further, based on the idea of hypercubes in data warehousing applications, we also demonstrated how SRHs can be used to effectively analyze the recorded observational data using today's widespread data analytics tools. The material and data for these experiments is publicly available (Kessel and Atkinson, 2023).

Although the existing MultiPL-E benchmark used in our running example was realized in the traditional style by writing large quantities of dedicated study code, we showed how it can be relatively easily translated into, and replicated using, the sequence sheet, SRM and SRH data structures supported by LASSO. Further, we showed that once in this format, the observation data can be readily extended and repurposed to support new experiments with new research questions.

The experiments presented in the running example not only provide empirical evidence of the utility of the proposed approach, but also contribute to the growing body of knowledge about AI-powered code models by providing new insights into the characteristics of the software components they generate. Looking ahead, future work could build on our findings with additional research questions such as exploring the quality of the components and tests as well as the levels of redundancy they contain (*e.g.*, to address the ongoing challenge of "attribution" Chen et al., 2021; Li et al., 2023).

Overall, we hope that third-parties will use the proposed data structures and the LASSO platform in future TDSEs. In the long run, we hope to establish a community of the kind found in the data science and machine learning disciplines, that openly share their SRMs (*e.g.*, as data frames on popular sharing platforms) in order to grow an open observation warehouse (*i.e.*, a large collection of SRMs) that can be mined for additional purposes (*e.g.*, oracle recommendation). In the meantime, recent publications in the field of AI-based program synthesis have proposed many more (improved) benchmarks that we would like to explore using SRMs to include them in open benchmark suites (*e.g.*, EvalPlus in Liu et al., 2023). Using the LASSO platform used in this work, we aim to automate and support as much of the experimental process as possible.

```
dataSource 'multipleBenchmark23'

String benchmarkId = "humaneval-java-reworded"

study(name: 'MultiPLE-HumanEval-Java') {
  def humanEvalBenchmark =
   ↪ loadBenchmark(benchmarkId) // load
   ↪ benchmark

  action(name:'select', type:'Select') {
    execute() {
      // create abstraction containers from the
       ↪ benchmark's problems
      humanEvalBenchmark.abstractions.each {
       ↪ problemId, problem ->
        abstraction(problemId) { // retrieve
         ↪ all code completitions for given
         ↪ problem
          queryForClasses "*:*" // always
           ↪ 'Problem'
          rows = Integer.MAX_VALUE

          filter 'benchmark:"'+ benchmarkId
           ↪ +'"' // specify benchmark
          filter 'problem:"'+ problemId +'"'
           ↪ // specify problem
          //filter 'generator:"'+ generatorId
           ↪ +'"' // specific generators
           ↪ (code models)
        }
      }
    }
  }

  profile('myProfile') {
    scope('class') { // measurement scope
      type = 'class'
      methodBlacklist = ['main', '<init>',
       ↪ '<clinit>'] // ignore given methods
    }
    environment('java11') { // execution
     ↪ environment
      image = 'maven:3.6.3-openjdk-11' //
       ↪ (docker) image template
    }
  }

  action(name:'execute',type:'ArenaExecute')
   ↪ {
    // use tests and interface provided by the
     ↪ benchmark
    benchmark = humanEvalBenchmark.name

    features = ['cc'] // measure code coverage
     ↪ (here JaCoCo)

    dependsOn 'select'
    includeAbstractions '*'
    profile('myProfile')
  }
}
```

