Bauhaus-Universität Weimar
Faculty of Media
Degree Programme Medieninformatik

Systematic analysis of testing-related publications concerning comparability and reproducibility

Bachelor's Thesis

Artur Solomonik

1. Referee: Prof. Dr. Norbert Siegmund
2. Referee: Prof. Dr. Martin Potthast

Submission date: May 17, 2019

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Weimar, May 17, 2019

. . . . . .
Artur Solomonik

Abstract

Software Testing has always been a crucial part of the software development life cycle. Choosing a testing system that will most-likely identify faults in implementation might heavily influence either the cost and outcome of a project. As a matter of fact, authors provide empirical evaluations to showcase the major fortes of their frameworks. By classifying such publications and revealing strategies based on not only intrinsic attributes of a paper but also their role in the bibliographic network, it is possible to trace shared tasks and priorities of certain software testing research areas. Understanding the major qualities and flaws of the current software testing research community may lead to a decisive improvement of the software development process as a whole. In this thesis, common strategies behind assessing testing systems are analyzed. The main goal of the research is to find a common ground for software testing contributions in order to understand the latest situation of state-of-the-art software testing tool evaluations. By reflecting on properties that influence the reproducibility of an evaluation, publication data is created that can be observed in detail in a node-link-directed visualization. The collected data and insight is used to determine whether referencing patterns in certain research areas occur and can be used to imply properties of a publication concerning their reproducibility and overall quality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the ongoing technological evolution in delivering software that suffices the needs of a customer, software testing systems have majorly grown in popularity. Presenting numerous coverage metrics to clients unversed in software development has turned into a major trend, whereas empirical evaluations of testing tools greatly vary in their core strategy. Even though software engineering as a research field is highly connected to the current industry, the quality of an evaluation is a major indicator for choosing a suitable tool for development. Not only is the term reproducibility a highly discussed one when it comes to assessing empirical result data, but identifying the author's and developer's evaluation strategy should be paid attention to during any literature review.
With a rising amount of papers with every venue, understanding the state of research in a particular time period is essential for conducting a literature overview. Therefore, bibliographic networks have gained in popularity because of the extensive insight on references between publications.
In this bachelor's thesis common evaluation strategies in software testing papers are analyzed based on a number of attributes concerning not only the assessment part but also many different aspects of the publication itself. By taking resource availability, data set state and, most importantly, bibliographic referencing patterns and comparability into account, an ambivalent view on each paper can be established. Moreover, more complex data sets than simple bibliographic metadata may lead to a better understanding of testing tool evaluations while also enabling the improvement of current publication networks.
This research is conducted with the following questions in mind. What strategies do authors use when evaluating their testing tool? A very common approach of evaluations in scientific papers is to apply the implemented algorithm or system on a chosen data set while presenting the gathered data and the resulting metrics. That leads to the question of which data set to choose, why

to use it and how to make it available for the reader. Moreover, with common metrics like precision and recall, how are they reflected in software testing evaluations? As testing a testing tool is a more complex undertaking, measuring the amount of errors might not suffice which gives place to other metrics.
By the fact that evaluations heavily depend on the results of their related work, it is important to find out whether that is apparent in evaluations of software testing papers. As a paper might mention another publication of the same contribution in their related work, thus, comparing oneself to it, it seems logical to reuse the evaluation data and metrics in one's own evaluation. More specifically, when comparing oneself to the work of others with improvement in mind, comparing oneself to the reference's data should be a common approach. With this hypothesis in mind, how relevant is related work in software testing research and what role does comparison play when evaluating a testing system?
Another vital part of comprehending evaluations is the factor of reproducibility. As in most parts of scientific research, making the gathered data reproducible is a key effort for a valid research. That factor highly depends on a great number of aspects of the research including the exact methodology, the software, the time of conductance, the metrics used as well as proper documentation of every step. This research is interested in finding properties of papers that influence their reproducibility considerably. How much of importance is reproducibility in the research domain of software testing and how can it possibly be improved on?
This thesis explains the approach of working with bibliographic data in different ways to understand the nature of software testing tool evaluation. After a thorough literature review and development of a data set including numerous properties concerning the state of the evaluation and the paper itself, the first insight on trends and contributions of the software testing research community could be established. By making use of a graph database and multiple visualization techniques, already existing takes on publication networks are improved on in terms of the new properties defined during the literature review. By refactoring the data set using the publication graph and inspecting the strategies of authors of the community, patterns regarding citations, benchmarks and the overall state of the papers are established.
The results of the literature review are available as a Google spreadsheet online. [1]. The publication network visualization *TeLO-S* can be viewed from the internal network of Bauhaus University [2] or can be forked from GitHub [3].

[1] https://docs.google.com/spreadsheets/d/1CI2MTmAbCTllJPBuCV4Rfk4Kvn2dEaR-JeehCLbteh0/edit?usp=sharing

[2] http://webislab10.medien.uni-weimar.de/

[3] https://github.com/Arduqq/testing-paper-visualization

Chapter 2

Background and Related Work

2.1 Software Testing Paradigms

Software testing systems have evolved over the past decades when it comes to their fundamental approaches. In a perfect development process, most errors are eliminated early on to avoid highly expensive end-to-end and acceptance tests. For the main strategy lies in inspecting the smallest unit component up to the whole system, each step has to be implemented individually with the most fitting testing approach in mind. When discussing a software testing tool, one can mostly classify its model as either dynamic, static, symbolic and concolic. Each model has its benefits in different areas of the software testing process which leaves the developers many opportunities of combining them. By understanding the basic concept of a software testing tool, making assumptions on their evaluation is a lot more feasible.

2.1.1 Dynamic Execution

The basic concept behind testing lies in exploring every way the execution might turn out. As a matter of fact, taking the dynamic approach of exploring each execution path is based on the actual execution of a program under test, dynamic data flow analysis, and function minimization methods [Korel, 1990]. The test data is created by providing the execution with actual values and its data flow is monitored constantly. As soon as there is any termination or undesired flow, a path is annotated as such and will be reported as flawed. A consequent data flow analysis determines the input values that led to the undesired behaviour. That technique is due to the intuitiveness and performance predominantly used in many modern software testing frameworks.

2.1.2 Static Analysis

A contrary approach to directly executing software to detect possible faults is the use of static analysis techniques. When working with static analysis, one has to inspect a system on its structure, form, context or documentation to identify possible errors. [Electrical and Engineers, 1990] Originally, such tasks would be performed manually in dedicated inspections on possible uncaught exceptions, memory leaks, redundancies and other possible causes of future problems. Even though such techniques exist, they were strikingly often ignored by developers leading to errors in production that could have been circumvented by simply inspecting the code like in the Apple security vulnerability called the *goto fail* [Synopsys, 2014].
In their article, Sadowski et al. explain that developers tend to ignore static analysis techniques because of the high cost of removing found errors or their inability to comprehend possible warnings. Therefore, companies like Google take simple static analysis tools into consideration compared to more sophisticated ones while presenting their own way of working with it. Because of the on-going trend of automating every software development process, Zheng et al. evaluate the economic importance of including automated static analysis in the software development process showing that, for the time being, ASA was not efficient due to the high amount of maintenance to separate true errors from false positives. Therefore, over the past few years, researches take an effort in implementing the process into many different areas of software testing like GUI testing [Arlt et al., 2012] or web security testing [Medeiros et al., 2016] effectively.

2.1.3 Symbolic Execution

Compared to conventional methods of testing software, symbolic execution makes use of a different concept. In their paper, Cadar and Sen explains that the main goal is to explore as many testing paths as possible in a considerable amount of time while generating the concrete input tuples that were used and the corresponding errors that might have happened. Symbolic execution does not rely on directly given values as their input; more specifically, the program deals with symbolic values instead of concrete ones and represents each variable as a so-called symbolic expression. Applying that strategy to a common software test means creating test inputs for every possible execution path there is; as a result, all possible inputs making an assert fail will be identified.
[Baldoni et al., 2018] A path is defined by a binary sequence denoting at which point the paths branch out. The sum of execution paths represents an execution tree. Consequently, fully traversing the tree and taking each path at least

once is the main goal of that approach. In order to adjust and control the execution, a testing process makes use of a symbolic state and a quantifier-free first-order formula called the symbolic path constraint. After each execution of a symbolic execution path, a constraint solver generates an explicit input for the symbolic execution. As soon as the execution returns an exit statement or an error even, the program terminates and the constraint solver clarifies which inputs will result in either a flawless run or an error. As symbolic execution results in an infinite loop when exploring for-loops, one has to define certain control variables such as the number of loop iterations and paths to explore or the exploration depth.
With the idea of using symbols and generating inputs after the execution, symbolic execution has an advantage in finding errors compared to approaches relying on concrete input values at first hand. Up until today, symbolic execution might be a popular choice when generating test cases and sequences. The main problem, on the other hand, lies in the scalability of the technique. Consequently according to Baldoni et al., the focus of the research lies in parallelizing symbolic execution tasks.

2.1.4 Concolic Execution

As a consequence of test generation techniques being either concrete or symbolic, concolic execution was established as a combination of the two. Executing the program with both symbolic and explicit values results in a sequence of path constraints as explained above. The concrete execution serves a supporting role at the point where the symbolic execution reaches its limits. As soon as the constraint solver is unable to find fitting concrete values satisfying the constraint, symbolic constraints are simplified by replacing some of the symbolic values with concrete values. [Sen, 2007] Such unsolvable values mostly come from external code not traceable by the executor, as well as from complex constraints involving non-linear arithmetic or transcendental functions. [Baldoni et al., 2018]

2.2 Reproducibility

The main concern of designing experiments in empirical studies is the retention of reproducibility. The idea is that by reading any study there is, the reader is able to completely recreate the results. In most cases, the Methodology section of a paper is perceived as the best-suited place to explain how to end up with the presented data. Unfortunately, documenting the process of conducting the experiment is certainly not enough when it comes to more

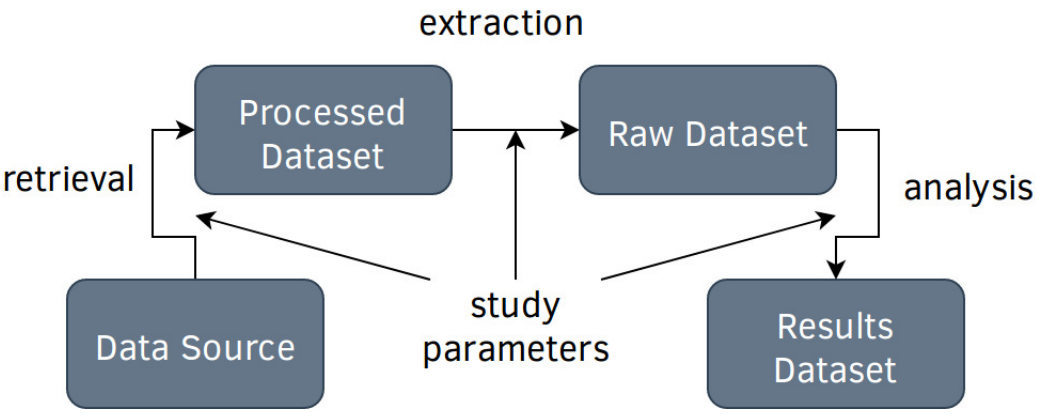

Figure 2.1: Elements of a research process influencing the reproducibility metric

complex data and different software systems [González-Barahona and Robles, 2012]. During the last years, the topic of reproducibility has gained a great interest in the empirical software engineering community. As a result, repositories like Notre Dame SourceForge Research Repository [1] and the Helix software evolution data set [2] have been established to facilitate the reproduction of studies. González-Barahona and Robles propose multiple elements of scientific research influencing reproducibility. By assessing attributes of the different stages a data set goes through and the corresponding modification techniques, a simple grading metric of reproducibility is achieved. In specific, 2.1 showcases the important parts of the research process. Each data set is assessed regarding certain attributes. Identification explains where the data can be obtained. Description shows with what level of detail the element is explained to the reader. Availability is the ease of accessing or obtaining the element. Persistence states whether the element will be available in the future and Flexibility denotes how adaptable the data is in new environments. Regarding different kinds of data, these attributes are not universal. A data source clearly can be assessed by the reader, whereas parameters are limited to their Identification and Description. By applying that technique to scientific research, the ease of reproducing the data can be estimated.

