A taxonomy of metrics for GUI-based testing research: A systematic literature review

A Taxonomy of Metrics for GUI-based Testing Research: A Systematic Literature Review

Riccardo Coppola[a], Emil Alégroth[b]

[a]*Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy*
[b]*Blekinge Institute of Technology, Karlskrona, Sweden*

Abstract

Context: GUI-based testing is a sub-field of software testing research that has emerged in the last three decades. GUI-based testing techniques focus on verifying the functional conformance of the system under test (SUT) through its graphical user interface. However, despite the research domains growth, studies in the field have low reproducibility and comparability. One observed cause of these phenomena is identified as a lack of research rigor and commonly used metrics, including coverage metrics.

Objective: We aim to identify the most commonly used metrics in the field and formulate a taxonomy of coverage metrics for GUI-based testing research.

Method: We adopt an evidence-based approach to build the taxonomy through a systematic literature review of studies in the GUI-based testing domain. Identified papers are then analysed with Open and Axial Coding techniques to identify hierarchical and mutually exclusive categories of metrics with common characteristics, usages, and applications.

Results: Through the analysis of 169 papers and 315 metric definitions, we obtained a taxonomy with 55 codes (common names for metrics), 17 metric categories, and 4 higher level categories: Functional Level, GUI Level, Model Level and Code Level. We measure a higher number of mentions of Model and Code level metrics over Functional and GUI level metrics.

Conclusions: We propose a taxonomy for use in future GUI-based testing research to improve the general quality of studies in the domain. In addition, the taxonomy is perceived to help enable more replication studies as well as macro-analysis of the current body of research.

Keywords: Software Testing, GUI-based Testing, Coverage Metrics, Taxonomies, Software Verification and Validation

PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

Taxonomies are an important tool to synthesise, summarize or structure knowledge [1]. This knowledge can be presented in different ways, for instance hierarchically, in tree structures or as paradigms. Regardless of structure, the goal of a taxonomy is to provide guidance by establishing a common view of how to interpret and use the current state of knowledge in a specific field in a consistent manner. Taxonomies are therefore used not only for research [1] but also industrial practice [2] and education [3].

In Software engineering the number of taxonomies is steadily increasing, as shown by Usman et al. [1]. In their systematic mapping study from 2017, they identified 271 taxonomies (from 270 papers) within the field. Out of these taxonomies, 27 (9.96%) were connected to the sub-field of Software Testing, covering knowledge areas such as *test challenge classification* [2], *basic testing concept definition* [4], *tool classification* [5],*model-based testing* [6], and *search-based software testing* [7].

The area of Software Testing is however vast and can be divided into several sub-areas. One of these sub-areas is Graphical User Interface (GUI)-based testing. This research area has over the past decades evolved in terms of techniques as well as focus, from desktop systems to web-based and mobile applications [8]. A change that is perceived to be caused by trends of usage of GUI-based testing in industrial practice.

Although GUI-based testing differs between platforms and domains, there are several fundamental similarities. For instance, the purpose of the tests is generally focused on automating system-level test cases using some type of scenario-based tests aimed at covering higher-level requirements, e.g. feature requirements. Furthermore, whilst various test modelling techniques are used (e.g. graphical models, scripts and recordings) and the tests are defined by adopting different strategies (e.g., the usage of Domain Object Model, or DOM, or image recognition) they typically exhibit similar behaviours. These observations are, for instance, supported by the work of Nass et al. that demonstrated, through a systematic literature review, that challenges in GUI-based testing are persistent regardless of what is tested [8].

However, despite the fundamental similarities of different types of GUI-based testing, an identified challenge within research is that studies seldom

provide comparable results and often lack well-defined coverage metrics. This implies—although not unique to GUI-based testing as exemplified by Siegmund [9] and Miller [10]—that meta-analysis becomes difficult and can explain why replication studies are uncommon. A perceived root cause for this state of research is the aforementioned lack of commonly-used coverage metrics and/or metric definitions. Hence, looking at the body of research, we note that studies often fail to adopt, define new, or redefine existing metrics. This unfavorable phenomenon can be explained by the research areas' lack of concise definitions of suitable and available metrics.

The objective of this work is to address the use of heterogeneously defined coverage metrics in GUI-based testing research. GUI-based testing, in this paper, refers to tests that are performed through the system under test's (SUT) GUI, e.g. to test the SUT's functional conformance to its requirements. This is achieved through the synthesis of metric definitions found in the body of research, that are formulated into a taxonomy. The taxonomy shall serve as a resource for researchers to (1) browse for suitable metrics for studies on GUI-based testing and (2) as a reference for standardized definitions of said metrics. Note that the taxonomy is delimited to presenting the metrics and their definitions, it does *not* cover approaches of how to measure them. We provide the interested reader with pointers to related literature that can serve as further readings to correctly adopt and utilize the metrics.

The taxonomy is created through an evidence-based approach [11] via a systematic literature review based on the guidelines defined by Fink [12]. Through coding analysis of 169 papers, 315 metric definitions are synthesized that are clustered into 55 codes that are associated with four main categories of coverage metrics used in GUI-based testing research. These categories are *Functional level*, *GUI level*, *Model level* and *Code level* metrics. From these results, a hierarchical taxonomy is formulated with four layers that present the different types, and definitions, of commonly used metrics. The paper will present the taxonomy and discuss its use case and importance in GUI-based testing research.

The continuation of this paper is defined as follows. In section 2, we report an extended background and motivation for this work as well as related work. Section 3 presents the research design and methodology that was used. The resulting taxonomy and metric definitions are then presented in Section 4, and are further discussed in Section 5. Finally, the paper is concluded in Section 6 along with possible directions for future work.

2. Background and Related Work

Software Engineering research has been critiqued for its lack of rigour of research design and poor use of metrics [13, 14, 9, 10]. Metrics are crucial to the scientific process [15] and, if not well defined, can hamper or prevent replication [9] and macro-research since results from different studies can not be compared [16, 10]. A stated reason for this phenomenon is a lack of common nomenclature/definitions of metrics [17], resulting in researchers re-defining existing metrics, defining new ones, or omitting completely the usage of metrics.

Our observations from the sub-field of Software GUI-based testing show that the challenges of heterogeneous use of metrics and lacking guidelines for research are also prevalent. This observation is the main justification for this work where we aim to use an evidence-based software engineering [12] approach to gather definitions of coverage metrics from relevant research articles and summarize and synthesize them into a common taxonomy. We believe, given the importance of metrics in research [15], that such a taxonomy could be a crucial first step to improve the quality of research in the Software GUI-based testing sub-field. In addition, such a taxonomy would mitigate the diverse use of metrics and thereby enable other forms of research, i.e. replications and macro-research, that are currently underrepresented in the area.

2.1. GUI-based Testing

GUI-based testing is defined as a form of End-to-End functional testing of a finite System Under Test (SUT), by exercising its graphical user interface (GUI). GUI-based testing can be conducted either manually or through the utilization of automated tools.

Several categorizations have been provided for GUI-based testing in related literature. Alégroth et al. classify the existing approaches to GUI-based testing in three chronological generations, based on the type of information that is used to find user interface elements on the screen (i.e., *locators*). The first generation uses exact coordinates of the interface components to be utilized in test cases [18].

Second generation GUI-based testing tools operate directly against a model of the GUI. Such a model can be the standard description of the GUI in a specific domain (e.g., XML layout files for Android, or the DOM

model for a web page), or a specific model employed by the GUI-based testing technique to describe the SUT's GUI (e.g., finite state machines or FSMs).

Third generation GUI-based testing, also referred to as Visual GUI Testing (VGT), uses image recognition/computer vision algorithms in order to interact and assert AUT correctness through its pictorial GUI.

Another categorization of automated GUI-based testing techniques can be defined based on the way the test scripts are generated. In the field of mobile GUI testing, Linares-Vasquez et al. provide a generalizable categorization of the different methodologies for GUI-based test case generation [19].

Automation Frameworks and APIs serve as interfaces to obtain GUI-related information for the hierarchy of components of the GUI, and to perform interactions with them. Examples of these tools are Selenium [20] for web application testing, and Espresso for mobile (Android) testing [21].

Record and Replay tools have been defined to make the writing of test scripts for GUI-based SUTs less time-consuming and error-prone. These tools are based on the registration of interaction sequences with a human tester against the GUI of the software, and the automated translation of such sequences into a repeatable test script [22].