Listing 2: LASSO Study pipeline script written in LSL (LASSO Scripting Language) for research RQ 1 and research RQ 2 in Section 5 to 7.

```
dataSource 'multipleBenchmark23'

def benchmarkId = "humaneval-java-reworded"
def studyPath =
 ↪ '90196763-4e3c-419e-9ff2-5e754bee3533:execute' //
 ↪ fetch results from past action
def sampleSize = 10 // sample 10 code completions subject
 ↪ for test generation
def evoSuiteSearchBudget = 30 // EvoSuite's time budget

study(name: 'MultiPLE-HumanEval-Java-TestGeneration') {
  action(name:'equivalent') {
    dependsOn studyPath // depends on other study
     ↪ execution
    includeAbstractions '*'
    execute {
      abstractions.each {abName, ab -> // filter by
       ↪ oracle
        def mySrm = srm(abstraction: ab, path:
         ↪ studyPath)
        def expectedBehaviour =
         ↪ toOracle(mySrm.sequences)
        def matchesSrm = mySrm
            .systems // select all systems
            .equalTo(expectedBehaviour) //
             ↪ functionally equivalent

        ab.systems = matchesSrm.systems
      }
    }
  }
  profile('myProfile') {
    scope('class') { // measurement scope
      type = 'class'
      methodBlacklist = ['main', '<init>',
       ↪ '<clinit>'] // ignore given methods
    }
    environment('java11') { // execution environment
      image = 'maven:3.6.3-openjdk-11' // (docker)
       ↪ image template
    }
  }
  action(name:'generate',type:'EvoSuite') {
    searchBudget = evoSuiteSearchBudget

    dependsOn 'equivalent'
    includeAbstractions '*'
    includeSystems {abName -> // take N unique(!)
     ↪ samples
      List systems = abstractions[abName].systems
      if(!systems  systems.size() == 0) {
        return
      }

      def random = new Random()
      Set indices = new HashSet()
      while(indices.size() < sampleSize) {
        indices.add(random.nextInt(systems.size()))
      }

      List samples = []
      indices.each { i ->
        samples << systems[i]
      }

      abstractions[abName].systems = samples
    }
    profile('myProfile')
  }
  action(name:'execute',type:'ArenaExecute') {
    configure {
      benchmark = benchmarkId
      noTestsFromBenchmark = true // only execute
       ↪ newly generated tests
      features = ['cc'] // enable code coverage

      populateTestsFromAction = 'generate'
    }

    dependsOn 'equivalent'
    includeAbstractions '*'
    profile('myProfile')
  }
}
```

Listing 3: LASSO Study pipeline script written in LSL (LASSO Scripting Language) for reproducing research RQ 1 in Section 6.

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

The data set is linked in the article.

Acknowledgments

We are grateful for the valuable suggestions and feedback from the anonymous reviewers. We also thank the authors of MultiPL-E (Cassano et al., 2023b) and StarCoder (Li et al., 2023) for making their benchmarking data and material publicly available.

Appendix A

For our experiments in Section 5 to 7, we developed two LSL pipeline scripts (see Listing 2 and 3) that were executed on the LASSO platform to address our research questions. Due to space constraints, however, we are unable to provide an in-depth explanation of the scripting language and its application within these pipelines. Interested readers should refer to Kessel (2023) for further information on this topic.

References

ACM, 2023. Artifact review and badging – version 1.0. URL: https://www.acm.org/publications/policies/artifact-review-badging.

Aghajanyan, A., Huang, B., Ross, C., Karpukhin, V., Xu, H., Goyal, N., Okhonko, D., Joshi, M., Ghosh, G., Lewis, M., Zettlemoyer, L., 2022. CM3: A causal masked multimodal model of the internet. arXiv:2201.07520.

Allamanis, M., 2019. The adverse effects of code duplication in machine learning models of code. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. In: Onward! 2019, Association for Computing Machinery, New York, NY, USA, pp. 143–153. http://dx.doi.org/10.1145/3359591.3359735.

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.

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

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering. ICSE '05, Association for Computing Machinery, New York, NY, USA, pp. 402–411. http://dx.doi.org/10.1145/1062455.1062530.

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250. http://dx.doi.org/10.1002/stvr.1486, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C., 2021. Program synthesis with large language models. arXiv:2108.07732.

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).

Basili, V.R., Selby, R.W., Hutchens, D.H., 1986. Experimentation in software engineering. IEEE Trans. Softw. Eng. SE-12 (7), 733–743. http://dx.doi.org/10.1109/TSE.1986.6312975.

Ben Allal, L., Muennighoff, N., Kumar Umapathi, L., Lipkin, B., von Werra, L., 2022. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness.