[1] `http://srda.cse.nd.edu/mediawiki/index.php/Main_Page`
[2] `http://www.ict.swin.edu.au/research/projects/helix/`

2.3 Graph Database Models

Graph database models are popular approaches for organizing data structures to manage information for certain use cases. According to the survey of Angles and Gutierrez, such a model relies on graphs and similar abstractions for their schema and instances with their respective operations. In most papers, a graph database is defined as a whole underlying graph in which nodes, links, labels and directions represent the abstracted data [Graves et al., 1995]. Moreover, Angles and Gutierrez explain that a graph database model separates the description of the schema from the respective instances in most definitions.
Manipulating and querying data is expressed by transforming the graph with operations on graph primitives like paths, neighbourhoods, subgraphs, patterns, and statistics like diameter, depth and centrality. Most importantly, integrity constraints enforcing the consistency of the data can be defined on top of the data structure.
In this case, the database is defined as a graph whose nodes represent instances of the schema and are labelled with their attributes and unique names [Graves et al., 1995]. Each node is connected to other nodes with directed edges that are, as well, labelled by the respective attributes.
The use and advantage of graph databases lie in their representation of relationships making the description of dependencies very clear. Information is condensed into a single node that is connected to others using arcs; consequently, more natural modelling of the data is achieved. Moreover, querying data does not require much knowledge of the schema as the graph structure seems rather intuitive when working with more complex rules for the selection.
Graph databases are used in many state-of-the-art infrastructures. Many database models have difficulties with the way they visualize connectivity and complex objects. Therefore, systems with their work based on networks tend to choose graphs as their underlying structure for their data. For instance, social networks would represent users as their nodes with all kinds of relationships between each other. Information networks that depend on information flow like cited papers would be obvious users of graph databases because of the similar structure of the data.

2.4 Related Work

The research is motivated by the rising importance of software testing and the existence of many different branches of the research field itself. In their case study, Kitchenham et al. explain the fundamental steps of a case study regarding the evaluation of presented tools regardless of the functionality and how important it is for a lasting improvement of the work. Poston and Sexton

emphasize the need for wholesome evaluations and information on the software so that companies actually consider to buy them. They also include the fact that the problem may come from a severe lack of communication while presenting possible strategies behind avoiding badly documented software and how to determine the most suitable testing tools.
Visualizing paper citations is a common approach when it comes to analyzing data. de Solla Price presented such networks as a way to get insight into certain research fields as the incidence of references and actual citations are rarely congruous. Moreover, heavily cited papers that appear as "classic" tend to be referenced even more in the future whereas others are never referenced again. He claims that every scientific paper ever published is cited about once a year which indicates a long-lasting growth of such a network.
That approach was adapted to many popular citation network visualizations like VOSViewer [3], Semantic Scholar [4] and Paper Embedding Visualization.
VOSViewer is a very popular tool for visualizing the publication landscape. The user can query the network by specific keywords which generate filtered, colour- and size-coded nodes that are connected by their citation, bibliographic coupling, co-citation, or co-authorship relations.
Such network visualizations are useful for analyzing large amounts of data as mainly core aspects of the many papers are visualized. That simplification tends to be a double-edged sword as a loss of information is inevitable. For instance, when a citation network is constructed, it is possible to see who is citing whom while the cause of the citation of another paper may be ignored. Therefore, citation networks should be used as a complement to expert judgment and only in times where a simple look at the raw data is overwhelming. Moreover, the research on publication networks branches into three main issues, that being the effort of visualizing great amounts of bibliographic data, the increase of interactivity for the sake of exploration and the dynamic visualization of the network evolution over time van Eck and Waltman [2014].
Semantic Scholar released a paper [Ammar et al., 2018] on their strategy on constructing a literature graph for proper algorithmic manipulation and discovery. They especially focus on the extraction of important structural, semantic data of papers using different node and link types. Not only do they take papers and their authors into account, but also entities for the different concepts presented in the paper and their respective mentions. Links are classified by the node types that are linked, e.g. citations, authorship, entity-entity relationships. In their paper, Waltman et al. describe their strategy on mapping and clustering the nodes in a network to enable proper insight on the relation of the research field and their development over time. As mapping is

[3] http://www.vosviewer.com/
[4] https://www.semanticscholar.org/

well-suited for obtaining the structure of the network, relations are displayed poorly. Clustering, on the other hand, does not suffer from dimensional restrictions, yet continuous dimensions cannot be visualized. Consequently, a unified approach is presented.
Paper Embedding Visualization is a cluster overview of a variety of papers and their affiliation with each other. Nodes are mapped by the distance to each other implying similar embeds. By colour-coding papers, one can easily make out papers of similar contributions, yet it is not explicitly annotated. Using Doc2vec [Shperber, 2017] and Latent Semantic Analysis [Landauer et al., 1998], the documents are represented numerically in a vector and compared to each other on how close they lie semantically to each other. The visualization represents a machine learning cluster approach on paper networks and the proximity of documents independent of only certain regions of the document like the evaluation specifically.

Chapter 3

Publication Data Network

In order to retrieve a network of publications, its references and the corresponding benchmarks that were evaluated on, reading and classifying a great amount of testing-related papers was a necessity. The process was split into different phases in which relevant papers were acquired, classified in terms of our research questions, modified for our graph database and visualized accordingly.
Based on this, our network could be developed using a graph database in which all of the references and properties of each paper could be visualized. By querying the database to content, the data on publications and benchmarks can be improved and used to determine common referencing and evaluation patterns.

3.1 Publication Data Acquisition

The primary goal was to create a thorough base for the research; moreover, collecting an exhaustive amount of testing-related publications had a high priority throughout the work. Different methods were used to create valid references between the different entities to enable carefree maintenance and improvement of the data. In the following, the strategy behind finding, filtering, organizing and classifying the publication data is presented.

3.1.1 Paper Collection

The research was conducted systematically in which a variety of publications of well-known software engineering conferences of the past seven years was collected and organized in a repository depending on their venue. Because of their connection to the research field of software testing and software engineering in general, popular conferences and journals were selected. In specific, papers of

the Association for Science Education (ASE), European Conference on Object-Oriented Programming (ECOOP), European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Fundamental Approaches to Software Engineering (FASE), International Conference on Software Engineering (ICSE), International Symposium on Software Testing and Analysis (ISSTA) and conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) were taken into consideration within the time period from 2013 up until 2018. In order to retrieve the papers that were relevant for the task, a Python script that would parse each PDF-file was written and publications containing certain keywords would be filtered. As described in chapter 2.1, papers containing approaches on dynamic, symbolic and concolic execution as well as static analysis seemed desirable for our prior activities. On top of trivial keywords involving software testing, we also filtered publications concerning popular testing techniques like regression, mutation or acceptance testing. Additionally, we involved papers on different testing levels too e.g. unit, integration and system testing.
By using a low global threshold for the minimal amount of keyword matches required, an optimistic selection of relevant papers for further processing could be constructed. Every paper is tagged with its year of publication and venue. Any further modification of the data set was conducted manually; moreover, irrelevant publications in the first reading sessions were eliminated. Over the time, numerous publications were added to the list; for instance, papers of the tools that were used as a comparing factor in certain publications and every paper that was ever referenced by any of the testing-related ones got part of the data set. Additionally, some publications of the two journals Empirical Software Engineering (ESE) and Transactions on Software Engineering and Methodology (TOSEM) were added.
Even though everything was documented in the spreadsheet, not every referenced paper could be stored. Therefore, only classified files were stored in a separate repository.

3.1.2 Paper Classification

The main documentation and analysis of each paper took place in a spreadsheet in which every record holds the corresponding classification of the paper. In the following, the classifying attributes are explained in their relevancy.

Basic Information

Every record holds trivial metadata that was extracted using science-parse, dblp or Python; that being title, year, conference, authors, BibTex entries

and a URL to the paper itself. Right now, most of the hyperlinks lead to a repository in the Webis GitLab [1].

Testing Tool Information

Each paper was classified in terms of their methodology of executing test cases making them either dynamic, static, symbolic or concolic. These paradigms are explained in Section 2.1. Uncertain cases were left untouched. The question of availability and whether the source code of the presented software testing frameworks was accessible to everyone or not turned out to be a major task in further proceedings. Unfortunately, not every reference turned out to be up-to-date; thus, it was not possible to prove the availability of each paper. In case the source code could not be accessed, the record was tagged as closed-source. Otherwise, a URL was added to every publication featuring an open-source testing framework. In case of a new version of the testing tool in which major features were added or reworked, the version number was documented as well.

Sub-Check System Information

Sub-check systems and benchmarks were a major part of the analysis. Therefore, a separate table dedicated to benchmarks, their names and their source code was added. On top of that, each record yields the used version number of the benchmark if mentioned in the paper. Unfortunately, documenting every sub-check system was not possible. Evaluations that would reference big amounts of unnamed GitHub repositories or loose code snippets were not taken into consideration. Therefore, a boolean control variable was used to tag papers with named sub-check systems. Several frameworks turned out to be under closed source; accordingly, such cases were left without a reference. In order to not completely lose track of unnamed or unspecified benchmarks, an attribute on the amount of sub-check systems was added.

Sub-Check System State and Reasoning

A major detail that turned out to be even more relevant was the state of the data set itself and whether it was modified or not. While reading, the modification causes and the intention of choosing said data set for the evaluation was extracted and added to the spreadsheet. Moreover, we determined the amount of sub-check systems used in every evaluation. In Table 3.1, the different classifications of the reasoning behind using a sub-check system are explained. As a matter of fact, modifications of sub-check systems are similarly reasoned to

[1] `https://git.webis.de/`

Table 3.1: Explanation for classifying selection causes of sub-check systems

Classification	Explanation
Defectiveness	The system is known for being faulty with a number of already defined deficiencies. The errors are mostly documented through issues in a VCS. Possible weaknesses can be either functional or depending on the performance.
Popularity	The system or the systems were chosen based on their popularity as a software testing benchmark or their relevancy ranking in a VCS. These tend to be unnamed because large amounts of such repositories are used in the evaluation (e.g. 50-100). Otherwise, the popularity comes from the ubiquity of the framework (e.g. MySQL, Apache Commons or Firefox).
Suitability	The system is proclaimed to be fitting for the evaluation. Especially with very specific contributions, frameworks with that specific issue or a certain use case are chosen for the evaluation.
Quantity	The evaluation takes up a large number of repositories, projects or functions regardless of their content and faultiness. That is useful for load testing systems that depend on large data sets and their performance metrics.
Quality	The system was chosen for its complexity. Many sub-modules, high amounts of possible paths and difficult operations are the key factor for the evaluation.
Miscellaneous	The system was chosen under a different factor that depends on specifications by the framework, the tool or the venue.

justify possible disadvantages in the prior choice for selecting a benchmark in the first place. The explanations are given in Table 3.2.

3.1.3 Reference and Evaluation

Creating the reference network required a complete list of many-to-many relationships between all of the publications; therefore, every bibliography of each read publication was extracted using the open-source tool science-parse [2]. A separate Python script was used to collect these references in Comma-separated values (CSV) file using both titles as foreign keys in our relationship.

[2]https://github.com/allenai/science-parse

Table 3.2: Explanation for classifying modification causes of sub-check systems

Classification	**Explanation**
Inject Defects	The system was artificially aggravated. As a result, the evaluator knows where the errors that need to be determined by the system are situated. That is predominantly achieved using mutation testing tools.
Dismiss Irrelevant	Complex and bloated benchmarks are reduced to the most important modules. That is especially useful with bigger sub-check systems that are especially faulty in certain regions. Consequently, the evaluator has a lot more control over the outcome of the results.
Made Suitable	The original implementation of the sub-check system was not sufficiently adapted to the evaluated system. By adding and modifying certain features some authors tailor a changed benchmark that is more appropriate to the task.
Compatibility	The sub-check system might be very suitable for the evaluation but is due to a lack of support or exceptions not working with the evaluated tool, hence they are reworked or new modules are added.

By using the API of dblp [3], BibTeX entries were extracted for each relevant testing publication and its corresponding venue. Each part of the data was imported into a Google Spreadsheet [4]. Accordingly, a separate table was used to model the foreign keys of the corresponding titles using a simple `VLOOKUP()`.

3.1.4 Data Modification

Most of the classified data was not perfectly suitable for importing the CSV files of each table into a graph database, hence certain modifications were made. In order to enforce a clean relational structure, separate tables of many-to-many relationships were added yielding each edge. In the process of importing all of the references of the papers, many undesired duplicates occurred. That and certain parsing errors were eliminated from the spreadsheet. The relationships between benchmarks and publications were reworked in the same manner. The data set was examined for errors concerning inconsistencies, spelling mistakes and semantics. As soon as these unwanted artefacts were eliminated, the data set was mostly ready for importing flawlessly.

3.2 Publication Data Processing

3.2.1 Visualization

For the first approach to creating a suitable visualization of the data, a graph database was created. By using Neo4j [5], a common platform for working with various kinds of graph visualizations, importing the modified spreadsheets from the CSV files was uncomplicated and fast. Firstly, each paper was stored as a node in the database containing an identification, the title and the corresponding contribution, hence that was the first priority for the visualization. Each kind of relation, whether it was a citation, a paper reference or a benchmark reference, was stored as a directed edge. Relations leading to benchmarks were classified based on the author's choice and reasoning behind using and modifying the benchmark itself. Therefore, proper colour coding was a necessity to enforce proper examination and interaction with the graph. By using the Community Edition of the Neo4j Server, first graphs and overviews of the data could be created and interacted with.