Automated Input Generation (AIG) techniques, allow the automation of at least part of the inputs provided to the SUT. AIG techniques can be roughly classified into three categories: *random-based* input generation, *systematic* input generation, and *model-based* input generation.

2.2. Related Work

To the best of our knowledge, no secondary large-scale study has been provided in the field of testing to systematize the available knowledge about coverage metrics for GUI-based testing. A tertiary study conducted by Garousi et al. demonstrates the absence of secondary studies focused on the concept of testing coverage [23].

Few secondary studies have reported classifications about coverage measurements for specific domains of software testing. As an example, in their systematic literature review about the application of search-based techniques for model-based testing, Saeed et al. describe a set of *adequacy* criteria for model-based testing, creating a taxonomy of coverage metrics. The taxonomy is composed of the following five categories: *fault-based* criteria, related to the goal of the tests in detecting faults; *data coverage* criteria, related to the coverage of boundary values in the test case generation; *script-flow* criteria, related to the coverage of statements in the SUT's source code; *model-*

flow coverage criteria, related to the coverage of transitions and paths in the models of the SUT; *requirement-based* criteria, related to the coverage of the requirements based on which the SUT is developed [24]. The study however is limited to model-based approaches for GUI testing and does not provide a finer categorization of the elicited categories to individual sub-metrics.

In their systematic literature review about Automated functional testing of mobile applications, Tramontana et al. reported a set of evaluation metrics for test approaches. They mention and describe effectiveness metrics (i.e., number of failures found), code coverage metrics (e.g., LOC coverage, method coverage, branch coverage, activity coverage), mutation coverage (i.e., ability in finding injected faults) [25]. The study however has limited generalizability, since it analyzed only tools for mobile applications, and does not take into account any coverage metric for model-based testing approaches.

Some focused and smaller-scale studies have provided conceptualizations of coverage metrics for specific testing methodologies. Rechtberger et al. provide an overview of test coverage criteria for test case generation in the model-based testing field, specifically for SUTs modelled as Directed Graphs. The article summarizes 14 most common test coverage criteria for Finite-State-Machine-based test case generation[26].

3. Study Design

The objective of this work is to identify the most commonly used coverage metrics in GUI-based testing research, through literature review, and define a common taxonomy for the GUI-based testing community. This taxonomy shall serve as a guide to align how metrics are used in GUI-based testing research to simplify secondary and tertiary studies as well as study replication. The objective is justified by the lack of guidance and established, common, metrics of the GUI-based testing research domain and an identified variance of which and how metrics are used in the discipline.

3.1. Research Questions

To facilitate the creation of a taxonomy of codes to meet the research objective, a set of research questions were defined to guide the study. The answers to these questions aim to give inputs to the formulation of the taxonomy. As a secondary objective, the study aims to identify meta-data to give an overview of the current state of usage of coverage metrics in GUI-based testing literature.

- RQ1: Metric Definition
  - RQ1.1: Which are the coverage metrics used in GUI-based testing literature?
  - RQ1.2: Which are the main categories of coverage metrics used in GUI-based testing literature?
- RQ2: Metric statistics
  - RQ2.1: Which of the elicited metric codes and categories are the most commonly used in the GUI-based testing literature?
  - RQ2.2: How varied are the definitions of metrics used in GUI-based testing literature?
  - RQ2.3: How common are metrics used in GUI-based testing literature without providing any definition?
  - RQ2.4: How has the usage of different types of coverage metrics evolved over time?
  - RQ2.5: How is the usage of different types of coverage metrics related to the domain of the SUT?
  - RQ2.6: How is the usage of different types of coverage metrics related to the GUI testing approach employed?

RQ1 aims to elicit commonly used metrics from the literature to facilitate a synthesis of metrics for the taxonomy. Complementarily, RQ2 aims to provide meta-data about how existing metrics are used and if the nature of the metrics has changed over the time span investigated in our literature review.

3.2. Methodology

The methodology used for this work can be broken down into a set of well defined steps, which are:

1. Literature review,
2. Data extraction and analysis, and
3. Formulation of taxonomy through Coding.

Table 1: Search strings

Academic repository	Search string
IEEE Xplore	GUI AND test* AND (coverage OR metric*)
ACM Digital Library	GUI AND test* AND (coverage OR metric*) on abstract and title
Science Direct	(GUI OR "Graphical User Interface") AND (metrics OR metric OR coverage) AND (test OR testing) in Title, abstract or author-specified keywords
SpringerLink	GUI AND test* AND (coverage OR metric*) on title
Google Scholar	GUI AND test* AND (coverage OR metric*) on title

All phases were conducted by using the mini-Delphi method for face-to-face meetings, with multiple one-hour meeting sessions between the authors. This approach was used to define the RQs, perform research planning, define categories of the axial coding, select the final metrics, and write the final definitions. Each meeting was closed once a consensus was reached [27].

3.2.1. Literature review

The first step of our research was aimed at finding a set of sources that could be used to find metrics for GUI-based testing coverage. To that extent, we performed a targeted literature review by applying a specific search string on a set of scientific literature repositories. We followed a subset of Kitchenham's guidelines to conducting Systematic Literature Reviews [28]. We did not perform forward or backward snowballing after the first collection.

Selected Digital Libraries. As digital libraries for our search, we selected IEEE Xplore, ACM Digital Library, Science Direct, Springer Link, and Google Scholar.

Search Strings. We formulated our search strings to be as broad as possible, including the mandatory terms *GUI* (or *Graphical User Interface*), *testing*, and *coverage* (or *metric*, or *metrics*). We did not limit our search results to specific domains or testing phases. The search strings that we defined for each digital library are reported in table 1. Because of the high number of results returned, we limited the search on Science Direct to title, abstract and keywords, on ACM Digital Library to abstract and title, and on SpringerLink and Scholar to titles. To enable reproducibility of the study, we limited our search results to the end of September 2021.

Table 2: Quality checklist

Id	Question	Scoring
QC-01	Is there a clear statement of the goals of the research	No=0; Partially=0.5; Yes = 1
QC-02	Are the research result well described?	No=0; Partially=0.5; Yes = 1
QC-03	Validity threats and limitations are described?	No=0; Yes=0
QC-04	How relevant is the study, measured by citations number?	Less than five=0; five to ten=0.5; more than ten=1;
QC-05	Does the study clearly indicate the testing methodology and procedure interested=	No=0; Partially=0.5; Yes=1;
QC-06	Does the study clearly indicate the metrics that are used and how they are applied?	No=0; Partially=0.5; Yes=1;
QC-07	Does the study present the benefits and drawbacks of adapting the metrics?	No=0; Yes=1
QC-08	Does the study clearly indicate the domain on which the metrics are applied?	No=0; Yes=1;
QC-09	Does the study provide a clear discussion of the related work and metrics?	No=0; Yes=1.
QC-10	Does the study provide an evaluation of the metrics on real applications?	No=0; Yes=1.
QC-11	Does the study provide mathematical formulations of the metrics?	No=0; Yes=1.

Inclusion Criteria. To ensure gathering only the sources relevant to our research goals, we defined the following Inclusion Criteria:

- IC1: The source is directly related to the topic of Graphical User Interface testing;
- IC2: The source defines or adopts explicitly metrics to measure the coverage of the test execution;
- IC3: The source explicitly describes the way the test cases are generated and/or executed;
- IC4: The source is an item of white literature with available full-text and it is published in a peer-reviewed journal or conference (companion) proceedings, i.e., the source is not grey literature;
- IC5: The source is written in a language that is directly comprehensible by the authors: English, Italian or Swedish;
- IC6: The source has been published before September 2021.

We do not explicitly list Exclusion Criteria since we consider them as the opposite of the inclusion criteria.

Quality assessment of sources. We performed the quality assessment phase of the sources by utilizing a simplified version of the quality checklist

template provided by Keele in the guidelines for performing systematic literature reviews for performing systematic literature reviews in Software Engineering [29], which was tailored for the quality assessment of sources related to GUI-based testing. The quality assessment checklist is reported in table 2. For each item, we obtained a normalized final quality score in the range 0-10. All the sources who had a score higher than 5 were accepted in the final pool of sources to examine.

At the end of the analysis of the final pool of sources, in fact, we experienced the inability of finding new information. We considered this situation as evidence of theoretical saturation in the sources that we considered.