Caserta, J., Kimball, R., 2013. The Data Warehouseetl Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data. Wiley.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.H., Zi, Y., Anderson, C.J., Feldman, M.Q., Guha, A., Greenberg, M., Jangda, A., 2023a. Completions Data Set: Multi-Programming Language Evaluation of Large Language Models of Code (MultiPL-E). URL: https://huggingface.co/datasets/bigcode/MultiPL-E-completions.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.H., Zi, Y., Anderson, C.J., Feldman, M.Q., Guha, A., Greenberg, M., Jangda, A., 2023b. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. IEEE Trans. Softw. Eng. 1–17. http://dx.doi.org/10.1109/TSE.2023.3267446.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.H., Zi, Y., Anderson, C.J., Feldman, M.Q., Guha, A., Greenberg, M., Jangda, A., 2023c. Problem data set: Multi-programming language evaluation of large language models of code (MultiPL-E). URL: https://github.com/nuprl/MultiPL-E.

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

Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., Baudry, B., 2019. A snowballing literature study on test amplification. J. Syst. Softw. 157, 110398. http://dx.doi.org/10.1016/j.jss.2019.110398, URL: https://www.sciencedirect.com/science/article/pii/S0164121219301736.

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.

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.

Ernst, M.D., 2003. Static and dynamic analysis: Synergy and duality. In: WODA 2003: ICSE Workshop on Dynamic Analysis. New Mexico State University Portland, OR, pp. 24–27.

ESE, 2023. Empirical software engineering - An international journal. URL: https://www.springer.com/journal/10664.

ESEM, 2023. Empirical software engineering and measurement. URL: https://www.esem-conferences.org/.

Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J., 2022. The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In: Proceedings of the 24th Australasian Computing Education Conference. ACE '22, Association for Computing Machinery, New York, NY, USA, pp. 10–19. http://dx.doi.org/10.1145/3511861.3511863.

Fraser, G., Arcuri, A., 2011. EvoSuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. In: ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA, pp. 416–419. http://dx.doi.org/10.1145/2025113.2025179.

Fraser, G., Arcuri, A., 2012. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.

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.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., tau Yih, W., Zettlemoyer, L., Lewis, M., 2023. InCoder: A generative model for code infilling and synthesis. arXiv:2204.05999.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., Leahy, C., 2020. The pile: An 800gb dataset of diverse text for language modeling. arXiv:2101.00027.

Gousios, G., 2013. The GHTorent dataset and tool suite. In: 2013 10th Working Conference on Mining Software Repositories. MSR, pp. 233–236. http://dx.doi.org/10.1109/MSR.2013.6624034.

Gulwani, S., Polozov, O., Singh, R., 2017. Program synthesis. Found. Trends Program. Lang. 4 (1–2), 1–119. http://dx.doi.org/10.1561/2500000010.

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

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. In: ISSTA 2014, Association for Computing Machinery, New York, NY, USA, pp. 437–440. http://dx.doi.org/10.1145/2610384.2628055.

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., 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: Promoting Open Science in Test-Driven Software Experiments. http://dx.doi.org/10.5281/zenodo.8208246.

Kocetkov, D., Li, R., Allal, L.B., Li, J., Mou, C., Ferrandis, C.M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., de Vries, H., 2022. The stack: 3 TB of permissively licensed source code. arXiv:2211.15533.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., Liang, P.S., 2019. SPoC: Search-based pseudocode to code. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché Buc, F., Fox, E., Garnett, R. (Eds.), In: Advances in Neural Information Processing Systems, vol. 32, Curran Associates, Inc., URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf.

Langdon, W.B., Yoo, S., Harman, M., 2017. Inferring automatic test oracles. In: 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing. SBST, pp. 5–6. http://dx.doi.org/10.1109/SBST.2017.1.

Li, R., Allal, L.B., Zi, Y., et al., 2023. StarCoder: may the source be with you!. arXiv:2305.06161.

Li, Y., Choi, D., Chung, J., et al., 2022. Competition-level code generation with AlphaCode. Science 378 (6624), 1092–1097. http://dx.doi.org/10.1126/science.abq1158, URL: https://www.science.org/doi/abs/10.1126/science.abq1158.

Liu, J., Xia, C.S., Wang, Y., Zhang, L., 2023. Is your code generated by chatGPT really correct? Rigorous evaluation of large language models for code generation. In: Thirty-Seventh Conference on Neural Information Processing Systems. URL: https://openreview.net/forum?id=1qvx610Cu7.