In order to finalize the graph, the most relevant properties of each paper were imported. The visualization had to fulfil certain requirements for the next

[3] `https://dblp.uni-trier.de/`
[4] `https://www.google.de/intl/de/sheets/about/`
[5] https://neo4j.com/

steps.
First of all, one has to be able to discriminate papers that were actually classified and papers that were just a reference. In that way, it would be easier to identify common research questions, missed opportunities for comparisons within the evaluations or relevant candidates for further improvement and addition to the framework itself.
It is important to distinguish different kinds of relationships; therefore, not only is a proper colour coding between the three kinds of relationships important, but also a separation between benchmarks and papers has to be made. In that case, navigating and selecting important nodes is significantly easier.
The last and most important requirement is the ability to prioritize only nodes with certain properties, whether we are interested only in open source frameworks, papers of static analysis or simply publications from 2016. That was achieved by selective querying described in the next section.

While working with Neo4j as the main tool for visualization, certain limitations were reached. Modifying data with a certain property turned to be laborious, especially with a continuously growing network. Additionally, the visualization itself does not give the user a lot of freedom when it comes to styling the graph to content aside from the editable GRASS-file in the Enterprise version of the tool. Therefore, on top of the running Neo4j server, a D3-visualization takes the graph data from Neo4j and simply processes it in an even more wholesome manner.
The fact that one node may contain many properties as described in Chapter 3.1.2, each node has to represent the most important information in specific. Furthermore, different layouts of the graph aside from the common D3 Force Layout were created. For instance, each node can be sorted by the publication date to get an overview over referencing trends over the years, influential publications and the amounts of papers with shared contributions in a certain year. In that way, it was easier to comprehend which software testing field of research was explored more extensively than any other over the past half decade.
In addition, one can highlight certain parts of the graph by hovering nodes, such that, it is a lot easier to keep track of a certain node. By selecting and querying, finding connections between the individual contributions is possible while opening up a new angle for the research.
On top of that, a user can get a fundamental analysis of interesting papers surrounding one node. By clicking on a node, the visualization reveals paths that might be relevant in connection with said node. With additional highlighting options and colour-coding, the many dimensions of the data can be further explored. By right-clicking a node one can immediately query after

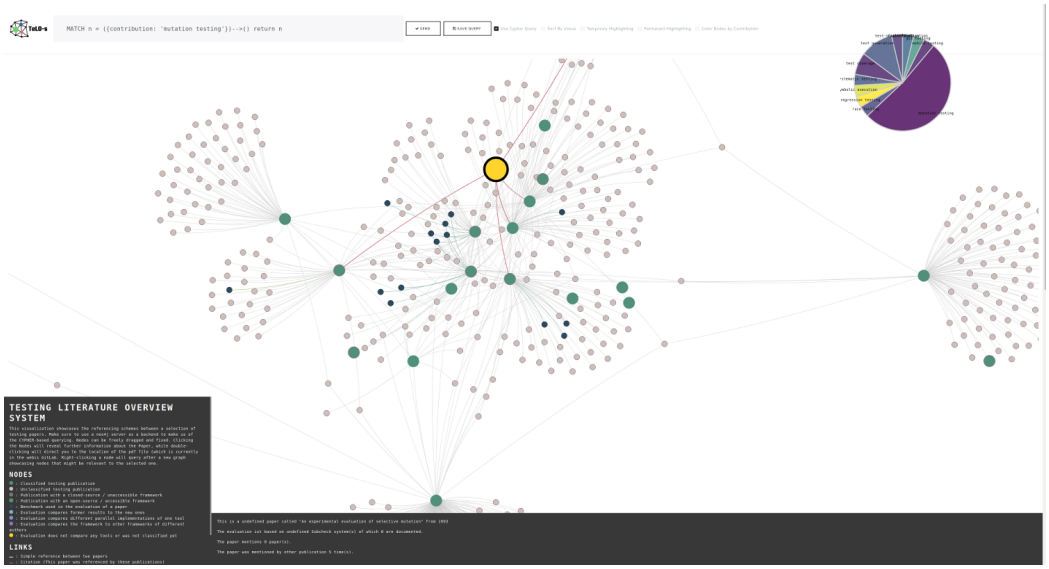

Figure 3.1: Basic interface of the visualization including a query input, control check boxes, a view revealing node information, a pie chart for the distribution of contributions and the current sub-graph

the selected publication revealing a network of papers that are related to the selected one. That selection is mainly based on the transitive dependency. In Figure 3.1 the final

3.2.2 Graph Querying

With 8000 records in our collection, getting a practical view of the data is rather tedious without selecting only specific nodes and relationships. Therefore, Cypher queries were used to not only group nodes by their properties but also help with identifying attributes of papers where the classification was uncertain the first time around. Cypher is a declarative query language that is mostly used in processing property graphs like our network. While some characteristics of SQL were adapted, most of the language relies on intuitive and easier to comprehend.
By selecting nodes of the same contribution, the process of finding shared tasks and most-used references was trivial. By highlighting outliers and adding other related contributions, papers with overlapping software testing paradigms and techniques could be determined. Filtering the nodes only by their connection to benchmarks was the best way to find common evaluation data and led to an overview of papers using specific real-world data on their evaluation instead of mentioning a variety of file repositories. Moreover, by aggregating the data, statistics were easily established and shown in the visualization.

3.2.3 Data Refactoring

With the help of the graph database, a proper visualization and extensive querying of the data, adding information on unfamiliar, solely referenced publications was much more precise and efficient than picking unrelated conferences and topics.
By exploring the graph and taking deeper looks on papers and their relations, possible initial errors could be found and proofread. On top of that, missing properties could be added and disturbing inconsistencies could be removed quicker.
By closing such information gaps, the network could be extended by papers that are relevant while opening up new routes and prominent topics for the different domains of software testing. In that manner, creating a dynamic workflow consisting out of traversing the graph, finding highly-referenced nodes, classifying them and adding the information to the network can be established. The more cycles of the routine are accomplished, the more extensive the network itself is, which enables a better insight into the research strategies of the broad field of software testing and software engineering itself.

Chapter 4

Results and Analysis

In the light of progress towards attaining a sufficient amount of consistent and comprehensive data, the spreadsheet and the corresponding visualization yielded a list of results. By making use of the methods explained in Chapter 3 the publications could be analyzed in terms of their intrinsic classification, their justified choice of benchmark, their strategy of evaluation and their definition of working with errors and their annotation. On top of that, patterns in the referencing between each publication give a better insight into the different software testing research tasks. By classifying such patterns, one is able to make assumptions on certain properties of publications that were not classified in the first place. The analysis enables a clear view of the trends of evaluating state-of-the-art software testing frameworks and emphasizes referencing strategies throughout the year.
The analysis of the data is conducted with the following research questions in mind.

RQ1 What are predominant evaluation strategies in the research field of software testing?

RQ2 How are comparing evaluations conducted?

RQ3 How is related work reflected on in the evaluation?

RQ4 Is it possible to make assumptions on evaluation strategies based on certain classifications of individual publications?

RQ5 What are the limits for reproducibility of software testing systems when it comes to their evaluation?

During the analysis, we expect that testing-related research has its own mannerisms compared to other areas of software engineering. With evaluations being conducted similarly, metrics might significantly vary. We assume

that in case recent literature of the same research field was mentioned in the related work sections, it is somehow reflected in the evaluation. With that being said, using the resulting data from the citation is obligatory for conducting a comparing evaluation. Moreover, we hope that by looking at references and classification attributes of a paper, indications on the state of the evaluation can be made. We do not consider the assumptions to be complete or correct even. It rather is help for simplifying the literature review process and algorithmic learning processes. Lastly, we do expect most papers to be difficult to reproduce regarding the high effort and the fact that aged researches were not confronted with reproducibility at all.

4.1 Classified publication data

In this section, statistics of the raw tabular data is presented. The results give insight on predominant classifications and characteristics that stood out in the process of annotating and refactoring the data set.

4.1.1 Classification Statistics

By creating an overview of the data and making use of plotting libraries like matplotlib[1] and pandas[2], a proper analysis of the data could be conducted. The main goal was to (1) give insight on how the proportions for each characteristic have changed over the years, (2) determine relevant factors for authors to use sub-check systems and benchmarks, (3) identify which metrics are relevant to which research field, (4) find out whether faults are annotated in sub-check systems by their respective author and (5) assess when and whether papers are reproducible.

Fundamental Characteristics

Figure 4.1 shows the number of papers that were read and classified compared to the overall amount of papers gathered in the data set. As the number of publications was rising because they were unclassified references of the read papers, the following statistics will focus on papers that were certainly read. The main goal is a regular addition of new papers that might open up new references to new and especially already classified publications.
In Table 4.1 and Figure 4.2 the proportions of each software testing contribution found are visualized. Certainly, test generation is a predominantly studied

[1] https://matplotlib.org/
[2] https://pandas.pydata.org/

Contribution	#
test generation	67
symbolic execution	45
mutation testing	20
static analysis	18
regression testing	17
web testing	14
multithreaded testing	11
performance testing	10
test automation	9
fault localization	8
gui testing	8
test selection	8
mobile testing	8
concolic execution	8
test coverage	6
systematic testing	5
test prioritization	4
integration testing	3
race testing	3
defect detection	3
compiler testing	3
test-suite reduction	3
dynamic execution	3
test case reduction	2
dynamic analysis	2
data flow testing	2
others	17
$\sum$	**305**

Table 4.1: Overall amount of papers for each contribution

field whereas more specific research fields like fault injection, test oracles or dependency detection are not very present in the data. Nonetheless, it is important to state that especially test case generation expands to many different other research fields like mutation testing, symbolic execution or test automation. The classification is more of a hint on the primary research field. Based on the fact that we can be sure that many papers tend to elaborate on multiple areas of software testing, a proper visualization in a graph can be beneficial.

Figure 4.3 showcases the open source trend from 2012 until 2016. It is safe to say that even though the amount of papers with publicly available source code has risen over the years, it is still improvable regarding the number of researches with a closed source. More importantly, a lot of research results and the source code is simply unreachable rather than unpublished.
The data shows that even though there is an on-going trend towards providing the reader with the source code, it is often neglected like in this paper [Alipour et al., 2016].

RQ1 The research field of software testing is invested in finding different ways to generate tests. Over the past years, the interest in automating test generation has risen with regard to mutation testing as a popular technique. Moreover, symbolic execution is a popular research area even though it is not often used in the industry. Even though most modern testing libraries rely on

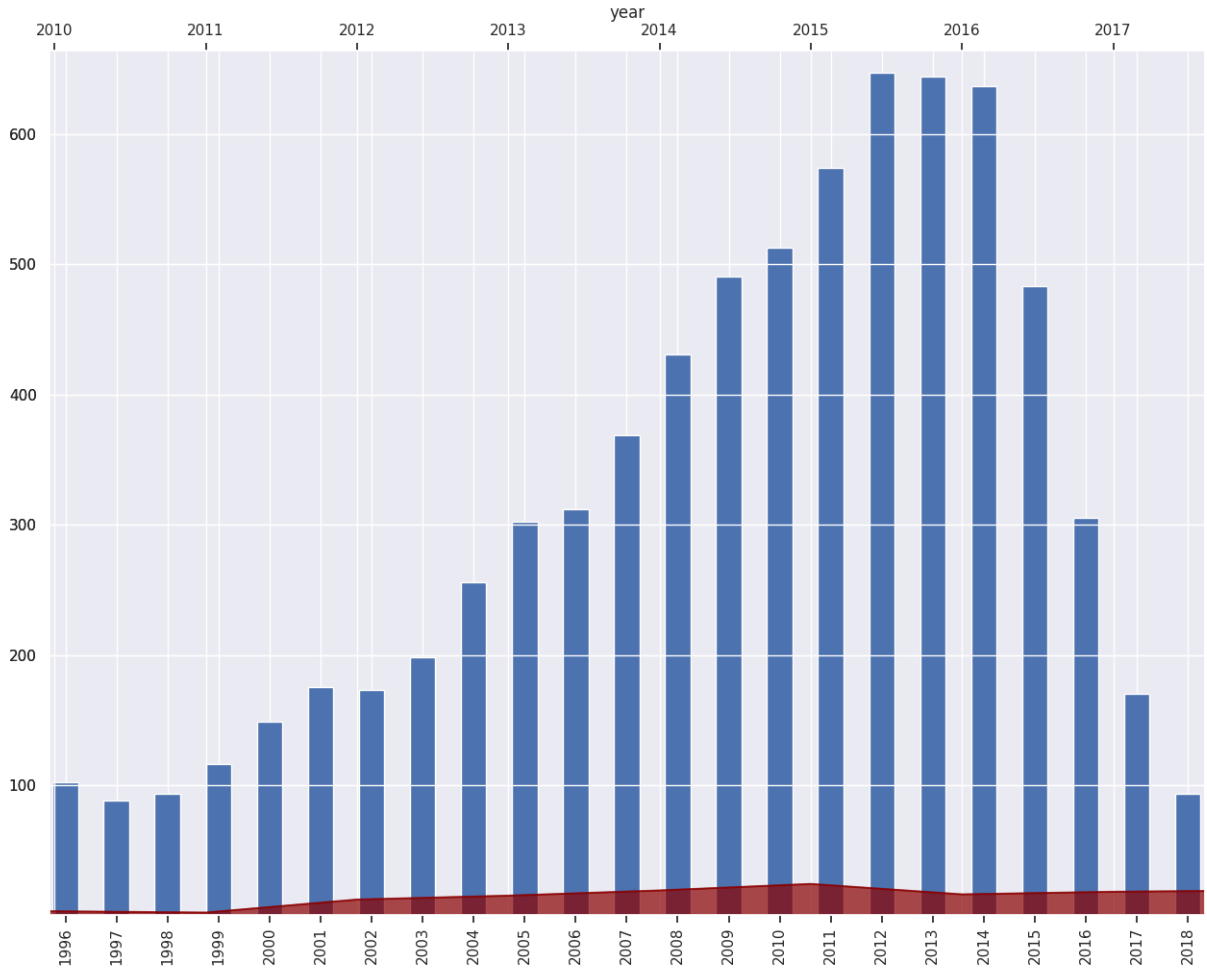

Figure 4.1: Amount of collected papers (blue) with amount of classified publications (red) from 1996 until 2018