3.2.2. Data Extraction and Analysis

Once the final pool of literature sources was defined, we extracted from each paper all the metrics that were mentioned to measure coverage. For each metric we collected:

- The name of the metric in the paper;
- The textual definition of the metric, if present;
- The mathematical definition of the metric, if present;
- The literature reference to the source where the metric is first defined, if present.

We defined a set of inclusion criteria for the metrics to be included in our pool:

- ICM1: The metric has to be used in the paper, and not just mentioned. This inclusion criterion was defined to avoid repeated inclusions of base metrics that are only mentioned (e.g., as possible alternatives);
- ICM2: The metric has to be empirical and measurable, used for the evaluation and assessment of testing tools, techniques and methodologies;
- ICM3: If derived from one or more lower-level base metrics, the metric has to clearly state how it is computed and on which metric(s) it depends;

- ICM4: The metric has to be explicitly used for GUI-based testing.

We do not explicitly list Exclusion Criteria since we consider them as the opposite of the inclusion criteria.

3.2.3. Open Coding

To define a taxonomy of coverage metrics for GUI-based testing, we applied Ralph's guidelines for the construction of taxonomies and followed the Grounded Theory approach [30]. We adopted the Straussian definition of Grounded Theory, defining upfront our research questions, instead of making them emerge from the results themselves [31].

We identify the *site* for our taxonomy construction as the pool of literature sources mined in the first phase of our study. The *Data Collection* necessary for the creation of the taxonomy is performed through *technical observation* of the literature sources.

The codes of the taxonomy were generated by the application of the *Open Coding* technique, as defined in the Straussian Grounded Theory. Open Coding is the procedure of analyzing text data, which is examined line by line to capture the main concepts of the theory under construction, and the categories (*codes*) that can be based on them. We adopted Open Coding as a data-driven approach for the definition of the names of the lower-level categories of the taxonomy (from now on, referred to as *codes*). We hence formulated a set of common definitions to which we assigned the individual metrics that were defined in the literature sources, by applying the procedure detailed in the next paragraph. We considered the categories of metrics as mutually exclusive (i.e., 1-N relationship between codes in the taxonomy and metric definitions in the papers).

The taxonomy has been built incrementally, adding a new code every time a new metric mined from the literature sources was considered unclassifiable under the already available codes. The open coding procedure was performed together by the authors of the manuscript, in two iterations, with the set of metrics collected from the pool of literature sources.

3.2.4. Formulation of Metric Definitions

We followed the approach below to provide an harmonization of the definition to be assigned to each code in the taxonomy.

For metrics that are not provided with a definition in the paper, we performed the following steps:

- A code with the same name is searched in the taxonomy. If such code is present, the metric is assigned to that code;
- If a code which is semantically equivalent to the name of the metric in the paper is present, the metric is assigned to that code;
- If no equal or semantically equivalent codes are present in the taxonomy, a new code is created in the taxonomy.

For metrics that are provided with a definition in the paper, we performed the following steps:

- A code with the exact same definition is searched in the taxonomy. If such code is present, the metric is assigned to that code;
- A code with a semantically equivalent definition is searched in the taxonomy. If such code is present, the metric is assigned to that code (e.g., *"percentage of the activities of the application under test that have been visited"* [32] and *"percentage of reached activities"* [33]);
- A code with a definition that can be mathematically derived from the one in the paper is searched in the taxonomy. If such code is present, the metric is assigned to that code. For instance, for a numeric metric, the total number, the average, or the percentage, were all considered under the same code in the taxonomy (e.g., *"number of screens (activities)"* [34] and *ratio of activities explored during the execution* [35]).
- If no equivalent or mathematically derived definitions are found in the taxonomy, the metric in the paper is assigned to the most closely related code. In this phase, the definitions in the taxonomy can be modified and adapted to be generalizable to the metric under consideration. As an example, we can consider the metric definitions *"Percent of event pairs in EFG covered"* [36], *"Length-n Event-sequence Coverage"* [37], and *"number and percentage of t-sets (pairs or triples of events)"* [38], which were all filed under the code "Multiple event coverage" with a more general definition.

For the metrics that were grouped together as one code in our taxonomy, we adopted a majority-based procedure to define a unique common name in the taxonomy and the associated definition. Among all the metrics filed

under the same code, those that presented the exact same name and definition to the selected one were considered as *supporting* our final definitions.

We resorted to our experience and judgement when the metrics were not defined explicitly, or to clarify ambiguities about the assignation of a metric to a code in the taxonomy.

3.2.5. Axial Coding

After applying the Open Coding procedure to find lower-level codes, we applied *Axial Coding* to the found codes to construct a hierarchy in the taxonomy of metrics (i.e., by building *categories* of codes). As defined in the Straussian Grounded Theory, Axial Coding is the process of understanding how codes and related concepts are linked to one another, to identify a structure in the taxonomy and define levels in it. Axial Coding was applied by performing two passes (by both the authors) over all the codes of the taxonomy, and defining *themes* (i.e., higher level categories) of metrics. New themes were added to the taxonomy every time one code could not be filed under already available ones. Since our objective was to build a multi-level taxonomy, Axial Coding was applied recursively until no more code merging was feasible.

4. Results

The application of the search strings allowed to gather 1483 sources; the number was reduced to 1392 after duplicate removal; to 177 after the application of inclusion and exclusion criteria; finally, to 169 after the application of the quality assessment procedure. This final pool of sources included papers published between 1998 and 2021. In figure 1 we report the distribution of the publication years of the manuscripts, which sees a peak in 2018 (17 manuscripts) immediately followed by 2010 and 2016 (16 manuscripts).

By analyzing the manuscripts, we extracted 315 different metric mentions and/or usages. After the application of the open coding phase of our study protocol, we obtained 55 codes (i.e., different metric definitions). The codes and related definitions are considered as the main contribution of our work, and are reported in tables 3~6. In the following sections we discuss the categories of codes to provide a frame at a higher level of abstraction that can give guidance about the overall usages of each cluster of metric. The interested reader is encouraged to analyze the individual definitions of the

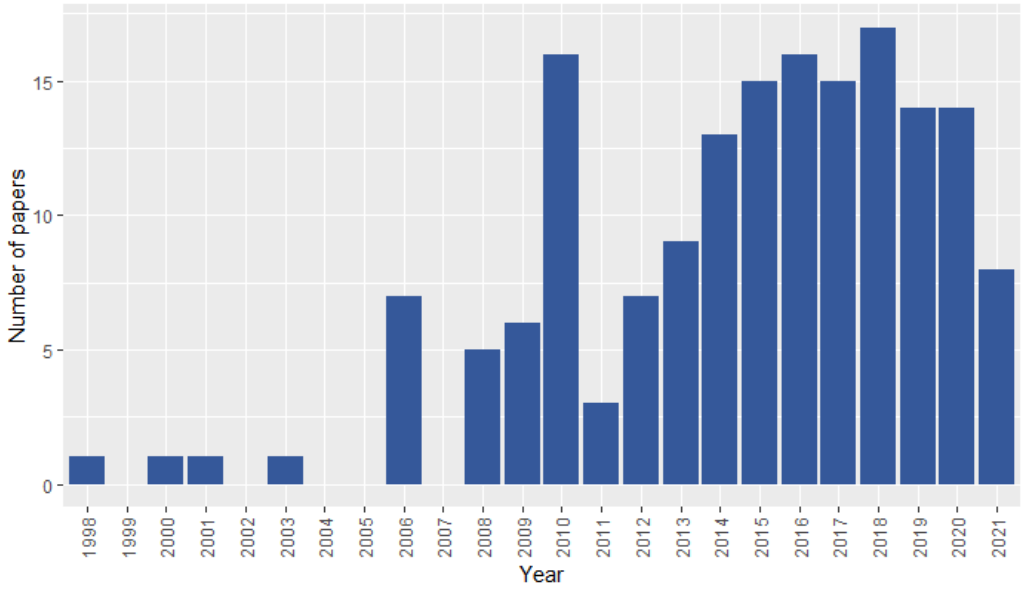

Figure 1: Number of papers per year

metrics reported in the related table after having identified a metric category of interest.

The application of Axial Coding allowed us to devise four main categories and 16 sub-categories of metrics.

4.1. Metrics definition (RQ1)

The inferred taxonomy is graphically reported in Figure 2. We identify four main categories of metrics, described in the following section. For each category of metric, we describe the sub-categories that were derived after the application of Axial Coding to the metric definitions mined from the papers. For each category, we considered an *Others* sub-category where all the metrics that could not be associated with other categories were included. For each category, we provide our names and definitions of the synthesized lowest-level metrics (in tables 3~6), along with the references to the papers providing in their text the exact same definition as the one we synthesized. As *supporting* definitions, we do not consider definitions that are partially divergent from the final definition that we obtained at the end of the code generation phase described in section 3.2.4.