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.

Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R., Mockus, A., 2021. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. Empir. Softw. Eng. 26 (2), 22. http://dx.doi.org/10.1007/s10664-020-09905-9.

Maj, P., Siek, K., Kovalenko, A., Vitek, J., 2021. CodeDJ: Reproducible queries over large-scale software repositories. In: Møller, A., Sridharan, M. (Eds.), 35th European Conference on Object-Oriented Programming (ECOOP 2021). In: Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 6:1–6:24. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2021.6, URL: https://drops.dagstuhl.de/opus/volltexte/2021/14049.

Markovtsev, V., Long, W., 2018. Public git archive: A big code dataset for all. In: Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18, Association for Computing Machinery, New York, NY, USA, pp. 34–37. http://dx.doi.org/10.1145/3196398.3196464.

Mendez, D., Graziotin, D., Wagner, S., Seibold, H., 2020. Open science in software engineering. In: Felderer, M., Travassos, G.H. (Eds.), Contemporary Empirical Methods in Software Engineering. Springer International Publishing, Cham, pp. 477–501. http://dx.doi.org/10.1007/978-3-030-32489-6{_}17.

Méndez Fernández, D., Monperrus, M., Feldt, R., Zimmermann, T., 2019. The open science initiative of the empirical software engineering journal. Empir. Softw. Eng. 24 (3), 1057–1060. http://dx.doi.org/10.1007/s10664-019-09712-x.

Minocher, R., Atmaca, S., Bavero, C., McElreath, R., Beheim, B., 2020. Reproducibility improves exponentially over 63 years of social learning research. http://dx.doi.org/10.31234/osf.io/4nzc7, URL: https://osf.io/preprints/psyarxiv/4nzc7.

Monperrus, M., 2018. Automatic software repair: A bibliography. ACM Comput. Surv. 51 (1), http://dx.doi.org/10.1145/3105906.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C., 2023. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474.

Nong, Y., Sharma, R., Hamou-Lhadj, A., Luo, X., Cai, H., 2023. Open science in software engineering: A study on deep learning-based vulnerability detection. IEEE Trans. Softw. Eng. 49 (4), 1983–2005. http://dx.doi.org/10.1109/TSE.2022.3207149.

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.

Papineni, K., Roukos, S., Ward, T., Zhu, W.J., 2002. BLEU: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. ACL '02, Association for Computational Linguistics, USA, pp. 311–318. http://dx.doi.org/10.3115/1073083.1073135.

Rice, H., 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.

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.

Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A., 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 201–211. http://dx.doi.org/10.1109/ASE.2015.86.

Siegmund, J., Siegmund, N., Apel, S., 2015. Views on internal and external validity in empirical software engineering. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1. pp. 9–19. http://dx.doi.org/10.1109/ICSE.2015.24.

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

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

The R. Foundation, 2022. The R Project for Statistical Computing. URL: https://www.r-project.org/.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (Eds.), In: Advances in Neural Information Processing Systems, vol. 30, Curran Associates, Inc., URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Vogl, S., Schweikl, S., Fraser, G., Arcuri, A., Campos, J., Panichella, A., 2021. EVOSUITE at the SBST 2021 tool competition. In: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing. SBST, IEEE, pp. 28–29.

Wohlin, C., 2021. Case study research in software engineering—It is a case, and it is a study, but is it a case study? Inf. Softw. Technol. 133, 106514.

Wohlin, C., Höst, M., Henningsson, K., 2003. Empirical research methods in software engineering. In: Conradi, R., Wang, A.I. (Eds.), Empirical Methods and Studies in Software Engineering: Experiences from ESERNET. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 7–23. http://dx.doi.org/10.1007/978-3-540-45143-3_2.

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

Zhang, L., Tian, J.H., Jiang, J., Liu, Y.J., Pu, M.Y., Yue, T., 2018. Empirical research in software engineering — A literature survey. J. Comput. Sci. Tech. 33 (5), 876–899. http://dx.doi.org/10.1007/s11390-018-1864-x.