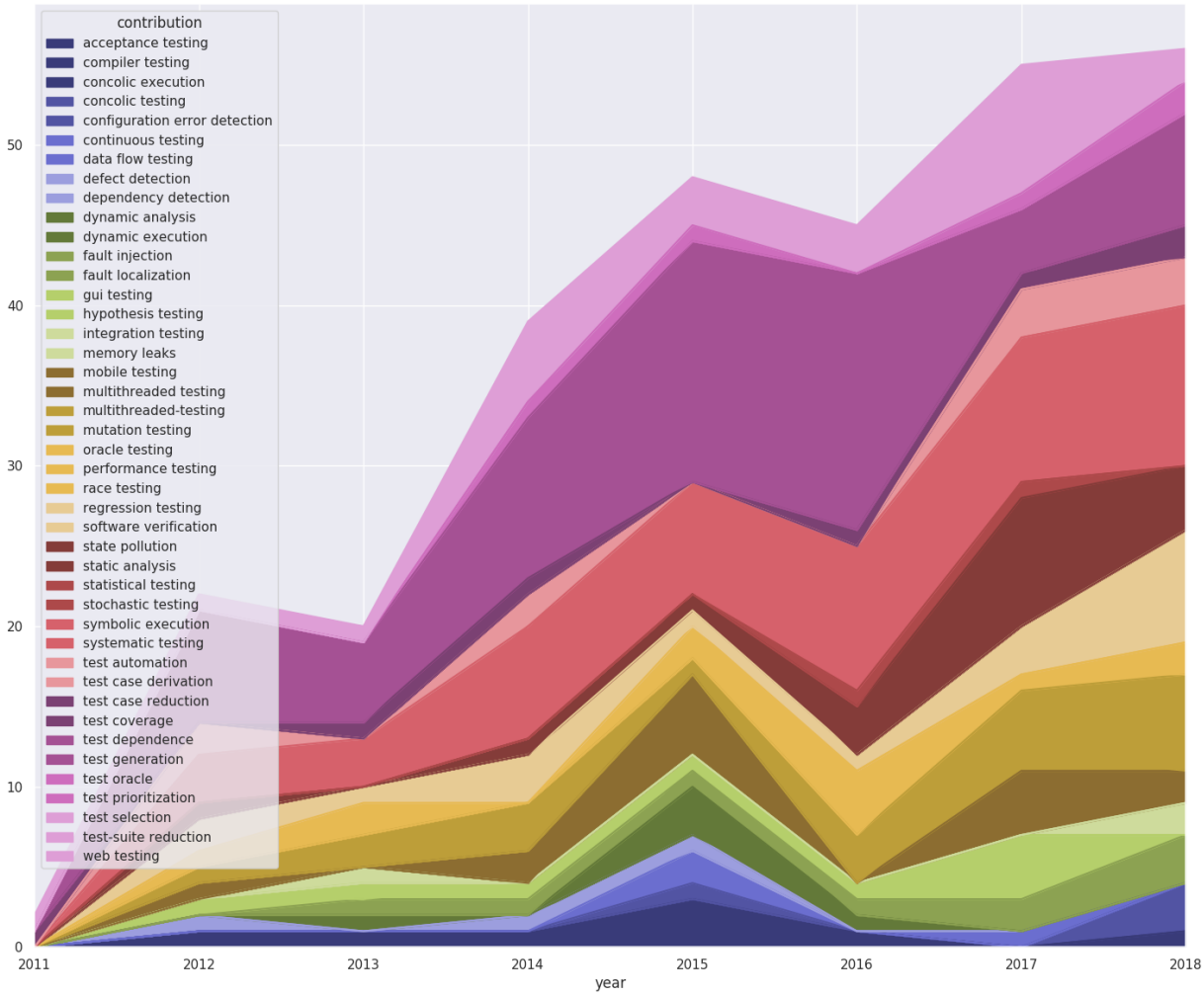

Figure 4.2: Proportions of contributions over time from year 2010 until 2018

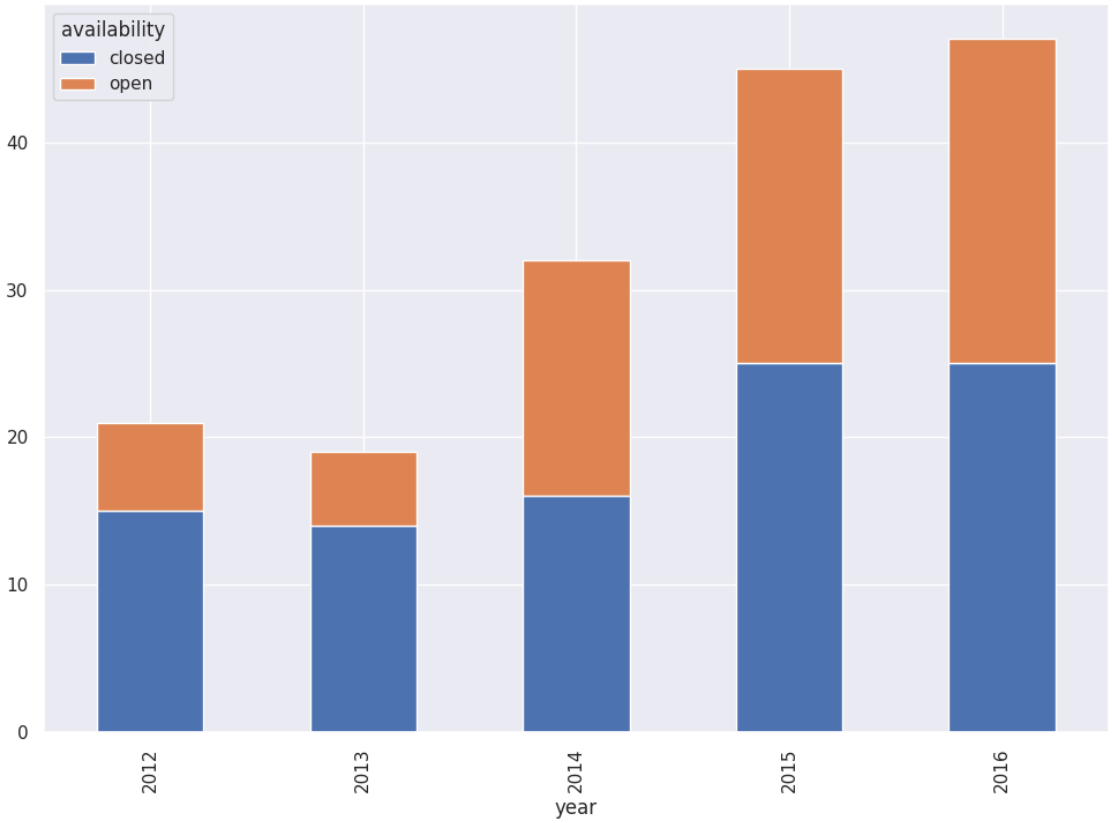

Figure 4.3: Amount of evaluated open (orange) and closed (blue) source software testing tools from 2015 until 2018

dynamic execution, other topics are researched a lot more frequent. With that being said, software testing is a broad field of different applications that are, in fact, increasingly influenced by the current industrial trends. As a result, availability is a major issue with the presented tools as the open source trend is not significant. Since many libraries and testing systems are developed with a closed source by third-party companies, it is difficult to access not only the source code but also the evaluation objects which is described in a later chapter.

RQ4 The availability of the presented tool is crucial for a reproducible evaluation. The causes of not providing the reader with the source code are numerous. Aside from dead links and expired domains, a library might be seen as unfinished by the author, hence it is finalized in later publications. Furthermore, presented concepts might be just theoretical and not yet implemented. Unfortunately, as those are understandable causes for not providing an open source, many authors tend to neglect that their code is unavailable. Nonetheless, the decision making behind that topic should be mentioned in the paper.

Benchmark and Sub-Check System Properties

For their evaluation, most authors do use benchmarks that are available for everyone. Nonetheless, not every benchmark is annotated with their corresponding version number or name even. In our selected data set of 148 files, only 118 papers were found to be with a named benchmark and only 23 with annotated version numbers. Corresponding to that, the amount of complete references including version and name is rather low.
A common convention in selecting benchmarks is the explanation of why a certain sub-check system has been chosen for evaluation purposes. As many papers tend to tailor their sub-check systems to their evaluation, some of the evaluation subjects are modified. Defining the cause of the modification is useful when it comes to understanding basic evaluation strategies. The Sankey Diagram in Figure 4.4 visualizes the predominant reasoning behind choosing and modifying a data set. From the whole data set, 143 papers could be determined in which the chosen sub-check systems were justified. The different classifications are explained in Table 3.1 and 3.2.
The selection cause for 42 papers was classified as *Miscellaneous*. Some justifications involve a very specific context like using a sub-check system provided by a venue or using systems that were utilized by the related work. For example, Tonella et al. do introduce five different benchmarks that are used for multiple reasons. Each sub-check system is properly referenced and purposefully picked based on the related work. With that in mind, in many cases, it

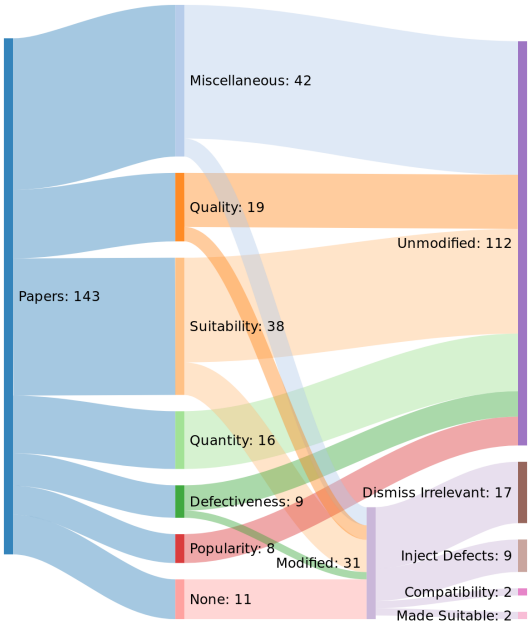

Figure 4.4: Distribution of selection and modification causes of sub-check systems

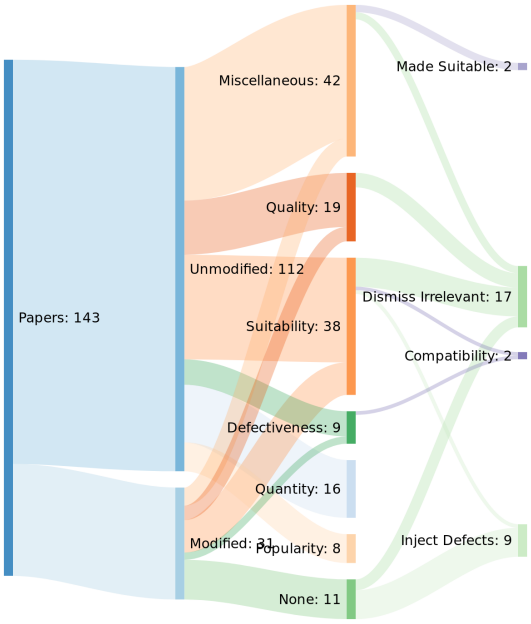

Figure 4.5: Distribution of modifications made depending on the selection cause

was not trivial to explicitly classify, yet again, the classification is a mere aid for further analyses.
38 papers chose their sub-check systems for the sake of the suitability for the evaluation. That means that the benchmarks were chosen because they fulfil certain properties or are easily adaptable to the presented testing system. For instance, Yandrapally et al. present their approach to automatically modularizing GUI test cases. As they explain common difficulties of automating GUI tests, AJAX-style applications and their forthcoming deficiencies are taken into consideration within the implementation of their tool. As a matter of fact, the evaluation consists of either AJAX and JSP applications. For these sub-check systems are specifically chosen because of their technical relation to the system, the paper's selection cause was classified as *suitability*. Consequently, the evaluation is not conducted with any other web technology for dynamically generating web pages e.g. PHP, PERL or ASP.
With only 11 papers without any cause of selection, all of them conduct an evaluation in which the sub-check system was additionally modified. That indicates that an approach exists, in which sub-check systems are chosen without any strategy and are purposefully adapted to the evaluation.