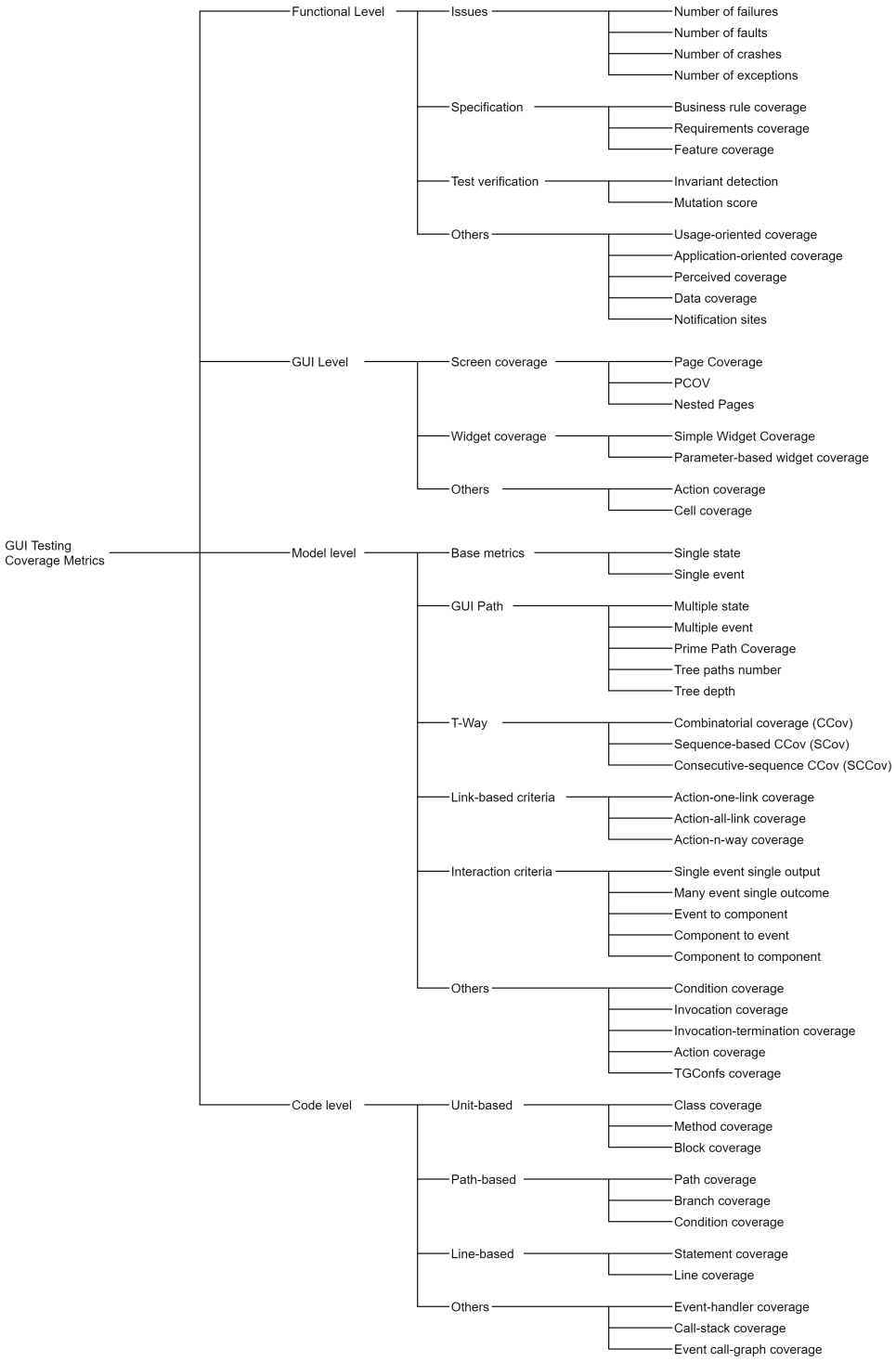

Figure 2: Taxonomy of Coverage Metrics

Functional-level metrics: We define *Functional-level* coverage metrics as the metrics that are related to the execution of use cases, and verification of functional requirements of the SUT. Functional-Based metrics can be measured at any level of software testing exercised, including End-2-End or exploratory testing with no access to the SUT's code, i.e. black-box tests. The metrics in this category are not specific to GUI-based testing.

In table 3 we report the definitions of all the metrics of this category that were mined from our pool of sources.

We identify three main categories of Functional-level metrics:

Issues: Measurements of how many issues are detected during the execution of the test suite. We consider as *issues* multiple types of unexpected behaviours in the execution of the SUT, ranging from exceptions triggered to visualization bugs, freezes and application crashes. A critical factor when considering issues is, however, that issue types vary in criticality—Many issues of a lower severity can require less effort to fix than single issues of high severity. This implication must be considered when using this metric to ensure that low- and high-severity issues are not considered in the same way.

Usage: To be used to test the quality of the SUT and/or to compare different testing tools or approaches in terms of effectiveness. The metrics are, as stated, severity-dependent, meaning that the metric can not be used in general, rather, should be applied according to the level of abstraction and severity of the issue being evaluated.

Specification: Measurements of how well the formal specifications of the SUT features are covered by the test suite.

Usage: To be used to perform end-to-end verification of a SUT.

Test verification: Verification of the capability of the GUI-based tests to detect injected modifications in the SUT or to behave in the same way in presence of invariant characteristics of the SUT.

Usage: These metrics are mainly used in the verification of test case quality, e.g. during mutation analysis.

Table 3: Functional-level metrics definition

Sub-Category	Metric	Definition	Supporting defs.
Issues	Number of Failures	How many failures are triggered —of a certain severity type— in the execution of the SUT.	-
Issues	Number of Faults	How many faults are discovered after root-cause analysis of failures manifested during the execution of the test casesIn the different studies we have considered undefined coverage for *bugs, defects and issues* as synonyms of faults.	[39, 40]
Issues	Number of Crashes	Number of crashes triggered during the execution of the SUT, where a crash leads to the termination of the SUT's process possibly with a prompt of a dialogue.	[35, 41, 42]
Issues	Number of Exceptions	Number of exceptions triggered during the execution of the SUT. It includes both caught and uncaught exceptions triggered by the tests.	[43]
Specification	Business Rule Coverage	Percentage of business rules covered by the test cases executed against the SUT.	[44]
Specification	Requirements Coverage	How many requirements of the SUT specification have been covered by the executed test cases.	-
Specification	Feature Coverage	Ratio between the number of tested functionalities of the SUT by the total number of identified functionalities for the SUT.	-
Test verification	Invariant detection	Invariant detection involves running a set of test cases and inferring, from this, a set of properties that hold on all test cases. Invariants represent high-level functionality of the underlying code and should be consistent between test runs.	[45]
Test verification	Mutation score	Mutation score or mutation coverage is defined as the ratio between the total mutants discovered and the total mutants injected.	[46, 47]
Others	Usage-oriented Coverage	Specific metric for infotainment systems. Defined as the number of functions covered among all those offered by infotainment systems considered.	[48]
Others	Application-oriented Coverage	Specific metric for infotainment systems. Defined as the coverage of different applications that may produce incoming data for the infotainment system.	[48]
Others	Perceived Coverage	Metric defined for Usability testing, where the subjects assessed the coverage of their testing sequences by assessing the coverage of each feature set of the SUT in a four-step ordinal scale	[49]
Others	Data Coverage	Proportion of the equivalence classes of inputs covered by the test cases.	[50]
Others	Notification Sites	Specific metric for Android wear systems. Defined as the number of possible types of notifications, along with possible call sites where they are issued, that are covered by a test suite.	[51]

GUI-Level: We define as *GUI-Level* coverage metrics the metrics that are based on the evaluation of pictorial components of the SUT as well as their characteristics and properties, i.e., verification that components are rendered and/or operate correctly. The metrics in this category are specific to GUI-based testing.

In table 4 we report the definitions of all the metrics of this category mined from our pool of sources.

We identified two main sub-categories of GUI-level metrics:

Screen Coverage: every SUT in any domain can be organized in different screens that are traversed by the users to access different features. Screen coverage metrics are quantitative measures based on the count of screens that are traversed by test suites. As such, this metric refers to the concrete screens of the application, that are reached during the test case through proper navigation. Screen coverage metrics hare therefore not adequate for logical screens that can be changed dynamically at runtime, e.g., *components* in web single page applications, or *fragments* in Android applications.

Usage: To cover pages of the application.

Widget Coverage: every graphical user interface can be described as a collection of different pictorial components, that can be used to show information to the final user or to gather his/her input. We generally refer to these components as *Widgets*. Widget Coverage metrics are quantitative measures based on the count of widgets that are exercised during the execution of a test suite.

Usage: To test the GUIs at a finer-grained level.

Model-level Metrics: We define as *Model-Level* the metrics that are based on a representation of the SUT's GUI according to a specific model. The metrics in this category are specific to GUI-based testing.

In our categorization of these metrics, we did not discriminate between the different specific models that are used in the studies (e.g., State Charts, Finite State Machines, or Oriented Graphs). Instead, we divide these metrics into sub-categories according to the type of atomic unit they are based on.

Table 4: GUI-level metrics definition

Sub-Category	Metric	Definition	Supporting defs.
Screen Coverage	Page Coverage	Number of reached pages during the exploration of a generic SUT[1].	[33, 32, 35, 52, 35, 34, 53, 54, 48]
Screen Coverage	PCOV	Proportion of conceptual pages that are covered by a headless crawler.	[55]
Screen Coverage	Nested Pages	Specific metric for Android Wear SUTs. Nested pages are optionally used by a notification to display supplementary information. The coverage metric computes the number of such pages opened by a test suite.	[51]
Widget Coverage	Simple Widget Coverage	Covered graphical components of the SUT, the so-called widgets (i.e. buttons, labels, text fields, scroll-bars, menus).	[56]
Widget Coverage	Parameter-based Widget Coverage	Number of parameter values for the widgets that appear in the tests.	[54]
Others	Action Coverage	Ratio between the number of actions executed by the users/testers in a real test sessions, and the number of theoretical distinct actions in a session built by a GUI analyzer.	[57]
Others	Cell Coverage	Proportion of GUI covered by same-size blocks, to evaluate the GUI grid quality	[58]

In table 5 we report the definitions of all the metrics of this category mined from our pool of sources.

Base Metrics: These metrics are based on the count of the different states (i.e., the current screen of the SUT rendered, the combination of widgets and their properties) or on the count of different events (i.e., operations performed on the GUI that can lead to a transition between the aforementioned states).

Usage: To be used as a foundation for derived model-level metrics.

GUI Path: These metrics are based on the count of the number of executed paths on the conceptual model of the GUI. The path can be defined as a sequence of single states or single events of the model, and the number of elements composing the path can vary between different metrics of the category.

Usage: To be used to evaluate the effects of multiple interactions over the GUI model.

T-Way: These criteria, defined by Mayo et al., take into account the combinatorial combinations of different events in event sequences executed by test cases [59].

Usage: To be used when the combined effects of events need to be tested.

Link-based Criteria: criteria based on the action-event links between the action model and the GUI model, as defined in the Action-Event Framework (AEF) by Nguyen et al. [60].

Usage: To generate test cases at the model level in a behaviour-oriented way.

Event Interaction: these criteria, defined by Reza et al., consider combinations between GUI states (called *components* by the authors), events, and outcomes (i.e., final states of the GUI execution) [61]. *Usage:* To be used when the combination between the current screen and application state has to be tested.

Code-level metrics: We define as *Code-Level* the metrics that are based on the quantitative evaluation of the coverage of elements of the source code. The metrics in this category are not specific to GUI-based testing.

Code-level metrics are mostly gathered by using specific tooling, e.g. EMMA or JaCoCo for coverage of Java programs, when the tested projects are open-source or the testers have access to the source code, i.e. white-box. Otherwise, Code-level metrics can be computed on byte-code for closed-source apps.

In table 6 we report the definitions of all the metrics of this category mined from our pool of sources.

We identify three main categories of Code-level metrics:

Unit-based: Coverage metrics based on the count of the number of code units that have been covered. Units are defined as entities, of varying sizes, in which the source code can be arranged.

Usage: When interested in the count of units in which the code is divided and for traceability to requirements.

Line-based: Coverage metrics based on the raw count of the number of lines of code executed by the test cases. Line-based metrics differ based on how individual lines are identified.

Usage: When interested to measure the quantity of code inside a specific unit.

Table 5: Model-level metrics definition

Sub-Category	Metric	Definition	Supporting Refs.
Base Metrics	Single State	Number (or percentage) of GUI states that are covered in the SUT by the test suite. The state is the combination of the GUI elements visualized on screen and their properties.	[45, 62, 63, 64]
Base Metrics	Single Event	Number of executed available actions over all states and screens of the SUT. It is worth noting that in many works in the selected literature, Single Event coverage is referred to as *State Coverage*. This ambiguity is caused by the adoption of models for the GUI where the model's states represent GUI events.	[65, 66, 67, 39, 68, 69, 70, 71, 72, 73, 74, 75, 64, 76, 37, 77, 78, 79, 80, 78]
GUI Path	Multiple State	Number of available sequences of n states (where n is a parameter of the coverage metric) that are explored on the SUT with the test suite.	[66, 74]
GUI Path	Multiple Event	Number of the available sequences of t events (where t is a parameter of the coverage metric) that are executed on the SUT with the test suite.	[81, 82, 38, 36, 83, 74, 61, 84, 85, 64, 37, 86, 87, 88, 50, 64, 73]
GUI Path	Prime Path Coverage	A test path satisfies prime path coverage if and only if it starts from a starting node and ends in a final node while covering a prime path in the graph modelling the SUT's GUI.	[89, 64]
GUI Path	Tree Paths Number	Total number of tree paths that can be generated on the model of the SUT's GUI	-
GUI Path	Tree depth	Maximum length of the generated test sequences when the SUT's GUI is modelled with a tree.	[90]
T-Way	Combinatorial Coverage (CCov)	Count every event combination only once without respect to the order.	[59, 91, 92, 93, 94]
T-Way	Sequence-based Combinatorial Coverage (SCov)	Counts event combinations with respect to the order in which the events are executed.	[59, 91, 92]
T-Way	Consecutive-sequence Combinatorial Coverage (SCCov)	Counts event combinations that appear adjacent to each other in a test case and respects the order of events.	[59, 91, 92]
Link-based criteria	Action-one-link coverage	This criterion requires that, for each action, only one action-event link is covered. This means the number of generated event sequences is equal to the number of action sequences.	[60]
Link-based criteria	Action-all-link coverage	This criterion requires that, for each action, all action-event links are tested.	[60]
Link-based-criteria	Action n-way coverage	The Action-All-Link coverage criterion can be considered as a one-way coverage criterion over the sets of action-event links because it covers all links of individual actions. So, it can be generalized to the Action-N-Way (ANW) coverage criterion, which requires the coverage of all possible combinations of action-event links of N actions.	[60]
Interaction Criteria	Single event and single outcome	The criterion requires each single outcome to be performed at least once.	[61]
Interaction Criteria	Many events and single outcome	The criterion requires many events leading to single outcome to be performed at least once.	[61]
Interaction Criteria	Event to component	The criterion requires that each event (transition) leading to a component is performed at least once.	[61]
Interaction Criteria	Component to event	The criterion requires each component leading to a transition to be performed at least once.	[61]
Interaction Criteria	Component-to-component	The criterion requires each component leading to another component to be performed at least once.	[61]
Others	Condition coverage	Coverage of transitions between states (i.e., events) that have guard conditions, e.g. events triggered from forms with submit buttons that involve the check of values given as inputs in the field.	[84, 73]
Others	Invocation coverage	Based on the concept of *Restricted-focus events*, that open modal windows. Invocation coverage is defined as the amount of restricted-focus events in the GUI performed at least once.	[83]
Others	Invocation-termination coverage	Coverage of all the possible length-2 sequences of events, where the first event invokes a component C, and the second event terminates such component.	[83]
Others	Action coverage	Specific metric for wearable devices, defined as the percentage of actions that are executed by a test suite.	[51]
Others	TGConfs Coverage	Percentage of TGConfs (i.e., number of test configuration data for UI test patterns) that are executed by a test case. Several derived metrics can be defined, e.g., the number of TGConfs with preconditions that are checked, or the number of TGConfs covered by a custom tester-defined script.	[95]