RQ1 A common strategy in making use of benchmarks and sub-check

systems is their justification. By explaining the thought behind using and modifying the evaluation objects, the author facilitates the evaluation's purpose as that choice is crucial for the experimental results. Most papers choose software that is suitable for their own system which implicates that it is possible that not every SUT may be equally well-tested as in the evaluation. A very common approach in choosing a benchmark is connected to the popularity and quantity of the sub-check system. Some researches tend to simply use very popular repositories while not even naming them. That leads to many misconceptions and is a major issue in reproducibility of the results.

RQ2 + RQ3 Some authors tailor their evaluation data based on their related work [Tonella et al., 2014]. Not only does it represent a lot of connectivity between authors and their research, but it enforces reproducibility. By using the same benchmarks as the related work the author implicitly compares himself to other work. This, on the other hand, is just speculation, as some benchmarks simply feature many different qualities that may benefit each research in a different way. For instance, one paper may work with a popular database system based on the LOC measure while another one references an unusual error that occurred in the research while evaluating their way of indicating the specific bug.

RQ4 Classifying the cause for selecting and modifying a benchmark may be crucial for understanding the relationship between the paper and their references. Not only is it possible to get an idea of why certain sub-check systems are relevant in a certain research field or explicit study tracks. It gives also additional insight on the relation to related work and the way they are used as a comparison. Taking for granted a benchmark was used multiple times by many studies, different justifications for their choice and a correspondence between the papers may reveal that presented errors in one paper may be the main aspect of the other.

Evaluation Metrics

In order to find out which metrics were mainly used in which cases, 4.6 shows the number of papers over the year that had either functionality, performance or both as the main focus of the evaluation. It is obvious that most papers either want to evaluate the functionality of the tool rather than the performance only. As most software testing systems are designed to correctly identify errors, the functionality being the main focus of most researches is understandable.

Most of the performance evaluations came from many different research

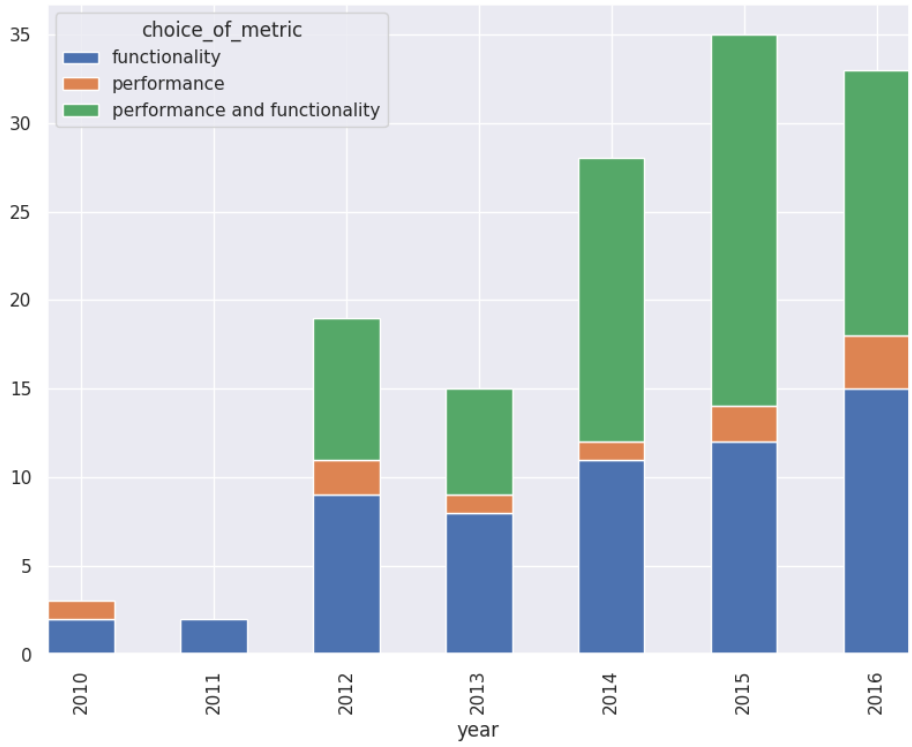

Figure 4.6: Main focus of evaluation based on the chosen metric

fields which implies that performance is the main issue in many areas of software testing. Publications featuring an evaluation based on both aspects are mostly present in test generation papers as that is the most popular research area. Not only do their authors want to improve the correctness of generated test suites but also show how rapidly they can be created.
One possibility of why performance-based evaluations are not as common was the idea of them being exclusively comparing ones. By comparing the run time of a tool compared to another one, the measured data could be a convincing metric for the contribution. Figure 4.8 shows the distribution of the comparing evaluations depending on their choice of metric. Contrary to the hypothesis of performance evaluations being rather comparing, they mostly are not. Publications featuring a two-fold evaluation, on the other hand, are most likely to compare themselves to each other.
By the fact that different key aspects of evaluation might be using different metrics, Figure 4.7 showcases the metrics that were predominantly used in functionality- and performance-driven evaluations. It has to be stated that with the low amount of performance-only evaluations, proper data could not be determined. It is mostly clear that many papers have different ways of showcasing their performance, whether it be a simple time measure or mutant kill rates. As metrics like test suite size and number of wins shows, many metrics are adapted to the system's forte which might be very one-sided. Coverage, on the other hand, is a very broad term which, again, is defined by most papers individually. By specifically looking into the papers, it gets clear that branch and path coverage are frequently used, whereas code or line coverage is not used too often. With that being said, the mutation score or the amount of killed mutants is a popular measure for assessing software testing systems.

RQ1 Software testing is not only a matter of functionality but also of performance. In order to create test suites as fast as possible, many authors evaluate their software in their run time. Even though the functionality lies in the centre of attention, papers covering both aspects tend to compare themselves to other publications frequently.

RQ2 By looking at specific papers of either functionality- and performance-based evaluations it is revealed that they mainly vary in the metrics used. While performance is usually determined by the exact execution time, functionality-based evaluations are ambivalent. Even though coverage of any kind is a predominant metric in software testing, authors find many different ways to evaluate very distinct properties of their tool. Nonetheless, it can be said that most systems are evaluated using branch, line or path coverage or even a mutation score based on the mutants killed. Surprisingly enough, with three papers

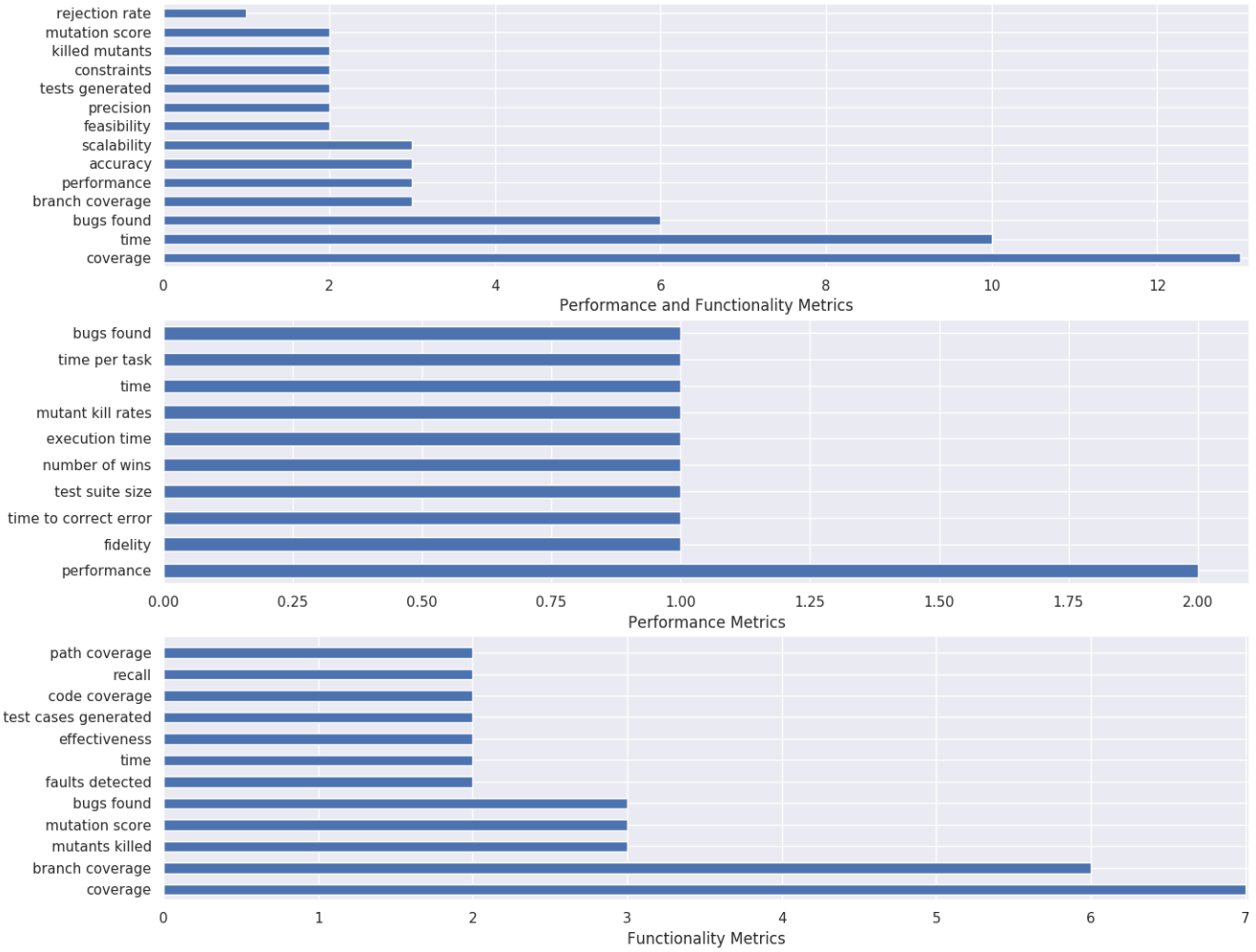

Figure 4.7: Amount of used metrics in functionality- and performance-based evaluations

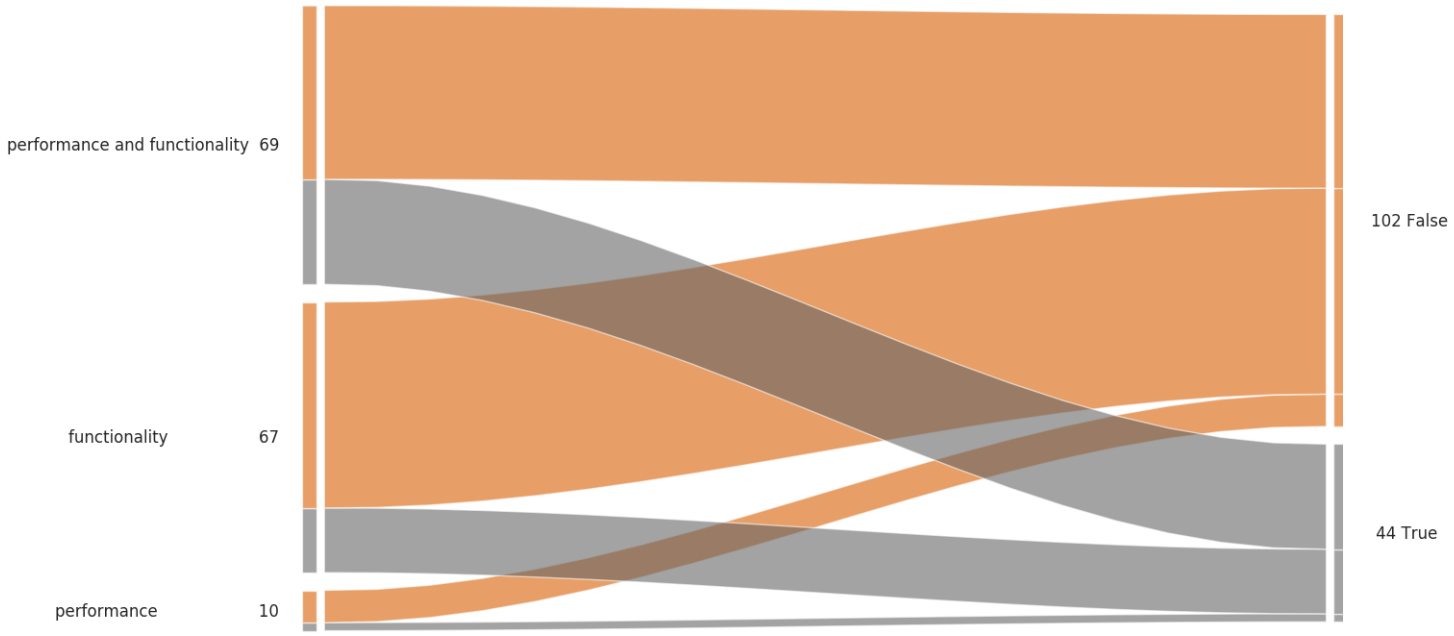

Figure 4.8: Amount of papers having a main focus on either performance or functionality (left) and whether they are comparing evaluations or not (right)

only, code coverage was not used as a metric too often, even though it is an extremely relevant measure in the industry.