Table 6: Code-level metrics definition

Sub-Category	Metric	Definition	Supporting refs.
Unit-based	Class coverage	Number (or ratio) of the classes of the SUT source code covered by the executed test suites.	[36]
Unit-based	Method coverage	Number (or ratio) of the methods called during execution of test suite against the SUT.	[96, 42, 35, 36, 71]
Unit-based	Block coverage	Number (or ratio) or the blocks of the SUT source code covered by the executed test cases. A block is defined as a portion of source code not having any branch point within.	[36]
Path-based	Path coverage	Number (or ratio) of the total independent execution paths in the SUT source code, that are covered by the executed test suite.	[97]
Path-based	Branch coverage	Number (or ratio) of the code branches in the SUT source code, that are covered by the executed test suite.	-
Path-based	Condition coverage	Condition coverage measures the proportion of conditions within decision expressions that have been evaluated to both true and false	-
Line-based	Statement coverage	Number of source code statements that are executed by a given test suite for a SUT.	[40, 71, 72]
Line-based	Line coverage	Amount of code that has been exercised based on the number of Java byte code instructions called by the tests.	[98, 71]
Others	Event-handler coverage	Number (or ratio) of event handlers called by the test suite. An event handler is a specific method used to handle one specific event in an event-based SUT.	[99]
Others	Call-stack coverage	Number (or ratio) of call stacks tested by the executed test suite. A call stack is a sequence of active calls associated with each thread in a stack-based architecture.	[100]
Others	Event Call-graph coverage	Given a specific event sequence from a SUT, the call-graph coverage is the ratio between the number of source code statements covered by the sequence of events and the total number of statements from the methods in the call subgraph of the events.	[98]

Path-based: Coverage metrics based on the execution paths exercised by test cases, and how the decisions for certain paths were taken during their execution.

Usage: When interested in testing the main execution paths of the software and control cycles.

Answer to RQ1.1: We identify 55 codes (common names for metric mentions in literature) for GUI Coverage metrics. All names and definitions of metrics are reported in tables 3~6).

Answer to RQ1.2: We define a hierarchical taxonomy where codes are grouped in 17 metric categories, and 4 higher-level categories: Functional-Level, GUI-Level, Model-Level, Code-Level.

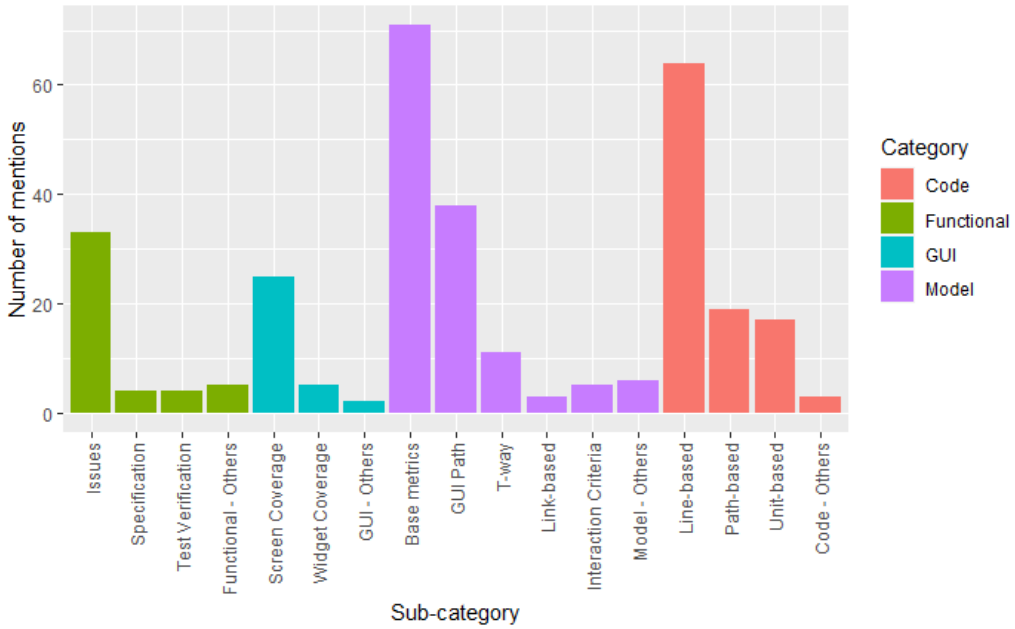

Figure 3: Number of mentions for each sub-category of metrics

Table 7: Statistics for metric sub-categories

Category	Sub-category	# mentions	total definitions	Name Variability	Undefined mentions	Definition variability
Functional	Issues	33	7	30.3%	36.4%	14.3%
Functional	Specification	4	2	0%	25%	0%
Functional	Test Verification	4	3	0%	25%	33.3%
Functional	Others	5	5	0%	0%	0%
Functional	Total	46	17	21.7%	30.4%	20%
GUI	Screen Coverage	25	12	28.0%	51.7%	8.3%
GUI	Widget Coverage	5	5	60.0%	0%	8%
GUI	Others	2	2	0%	0%	0%
GUI	Total	32	19	31.2%	40.4%	7.3%
Model	Base metrics	71	37	42.3%	47.9%	18.9%
Model	GUI Path	38	32	68.4%	15.8%	12.5%
Model	Interaction Criteria	11	9	0%	18.2%	0%
Model	Link-based	3	3	0%	0%	0%
Model	T-way	5	5	0%	0%	0%
Model	Others	6	6	0%	0%	0%
Model	Total	134	92	42.4%	31.4%	11.9%
Code	Line-based	17	5	23.5%	70.6%	0%
Code	Unit-based	19	1	5.3%	94.7%	0%
Code	Path-based	64	5	7,8%	82.8%	0%
Code	Others	3	3	0%	0%	0%
Code	Total	103	14	9.7%	80.6%	0%
	Total	315	142	27.5%	48.3%	11.1%

4.2. Statistics of coverage metrics (RQ2)

In this section, we report the statistics about the metrics that we identified in the mined research manuscripts. In figure 3 we report the number of

mentions for each of the sub-categories of metrics that we identified through coding. The most mentioned and used metrics in GUI-based testing literature are the simplest metrics of the model-level category (i.e., Base Metrics) and the code-based category (i.e., Line-Based). The Model-level category as a whole is largely the one to which the most mentioned metrics belong (134 metric mentions out of 315 (42.5%)), followed by Code-level (103 out of 315 (32.7%)), Functional-level (46 out of 315 (14.5%)), and GUI-level (32 out of 315 (10.1%)).

Among Functional-level metrics, we identify 33 mentions for Issues, 3 for Specification metrics, 4 for Test Verification, and 5 for non-categorized metrics. Among GUI-level metrics, we identify 25 mentions for Screen Coverage metrics, 5 for Widget Coverage metrics, and 2 for non-categorized metrics. Among Model-level metrics, we identify 71 mentions for Base metrics, 38 for GUI-path metrics, 5 for Interaction-based metrics, 3 for Link-based metrics, 11 for T-way metrics and 6 for non-categorized metrics. Among Code-level metrics, we identify 64 mentions for Line-based metrics, 17 for Unit-based, 19 for Path-based, and 3 for non-categorized metrics.

It is worth underlining the very scarce usage of GUI-targeted coverage metrics even in testing research which is specifically about GUI-based testing. In fact, most of the tools and techniques proposed by literature are measured by evaluating lower-level properties on the inferred model of the GUI, or at the code level.

We have collected statistics for each metric, according to how they were used in the GUI-based testing research papers we mined.

For each metric, we computed: the total number of mentions, the number of unique (different) names with which the metric was mentioned, and that were merged under a single metric in the taxonomy by means of coding; the number of the papers providing an explicit definition of the metric; the total number of unique (different) definitions provided for a metric in our taxonomy.

We finally computed for each metric three derived measures, defined as follows:

$$Name\ Variability = \frac{Unique\ Names - 1}{Total\ Mentions}; \tag{1}$$

$$Undefined\ Mentions = \frac{Total\ Mentions - Unique\ Definitions}{Total\ Mentions}; \tag{2}$$

$$Definition\ Variability = \frac{Unique\ Definitions - 1}{Total\ Definitions}. \quad (3)$$

Table 7 provides aggregate statistics each category and sub-category of metric. In the table, we report the name variability and the undefined mentions computed as the average over the total mentions in the code set, weighted by the number of mentions for each code; and the definition variability, as the average over the total number of definitions for the code set, weighted by the number of definitions for each code.

We observe an average variability of 27.5% among the definitions for the whole set of codes, with a peak variability of 40.4% for Model-level metrics among categories, and 68.4% for GUI Path among sub-categories. We found 48.3% undefined mentions on the whole set, with a peak of 80.6% for Code-level metrics among categories, and 94.7% for Unit-based metrics among sub-categories. Finally, we computed an average definition variability of 11.1% over the whole set of metrics, with a peak variability of 20% for Functional-level metrics among categories, and 33.3% for Test Verification metrics among sub-categories.

For the interested reader, we report the full statistics computed at individual metric granularity in Appendix A of the paper.

In Figure 4 we report the distribution of mentions per year, grouped by metric categories. We identify from the graph an emerging trend for GUI-level coverage (with the first mentions appearing in 2010) and increasing interest towards Functional and Code-level coverage metrics. We observe a rather constant interest in Model-level metrics throughout the years in the sample of papers that we collected.

Answer to RQ2.1: We identify 315 metric mentions divided as follows: 134 mentions of Model-level metrics, 103 of Code-level, 46 of Functional-level, 32 of GUI-level.

Answer to RQ2.2: Overall, we measure a variability of 27.5% among the definitions used for metrics that we assign to the same code in our taxonomy.

Answer to RQ2.3: Overall, 48.3% of the metric mentions that we found in the selected literature were defined in neither a qualitative nor quantitative way. When provided, definitions had a variability of 11.1%.

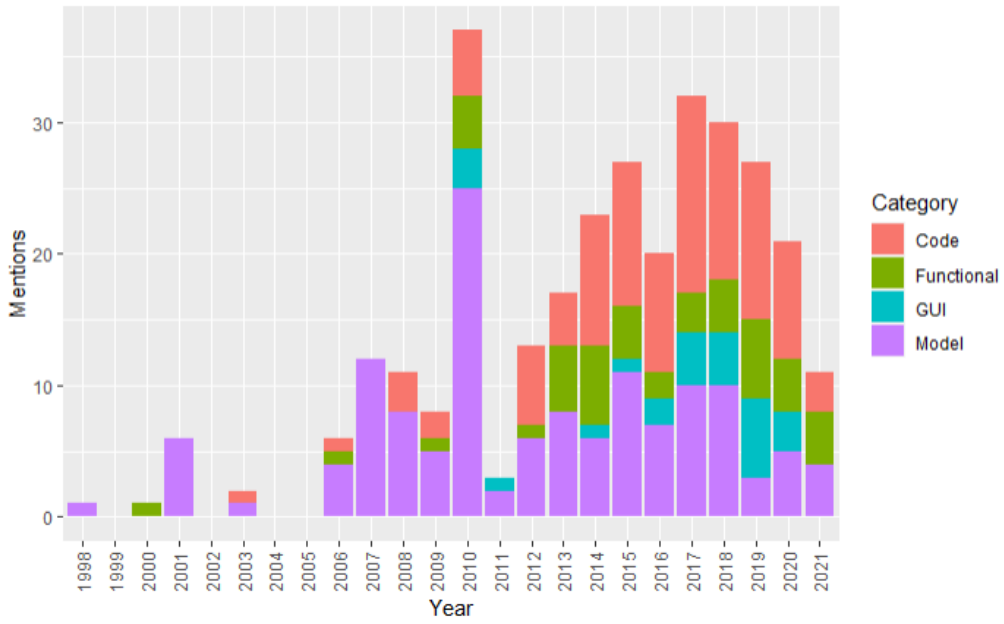

Figure 4: Number of mentions per each category per year

Answer to RQ2.4: We identify an increasing number of mentions of GUI-Level, Code-level and Functional coverage metrics, and a constant interest towards Model-level metrics.

The 169 papers were distributed among different domains in the following way: the most common domain was Desktop (81 papers, the 48% of the total set), followed by the Mobile domain (60 papers, of which 51 were specifically targeted to Android) and Web (13 papers). 4 papers were explicitly multi-domain, and 7 did not specify any domain in the text. On paper was targeted to Android Wear, one to Automotive Infotainment systems, and one to Embedded software.

In figure 5 we report the number of mentions to metrics of the four different categories for the three mostly mentioned domains (Desktop, Mobile, Web). The distribution of the mentioned metrics is different when the individual domains are considered. We identify a predominance of Model-level metrics in both the Desktop and Web domain (respectively, 64% and 55% of mentions). 31% of metrics mentioned in papers from the Desktop domain are code-level, while a very lower number of metrics mentioned were func-

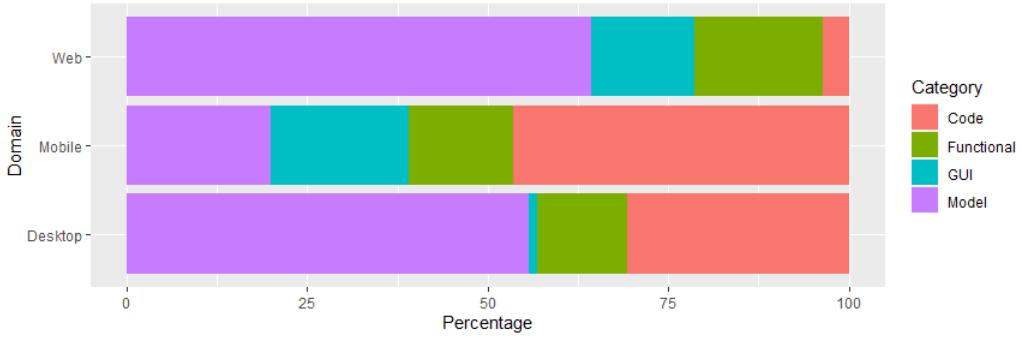

Figure 5: Percentage of mentions of each category per target domain

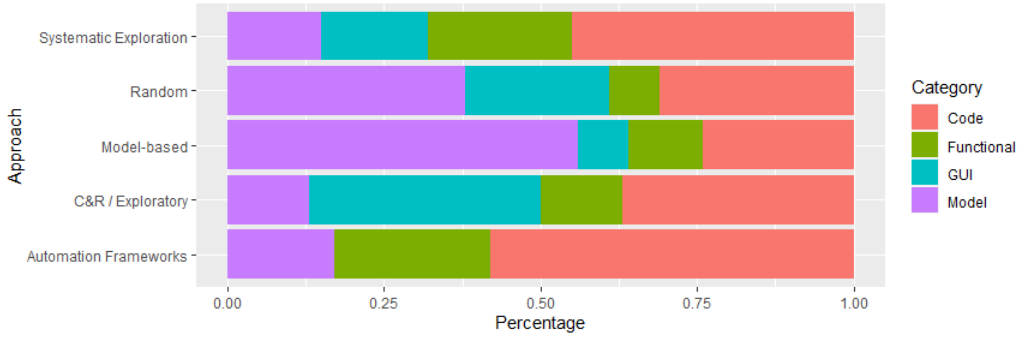

Figure 6: Percentage of mentions of each category per GUI testing approach

tional or GUI-level. By contrast, for the web domain we identified a single mention to code-level metrics. We hypothesize that the main motivation of this absence of code-level metrics in GUI-based web application testing is a focus on front-end testing, and the lack of access to the back-end when performing GUI-based testing. Finally, in the Mobile domain, we identify a lower number of mentions to model-level metrics (20%) and a higher number mentions to code-level metrics (46%). Studies in the mobile domain had the highest amount of mentions to GUI-level metrics (19%) implying a higher focus on testing screens, activities and widgets in this domain.

Answer to RQ2.5: Model-level metrics were the most mentioned in the Desktop and Web domain, as opposed to Code-level metrics for the Mobile domain.

We classified the type of GUI testing approach employed, described or evaluated in a paper by utilizing the classification provided by Linares-Vaszquez et al. [19]. The 169 papers were distributed among different approaches in the following way: the most common approach mentioned was *Automated Exploration: Model-based* (80 papers), followed by *Automated Exploration: Systematic* (44 papers, including search-based and fuzzying-based techniques), *Automated Exploration: Random* (9 papers), *Automation Frameworks & APIs* (8 papers) and Capture & Replay / Exploratory (6 papers). 18 papers in the final set could not be classified under any of the mentioned approaches, since they were related to other aspects of GUI testing (e.g., testing translation, refactoring or repair). 3 literature items did not specify any specific GUI testing approach.