RQ4 Papers evaluating only the functionality of their tool are rarely comparing themselves to other work. That comes either from the fact that they are simply incompatible in terms of their basic idea or it is simply neglected because of performance metrics being a lot more suitable for comparisons.

Error Annotation

Figure 4.9 showcases the ratio between publications specifically annotating their errors in the data set which the evaluation is conducted on and the publications that rely on either specific characterizations of their benchmarks like faultiness and prior fixed issues or nothing at all. The latter is by no means

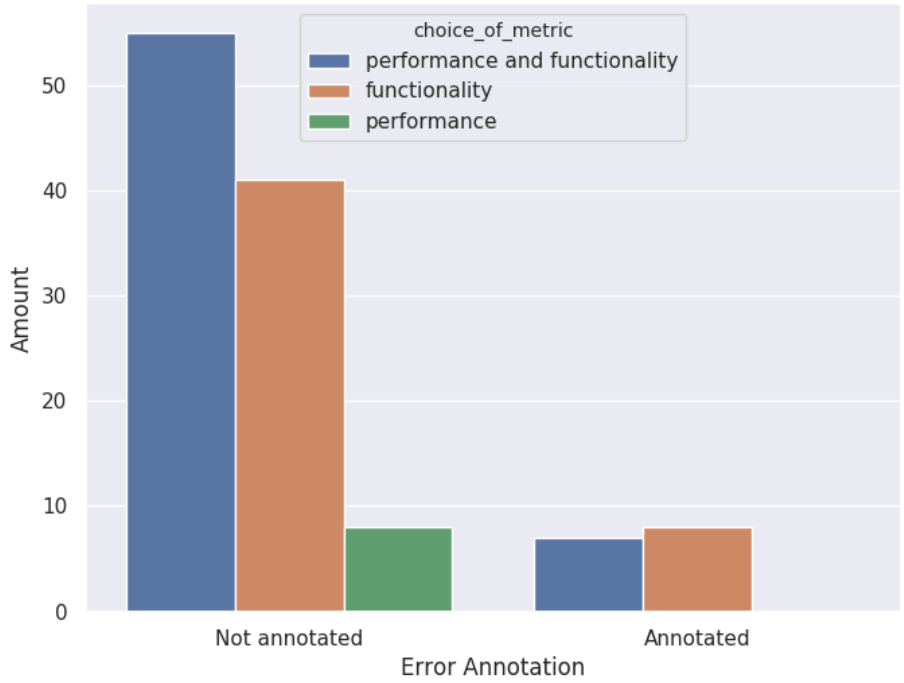

Figure 4.9: Amount of papers annotated errors with regard to the kinds of metrics used

an insured gap in the evaluation itself as it is possible that the main research of the presented system of the paper may not be necessary depending on annotated errors. For instance, some authors focus on the performance of their system including load and system tests that would not need any annotated errors. In most cases, even both aspects are relevant to the results.

RQ1 A common way of creating a suitable evaluation object is annotating which error should occur at which path of the execution. That practice is mostly used when a benchmark is provided for authors just like the SV-Comp benchmark. As a matter of fact, mutation testing is used more frequently, because mutators are explicitly assigned within the code. Therefore, error annotation is an infrequent practice in evaluating testing software.

RQ5 Annotated errors are a big influence on the reproducibility of an evaluation. Not only is it possible to recreate the results by their metric, but one can easily determine which exact parts of the code will produce which

error.

4.1.2 Graph Database

By using a graph database, an abstraction of the tabular data is achieved in which we can make use of the entity-relationship model to get a better insight into references and connections between papers. The possibilities of querying the data in both tabular and node-link directed format open up multiple views on the data as shown in Table 4.2 and Figure 4.10. By using special aggregate functions an overview of the distribution of the nodes along with their contribution, venue and classification can be achieved.

Paper	Referenced Paper
"Analyzing the [..]"	"How effective are mutation [...]"
"Analyzing the [...]"	"QTEP: quality-aware test [...]"
"Finding Deep [...]"	"A Survey of Symbolic [...]"
"Finding Deep [...]"	"Toward understanding [...]"
"Finding Deep [...]"	"Type Regression Testing [...]"
"Finding Deep [...]"	"Guided, stochastic [...]"
"Predictive Mutation [...]"	"Isomorphic Regression [...]"
"Faster Mutation [...]"	"A Large-Scale Empirical [...]"
"Faster Mutation [...]"	"Comparing and Combining [...]"
"Faster Mutation [...]"	"Selective Mutation [...]"
"Faster Mutation [...]"	"Balancing Trade-Offs [...]"
...	...

Table 4.2: Tabular representation of a CYPHER query of mutation testing publications and their bibliographic references

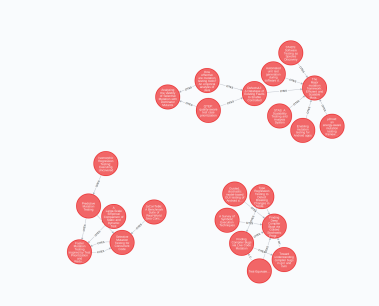

Figure 4.10: Graph representation of query after mutation testing publications and their bibliographic references in Neo4J

4.2 Publication Graph

Improving on the graph database by creating a custom visualization of the data brought up great additional insight into the data revealing many aspects of the multidimensional data. In the following, the findings from using a graph as a fundamental visualization strategy with its provided querying language are described. The results consist of the overall insight given by the force-link-directed layout and specific selections of sub-graphs representing important evaluation and referencing strategies in software testing.

Figure 4.11: Whole graph representation of the data set

4.2.1 Structural Properties

Figure 4.11 shows the whole data set in node-link-directed force-based layout. With around 8300 nodes of which 350 are classified publications and 205 are sub-check systems.

Obviously, examining the whole structure is rather tedious; therefore, queries were used to isolate nodes of interest. Figure 4.16 shows two clusters of two different contributions with a number of other related papers. By clicking on the individual nodes, one can follow the respective references. Red paths reveal publications that reference the selected node as shown in Figure 4.17, whereas black ones highlight the bibliographic references. By navigating through the graph and the number of links, one gets the impression of a well-connected network.
That is especially clear when using the force layout of the visualization. Papers that are frequently cited are found in the centre because of the many links connected to them. By using a temporal layout in which every node is ordered by its publication year, it is easier to clearly understand the references on the paper throughout the year. As Figure 4.13 shows, a paper on regression testing using mutation is frequently mentioned within the network. The many transitive relations imply that its related work is very relevant within the currently displayed domain, which is mutation testing. As we are currently not interested in finding any transitive links, Figure 4.14 shows only the direct relations of the node revealing not only many citations but also numerous references on the paper. With that being said, the node can be seen as well-implemented into the network as it reveals a lot of information of the paper as well as its relevancy just by its spatial arrangement. By ordering the nodes by their year of publication like in Figure 4.15, it is easy to make out the relevancy of the paper over time. Moreover, very old references are not in the focus which reduces the visual load. That view on the data also helps with understanding why papers with the same contributions do not reference each other. Such

patterns will be explained in a later chapter.
RQ1 It is safe to say that a lot of related work of software testing papers comes from old venues. Here, fundamental issues are still relevant to the research which implies that many tasks of the many contributions have stayed very consistent. It also has to be stated that consistently shared tasks lead to similar evaluation approaches as the authors are keen on tailoring their experimental setup to said task.
RQ2 Not only do nodes with many shared references imply a good implementation within the network but also their possible relevance to the evaluation of others. If a node has a central role in the graph, it can be seen as a "classic" paper. [de Solla Price, 1965] On the other hand, such papers and their result data is rarely used as a comparison. The more time lies between two publications, the lower the comparability. Even though their fundamental ideas might be relevant for modern researchers, their results are simply dated or not good enough to stress any significant improvement.

4.2.2 Benchmark References

Figure 4.18 shows a selection of benchmarks documented in our database connected to every paper they were ever used as a sub-check system. Querying after the papers referencing said benchmarks reveals the number of papers that were referencing the publications while not using their benchmark (e.g. Figure 4.19). Another constellation like Figure 4.20 reveals a similar pattern. The yellow node represents a paper on dynamic analysis [Samak et al., 2016] in which either dynamic execution and static analysis is used to find complex, concurrent bugs. Clicking on the node reveals that no implementation of their tool could be found during the classification process. Moreover, the paper compares itself to their related work which, on the other hand, is not reflected in their evaluation which consists of a number of popular java classes. While looking closer into the related work, it is revealed that a lot comes from former work of the same author. That also explains the predominant use of Java classes as evaluation objects. As shown in the figure, the two other papers of the constellation Samak et al., 2015, 2016 are both referenced and written by Samak et al. and do have similar java classes as benchmarks but not the same ones. As a matter of fact, authors tend to tailor their evaluation subjects to the approach they present rather than strictly depending on the comparison to other work.

RQ1 Benchmarks tend to be used without any basic system in mind. Every author decides which benchmark is the most useful in which cases. As a result, many authors stick with their choices and use the same sub-check

Figure 4.12: Multiple layouts of the visualization revealing different kinds of insight on the relevancy of a node

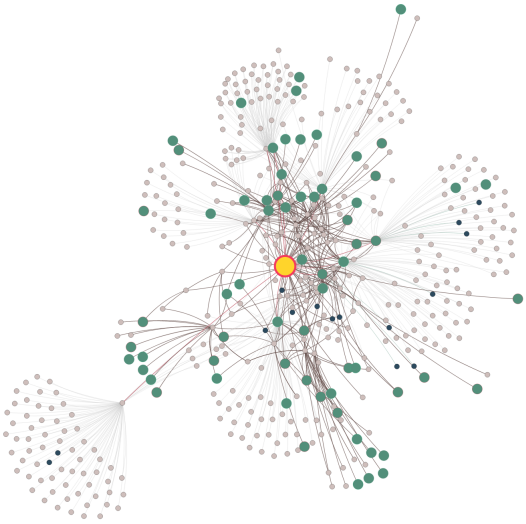

Figure 4.13: Example of a relevant paper being highly connected within the network highlighting direct and transitive relations

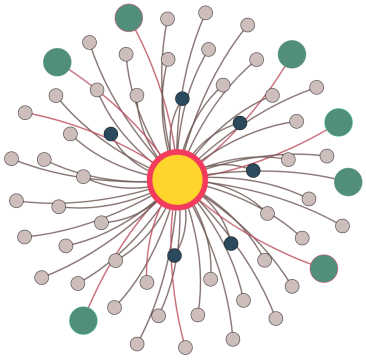

Figure 4.14: Central layout of only direct relations of a node

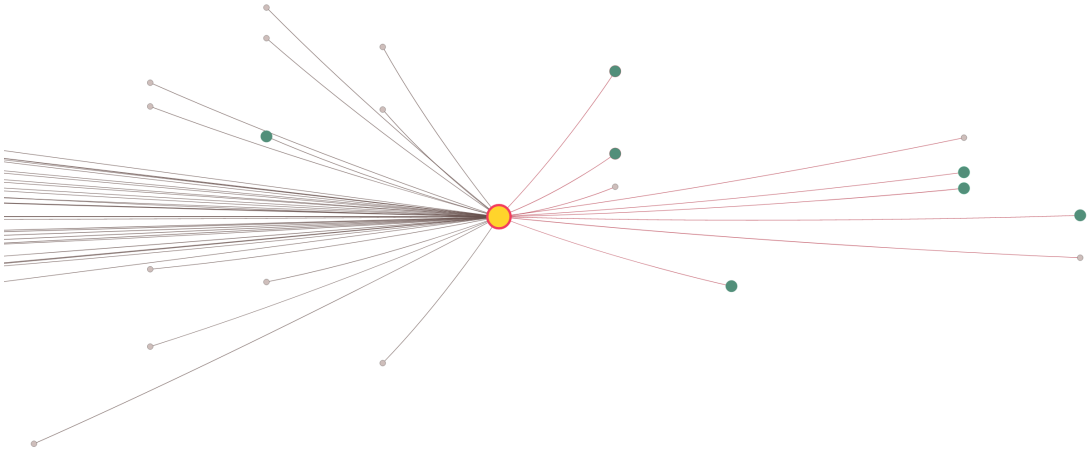

Figure 4.15: Temporal layout of the references of and by a paper

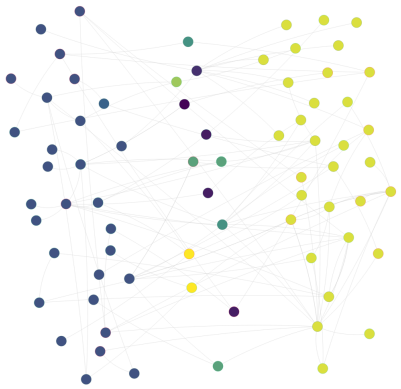

Figure 4.16: References between test generation (blue) and symbolic execution (lime) papers

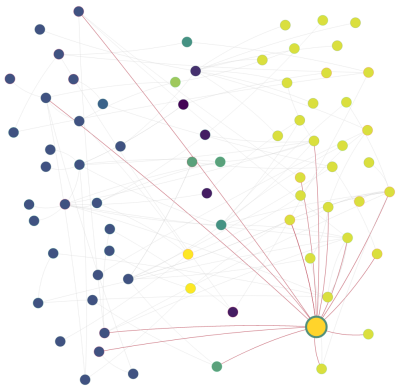

Figure 4.17: Bibliographic references on a selected symbolic execution paper

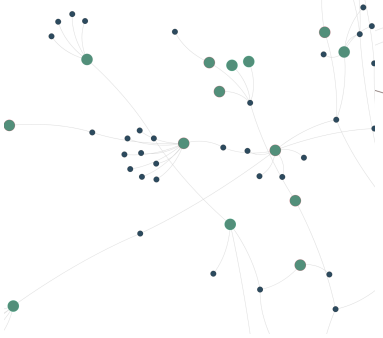

Figure 4.18: Selection of every benchmark (dark blue) connected by their use in an evaluation of a paper (green)

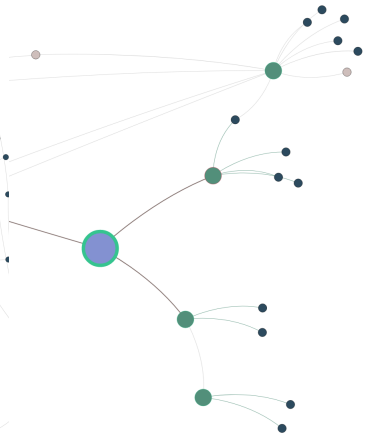

Figure 4.19: Queried node that references on a paper with documented benchmarks but does not use them

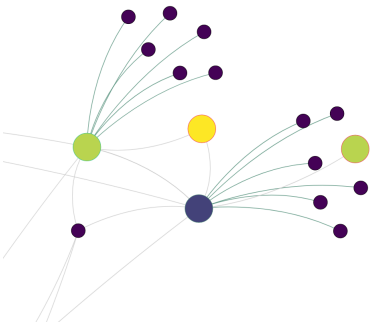

Figure 4.20: Constellation of three related nodes without any shared benchmark

systems over the years. Other authors rarely share benchmarks with others.

RQ2 + RQ3 A comparing evaluation strongly depends on the benchmarks used by the paper that is used as a comparison. Whereas some authors used the same benchmarks, many parts of the subgraph showed that the comparison is conducted with new benchmarks which negatively impacts the comparability of the paper.

RQ4 By examining the benchmarks used throughout a research area one can make assumptions on the evaluation. Since some sub-check systems are closely related to a research field, they can be used to determine what the train of thought of the author was and which qualities of his tool he wants to emphasize.

RQ5 With the fact that many papers badly annotate the version numbers of the benchmarks and references to unnamed sub-check systems are not clear, it is difficult for authors to conduct a comparing evaluation. As a result, some code turns inaccessible to the reader making the evaluation harder to recreate.

4.2.3 Referencing and Benchmarking Patterns

The insight one gets from working with node-link force-directed graph visualizations is especially useful with data sets that heavily depend on relationships. By combining such a layout to colour-coded attributes, it is possible to get a better view of the reference network. In 4.21 every mutation testing paper in the database is shown with their respective references to other contributions. As Figure 4.22 shows, the view on the graph can be distinguished into three areas: references to other contributions, publications of the considered contribution with their individual citations and the unclassified publications they reference. The goal of the classification process would be to continuously evolve the network by taking the unclassified, shared nodes into consideration.

The resulting hierarchy can be used to determine how many other contributions seem relevant for a contribution and which highly-cited references are representative of the network. For instance, Figure 4.23 shows a paper on mutation-based regression testing [Zhang et al., 2012] which is referenced by six other mutation testing papers. That implies relevance in the research field and is most likely to be read compared to other nodes. We call this structure a *vanishing point pattern* which implies a strong relevance for the research because of the high amount of references to it. Vanishing points can be found throughout the network and are extremely helpful for classifying new papers

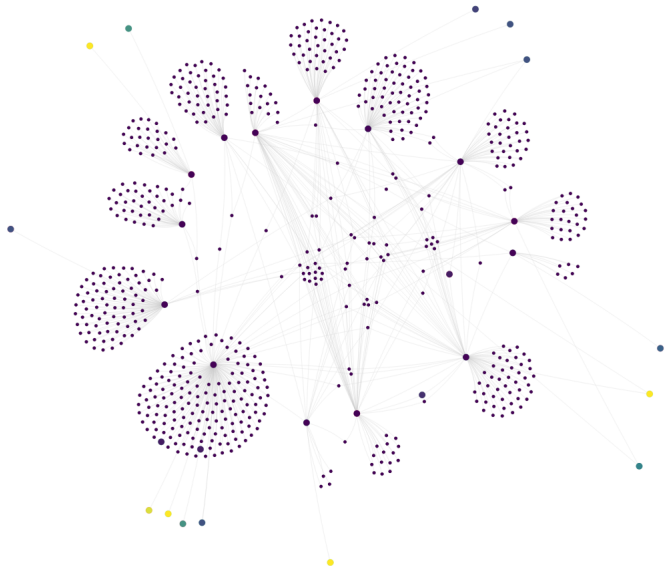

Figure 4.21: Mutation testing papers with their references between each other and to other contributions

because of them being heavily cited and therefore fundamental. Hereby it is important to state that these papers are mostly cited as early related work as they can be rather dated. Consequently, it is not crucial to use their evaluation data as a comparing factor in 2018.
Another noticeable structure within a sub-graph are nodes that do fall under a certain contribution or classification while not being referenced by any of the other papers present in the sub-graph. Even though the querying did not show any particular connection to other papers, the sub-graph around that particular paper may be insightful. Such an *outsider pattern* implies a connection to another contribution or a misclassification.

When some nodes have many references in common, it is safe to say that they must be related to each other in some way. In that case, when there is no explicit connection between the two, it is good to compare the papers by their publication year and their author. We call that topology the *familiar foreigner pattern* as the two do share common ground, but do not actually know each other because of them being published at the same time without knowing or because of any other misconceptions. That behaviour implies to our last example written by Zhang et al.. As Figure 4.24 shows, the two nodes

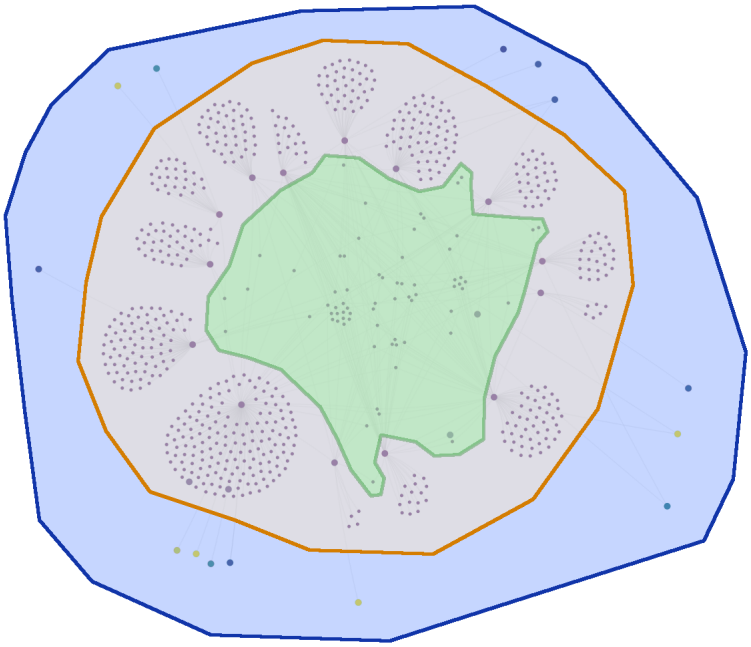

Figure 4.22: Distinguishable areas of a view on a graph in terms of their contribution, e.g. references of different contributions (blue), papers of the same contribution with their respective, individual references (orange) and shared, unclassified references (green)

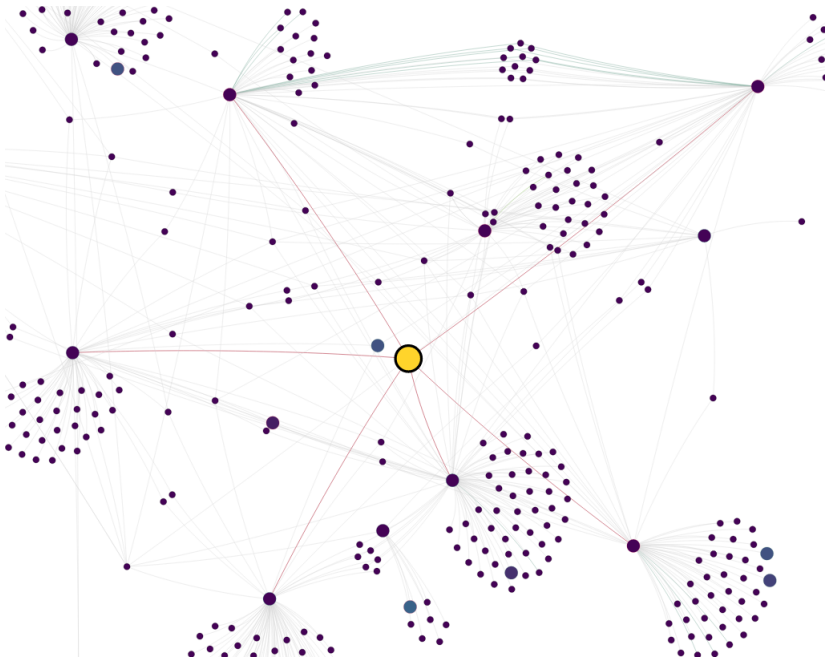

Figure 4.23: Vanishing point pattern implying a highly cited paper and its fundamental qualities as related work

share common references (grey) and benchmarks (blue). The paper was written by the same researchers which shows that authors do not deviate from their choice of benchmarks and related work; moreover, the pattern implies that the two publications are somehow related in their subject. Still, the older paper remains unmentioned in the evaluation which leads to the idea of researchers having their individual repertoire of relevant papers to cite and evaluation subjects to use. A different example is shown in Figure 4.25. Two papers from completely different universities and venues share a majority of references even though one paper deals with mutation testing for Android apps and the other with ecological species discovery using software testing fundamentals. Not only do the two have separate research tasks, but they also conduct a different experimental setup. That might explain why the two do not reference each other but it opens up many more questions regarding the credibility of assuming strategies based on referencing patterns.

A common way to determine the evolution of a research area is the analysis of papers that have referenced each other from year to year. That implies a constant evolution of the research questions and the introduction of new issues. We call that a *chain pattern* because the research of different authors is connected to each other implying that the topic is somehow relevant. Figure 4.26

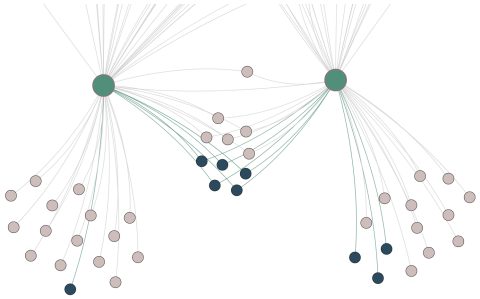

Figure 4.24: Shared references between papers of the same researchers

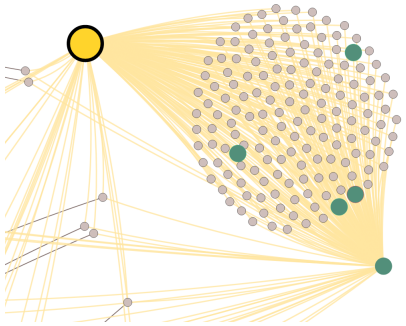

Figure 4.25: Multitude of shared references between two completely unrelated publications

is a perfect example for this behaviour. A paper from 2015 on systematic execution strategies of Android apps is referenced by an Android test automation research that even is a comparing paper. Moreover, that paper is referenced by a mutation testing approach on Android applications. Such a referencing chain strongly implies that test automation depends on different execution approaches and by figuring out a way to overcome manual writing of test cases, mutation testing can be seen as an enabler for that. Over three years, the research has been systematically specified.

RQ3 Related work is rarely reflected in the evaluation of the nodes that were inspected. Even though the visualization implies a strong connection between two papers, it is rarely shown in the paper.

RQ4 By defining referencing patterns, it is easier to navigate through a bibliographic network. Vanishing points may reveal classic papers that are used as related work but are not very comparable because of their age. Chain patterns reveal a continuous evolution of a research field and familiar foreigners show relationships between researchers not knowing each other. With that being a small part of interesting patterns, the number of assumptions one can make on the evaluation itself is enormous. They give insight into why the evaluation was not comparing even though they clearly knew of their existence and are closely related. Moreover, it is possible to understand the decisions of individual authors and their multitude of publications.

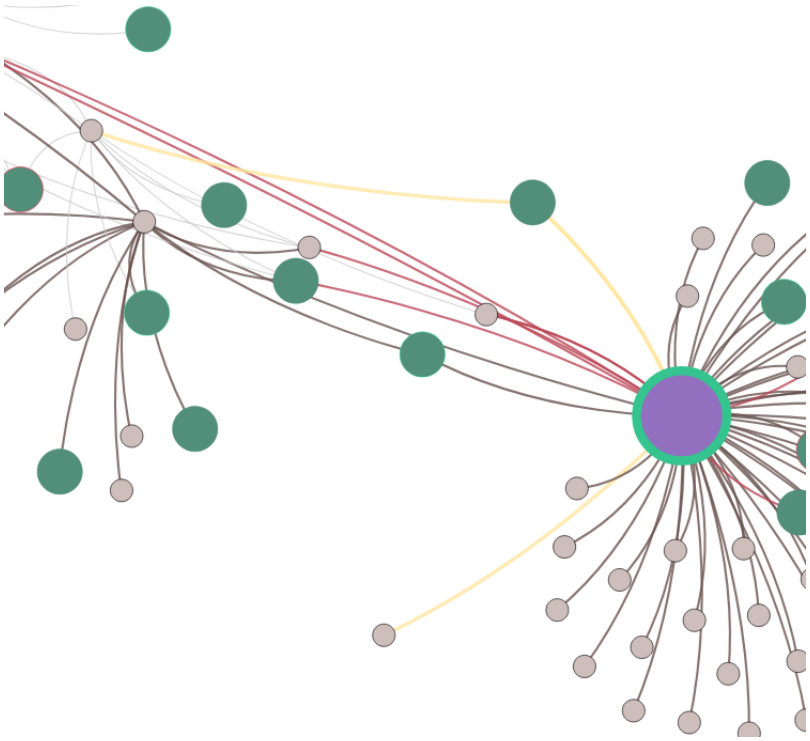

Figure 4.26: Yellow path denoting different researchers continuously referencing each other

Chapter 5

Discussion

Evaluation strategies of software testing systems are complex and depend on the properties of the paper itself and their place in the publication- and sub-check-system-network. During our research, we could identify numerous strategies in the evaluation of software testing tools. Papers that present their testing systems follow a straightforward pattern in their assessment.
Firstly, benchmarks and sub-check systems are described to the reader. With most authors trying to justify why they chose a specific evaluation object, this part is overlooked when benchmarks are generalized. Even though version control systems are a way for obtaining a convenient and concurrent error annotation through issues, bug fixes and overall history, readers cannot access the chosen repositories when they are unnamed. The same issue comes with lacking version numbers which turned out to be frequently forgotten. Moreover, in the case of benchmarks being properly described, their justification varies in many aspects. As authors tend to choose evaluation objects that would benefit their tool, it is not easy to classify their choice. Some sub-check systems tend to be oddly specific or inaccessible to the reader. Contrary to that, some authors simply choose popular real-world software that is sparsely annotated or behind paywalls. As a small number of publications showed, these problems can be circumvented by using benchmarks provided by testing conferences or individual data sets closely related to the different research areas and contributions.
After introducing the evaluation objects, results are presented. That includes the used metrics and data that could be collected. With coverage being a major key issue of software testing, it is an ambivalent term. Authors need to describe their coverage metric as different testing tools are better at improving different parts of the test suite. It also showed that a mutation score is a predominant metric which is suitable for evaluation because of the generalizability and automatic annotation. That also implies that the mutated code needs to

be accessible to the reader.
With the resulting data set, testing papers evaluate their findings. Even though we thought that related work would be frequently reflected in the evaluation, many papers do not even mention the findings of their predecessors. With that, a major issue for the comparability emerges. As soon as a paper does not compare itself to a paper of the same research task, the improvement is not comprehensible. Moreover, citations of publications with low comparability disturb the continuous improvement of the research data throughout the years.
Citations and references are an important part of a publication and its evaluation. By identifying different kinds of relations within the bibliographic network, we tried to connect noticeable referencing patterns and their reflection in testing tool evaluations.
In order to improve existing takes on bibliographic networks, we classified our papers according to numerous attributes that would deal with their semantics and structure. Defining different kinds of relations aside from simple citations is beneficial for any literature review. Because of the fact that references vary in their relevance to the paper, references to publications that are actually referred to in the evaluation are more important, hence they should be highlighted. Furthermore, benchmarks and sub-check systems play a vital role in the network; consequently, more kinds of relations are revealed. For instance, frequently used benchmarks will help authors with choosing an appropriate evaluation object and can bring up new choice justifications aside from popularity and size. As a result, researchers might use the same experimental setup as their related work more likely.
Regarding our hypotheses for our research, most of the assumptions could be validated. Software testing research follows many different strategies when it comes to their evaluation. The multitude of contributions facilitates different approaches to strengthen the arguments for your own testing system. As the analysis showed there is no streamlined or universal experimental setup for a testing paper as many of them try to prove a different point. With that in mind, the research community is very consistent with their choice of metrics as they are unique to the research field. Reproducibility remains a major issue as it requires a lot of effort, not every researcher is willing to contribute. Unfortunately, we could not validate our hypothesis on comparing evaluations being a predominant strategy. Even though the recent research of other authors was mentioned in the related work, it was rarely reflected in the experimental setup.

Based on the findings of our research one can say that with software testing being an immensely broad field, making assumptions on the evaluation strategy of an individual paper is non-trivial. Since one research field is re-

lated to one another, making precise decisions on the classification can be impossible. Nonetheless, making use of explicit classifications and bibliographic relationships reveals information that is lost in the paper as a whole and existing publication visualization techniques. By making use of that insight, it was easier to comprehend the issues of reproducibility and comparability in different research areas and contributions.

Chapter 6

Threats to Validity and Future Work

6.1 Threats to Validity

The data acquired for the research is by no means proof of mistakes. Threats to internal validity come from the process of the literature review. The data set was modified and reworked multiple times in several time periods. As the visualization showed, a more mature analysis can only be performed with a bigger amount of classified papers. Moreover, classification mistakes are a possibility of identifying some of the properties turned out to be non-trivial. The process of refactoring the data means a continuous change of the data set which inhibits the completion of the data set.
External validity comes from difficulties of generalizing the arrangements as this research was not a typical experimental setting. With the methodology describing a continuous process of improving the data set and tailoring the visualization to extract certain features, controlling possible treatments was impossible.

6.2 Future Work

The data set that was acquired from reading the software testing papers can be expanded and improved on. With every new paper, a collection of new references has to be classified and put into context. For instance, adding machine learning techniques for classifying the data depending on shared citations, benchmarks, referencing patterns and evaluation objects may be beneficial for expanding the data systematically. On top of that, the integration of the data into the graph database and consequently into other visualization methods

open up new perspectives on the evolution of research domains in software testing over time. For instance, Hierarchical Edge Bundling and Compound Graphs are possible techniques that can reveal possibly relevant publications. The current implementation of the network can be further improved as well. For example, annotating referencing patterns within the graph gives the possibility to interpret the data immediately. By pre-processing the data for every node or common queries, it is possible to handle bigger graph structures. The coherence between different views on the data (e.g. charts, statistics etc.) can reveal even more correlations in the data when it comes to selected regions of the graph.
Classifying contributions simplified the classification of the papers significantly. Nevertheless, one contribution can contain many different research tasks that have to be separated from each other. Moreover, contributions have to be merged so that more precise assumptions on the situation of the paper network can be made.

Bibliography

M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. Evaluating non-adequate test-case reduction. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 16–26, Sep. 2016. 4.1.1

Waleed Ammar, Dirk Groeneveld, Chandra Bhagavatula, Iz Beltagy, Miles Crawford, Doug Downey, Jason Dunkelberger, Ahmed Elgohary, Sergey Feldman, Vu Ha, Rodney Kinney, Sebastian Kohlmeier, Kyle Lo, Tyler Murray, Hsu-Han Ooi, Matthew Peters, Joanna Power, Sam Skjonsberg, Lucy Wang, Chris Wilhelm, Zheng Yuan, Madeleine van Zuylen, and Oren Etzioni. Construction of the literature graph in semantic scholar. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, pages 84–91, New Orleans - Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-3011. URL `https://www.aclweb.org/anthology/N18-3011`. 2.4

Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), February 2008. ISSN 0360-0300. doi: 10.1145/1322432.1322433. URL `https://doi.org/10.1145/1322432.1322433`. 2.3

S. Arlt, A. Podelski, C. Bertolini, M. SchÃf, I. Banerjee, and A. M. Memon. Lightweight static analysis for gui testing. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 301–310, Nov 2012. doi: 10.1109/ISSRE.2012.25. 2.1.2

Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `http://doi.acm.org/10.1145/3182657`. 2.1.3, 2.1.4

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. ISSN

0001-0782. doi: 10.1145/2408776.2408795. URL `http://doi.acm.org/10.1145/2408776.2408795`. 2.1.3

Derek J. de Solla Price. Networks of scientific papers. *Science*, 149(3683): 510–515, 1965. ISSN 0036-8075. doi: 10.1126/science.149.3683.510. URL `http://science.sciencemag.org/content/149/3683/510`. 2.4, 4.2.1

Institute Electrical and Electronics Engineers. Glossary of software engineering terminology, ieee standard 610.12. 09 1990. doi: 10.1109/IEEESTD.1990.101064. 2.1.2

Jesús M. González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1):75–89, Feb 2012. ISSN 1573-7616. doi: 10.1007/s10664-011-9181-9. URL `https://doi.org/10.1007/s10664-011-9181-9`. 2.2

M Graves, E.R. Bergeman, and C.B. Lawrence. Graph database systems. *Engineering in Medicine and Biology Magazine, IEEE*, 14:737 – 745, 12 1995. doi: 10.1109/51.473268. 2.3

B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995. ISSN 0740-7459. doi: 10.1109/52.391832. 2.4

Bogdan Korel. A dynamic approach of test data generation. pages 311 – 317, 12 1990. ISBN 0-8186-2091-9. doi: 10.1109/ICSM.1990.131379. 2.1.1

Thomas K Landauer, Peter W. Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(2-3):259–284, 1998. doi: 10.1080/01638539809545028. URL `https://doi.org/10.1080/01638539809545028`. 2.4

Ibéria Medeiros, Nuno Neves, and Miguel Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 1–11, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931041. URL `http://doi.acm.org/10.1145/2931037.2931041`. 2.1.2

R. M. Poston and M. P. Sexton. Evaluating and selecting testing tools. *IEEE Software*, 9(3):33–42, May 1992. ISSN 0740-7459. doi: 10.1109/52.136165. 2.4

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018. ISSN 0001-0782. doi: 10.1145/3188720. URL `http://doi.acm.org/10.1145/3188720`. 2.1.2

Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. *SIGPLAN Not.*, 50(6):175–185, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737998. URL `http://doi.acm.org/10.1145/2813885.2737998`. 4.2.2

Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. Directed synthesis of failing concurrent executions. *SIGPLAN Not.*, 51(10):430–446, October 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984040. URL `http://doi.acm.org/10.1145/3022671.2984040`. 4.2.2

Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321746. URL `http://doi.acm.org/10.1145/1321631.1321746`. 2.1.4

Gidi Shperber. A gentle introduction to Doc2Vec. `https://medium.com/scaleabout/a-gentle-introduction-to-doc2vec-db3e8c0cce5e`, 2017. [Online; accessed 11-April-2019]. 2.4

Editorial Team Synopsys. A quick post on Apple Security 55471, aka goto fail. `https://www.synopsys.com/blogs/software-security/apple-security-55471-aka-goto-fail/`, 2014. [Online; accessed 04-April-2019]. 2.1.2

Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 562–572, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568242. URL `http://doi.acm.org/10.1145/2568225.2568242`. 4.1.1, 4.1.1

Nees Jan van Eck and Ludo Waltman. *Visualizing Bibliometric Networks*, pages 285–320. Springer International Publishing, Cham, 2014. ISBN 978-3-319-10377-8. doi: 10.1007/978-3-319-10377-8_13. URL `https://doi.org/10.1007/978-3-319-10377-8_13`. 2.4

Ludo Waltman, Nees Jan van Eck, and Ed C.M. Noyons. A unified approach to mapping and clustering of bibliometric networks. *Journal of Informetrics*, 4(4):629 – 635, 2010. ISSN 1751-1577. doi: https://doi.org/10.

1016/j.joi.2010.07.002. URL `http://www.sciencedirect.com/science/article/pii/S1751157710000660`. 2.4

Rahulkrishna Yandrapally, Giriprasad Sridhara, and Saurabh Sinha. Automated modularization of gui test cases. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 44–54, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL `http://dl.acm.org/citation.cfm?id=2818754.2818763`. 4.1.1

Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 331–341, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1454-1. doi: 10.1145/2338965.2336793. URL `http://doi.acm.org/10.1145/2338965.2336793`. 4.2.3, 4.2.3

Jiang Zheng, Laurie A. Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32:240–253, 2006. 2.1.2