In figure 6 we report the number of mentions to metrics of the four different categories for the papers utilizing each of the GI testing approaches considered. We identify an expected prevalence of model-level metrics in Model-based GUI testing approaches (56% of the total). However, model-level coverage metrics were the most used category also for Random GUI testing approaches (38%). Code-level metrics were used consistently in all approaches, and were prevalent for the Systematic Exploration methodology (45%), Automation Frameworks and APIs (58%). GUI-level coverage metrics were utilized in all approaches with exception of Automation Frameworks and APIs. A motivation for the absence can be the fact that in many cases the tools implementing this approach are executed to run test cases before the actual GUI is rendered, therefore it is not possible to access widgets and screen information to compute these metrics. GUI-level metrics were mostly mentioned in Capture & Replay and Exploratory testing tools, in which the test cases are captured through direct interaction with the GUI. Finally, we observe the presence of mentions to Functional-level coverage metrics for all approaches, with a higher prevalence for Automation Frameworks and APIs (25%) and Systematic Exploration (23%).

Answer to RQ2.6: Model-level metrics were the most mentioned metrics for Random and Model-based GUI testing tools. Code-level metrics were the most mentioned for Systematic Exploration tools, Automation Frameworks and APIs, and Capture & Replay / Exploratory (in the last case on par with GUI-level metrics)

5. Discussion

In this literature review, we have identified four categories of metrics that are used for GUI based testing research, which in total make up 55 unique metrics.

The objective of this work—the synthesis and the resulting taxonomy—is to give guidance to researchers in future studies of what metrics and definitions to use in their empirical research in the field of GUI-based testing.

In particular, we urge the use of this work in favour of redefining existing metrics or developing metrics with similar or equivalent definitions to the ones presented here. From our analysis, we identified that the latter, i.e. renaming of metrics, is common. In fact, through our synthesis, we were able to reduce the number of metrics from 315 to 55, i.e. a reduction of 82.9 percent. We hypothesise that homogenizing the use of metrics will have a positive impact on the ability to do meta-analysis and replicate studies, which are stated as general challenges in the Software engineering area [9, 10]. However, since the effects of this work are not possible to predict, nor study without extensive longitudinal study, we make no explicit claims on its effect. Looking at more mature disciplines, like physics or biology—Disciplines where the use of common metrics is more well-established—enables us to logically infer that it will have an impact.

We do not consider the presented taxonomy to be comprehensive, meaning that further metrics and definitions may be added in the future. This property is common to all taxonomies, regardless of field, and we hypothesise that it holds true for GUI-based testing research as well. Support for the hypothesis is given by the observed inclusion of new metrics over time as techniques change or evolve. We do not seek to limit researchers from adding metrics to the area, but encourage researchers to first consider existing metrics to, once more, simplify macro research.

To further enable study comparisons, we also urge researchers to consider the levels of abstraction of metrics used. The study concludes that an overall majority of papers utilize lower-level metrics for evaluation. Whilst this gives insights into the comparative performance of the GUI-based testing approaches to, for instance, unit testing, it does not provide comparative measures against other GUI-based testing approaches. The reason is simply that lower-level metrics are inherently affected more by the context of the evaluation than higher-level metrics. Using higher-level metrics, e.g. feature or component coverage, can at least make it more possible to evaluate ap-

proach efficiency. For instance by measuring the development time of the test specification of one approach, given the requirements, to another. However, since contextual factors will still play a part, e.g. size and type of requirements, contextual factors cannot be completely ignored in such comparison. We also note that the distribution of metrics (Figure 3) is skewed towards the lower-level metrics and model-level metrics. In fact, the GUI-level metrics are the least represented within our sample. Although this result is not surprising due to the comparison-based nature of tool- and approach-based research, we believe that more emphasis could be placed on the GUI-focused metrics.

The variability of mentions can also be explained by the tool-support for automated metric collection and/or analysis. This observation is supported by, for instance, the number of failures metric since it is the outcome of any testing tool. Other examples include page coverage, GUI events and statement coverage, which are all top mentions from different levels of the taxonomy—High-level referring more to GUI-level metrics whilst low-level refer more to source code metrics. Hence, looking at the research results from the perspective of automation, no clear correlation can be observed between the level of a metric and automation. Instead, the ease of collecting and/or impact/value of the metrics are more prominent markers of their use in research.

How to decide what metrics to use for a certain context to maximize replicability and meta-analysis is outside the scope of this work. We, therefore, urge the research community to investigate guidelines or best practices for GUI-testing research in the future, preferably on lines of the more general Software Engineering research guidelines proposed by Runeson et al. [101].

We furthermore believe that the recommendations discussed here will grow in importance over time as the field grows. This conclusion is judged on the study's finding (Figure 1) that the number of GUI-testing papers, which use metrics, has grown every year for the last 15 years. As the body of knowledge grows, the need for, and possibility, for synthesis and macro-research increases. However, as discussed, such progress is stagnated by the research practices employed in the area, supporting the need for our taxonomy.

In summary, this study provides a taxonomy over metrics used in GUI-based testing research. The results show that a vast amount of different metrics are used, but more troublesome that metrics are duplicated by name or definition, making comparison and macro-analysis problematic. We, there-

fore, urge the research community to use our work as a reference in an attempt to homogenize the combined efforts of the growing research area.

5.1. Threats to validity

In this section, we will discuss the main threats to our study and/or formulation of the taxonomy. As the basis for our discussion, we have used the types of validity threats presented by Runeson et al. [101].

Internal validity: These threats concern the design of our study to be able to answer the research questions. The first identified threat concerns potential research bias during coding since this was based on the researcher's best judgement. Although preventive measures like independent review and the Delphi method were used, we cannot guarantee that coding would have been conducted differently by other researchers. Despite this potential threat we still argue for the value of the taxonomy as such an artefact is currently missing in the body of knowledge on GUI-based testing research.

Another identified threat is that we may have missed papers with additional metrics or definitions. However, since no taxonomy is considered comprehensive and because we based it on synthesis from multiple sources, we see the possibility of missed metrics as a lesser threat to the value of the taxonomy. In fact, the main impact we consider is that possibly missed metrics are used in more specific cases and therefore not of as general value to the research community as the ones presented here. Note that the perceived value of a metric was not considered in the formulation of the taxonomy, i.e. efforts were made to mitigate potential bias of which metrics were included due to the researchers' preferences or experiences of certain metrics.

External validity: These threats concern the generalizability of the study's results. Due to the focus of the study on GUI-based testing, we perceive the generalizability within that domain to be high, i.e. that the results have high coverage. However, for other, adjacent, research areas (e.g. model-based testing) we make no coverage claims despite the perceived overlap of many of the found metrics. We also stress that taxonomies are seldom comprehensive and instead are built incrementally over time. As such, we expect this taxonomy to be extended and refined over time, as also discussed in the guidelines for taxonomies defined by Ralph et al. [30].

Another perceived threat is that the taxonomy is based on established state-of-art research, meaning that emerging technologies, e.g. machine learning and artificial intelligence, are not well covered. These technologies may require other metrics, which would thereby not be covered in this work. To

address this potential threat we refer to the previous discussion on taxonomy comprehensiveness.

The results related to RQ2.5 may suffer of limited generalizability because of the imbalanced number of papers specifically aimed at the different GUI-based testing domain in the final set of literature items that were analyzed. The same threat to external validity applies to the results related to RQ2.6 about the different testing approaches in the literature sources.

Construct validity: These threats refer to the study contexts' ability to answer the research questions. As an initial threat we identify that the study is only based on academic literature, i.e. no metrics used in practice were taken into account. Hence, although metrics used in empirical studies were considered, only metrics used in research were taken into account. Since the focus of the study is on research, we consider this threat to be minor.

Furthermore, the study process was based on academic best practices for literature reviews [12, 30]. However, there are still potential threats that the inclusion criteria for the study may have been too loosely defined or that the time frame for the sample may been too narrow. The impact of these potential threats is that trends or other conclusions may not have been representative enough for the context. For instance, not covering valuable papers with different nomenclature, e.g. GUI-based testing was in 1998 referred to as "interactive software". However, we once more consider this threat to be minor due to the observed increase in metrics over time and the reuse of more popular metrics. Hence, it is likely, but not certain, that the older metrics are still covered in our work. In addition, due to the study's emphasis on state-of-art, we are more interested in metrics currently used than metrics used over 20 years ago.

6. Conclusion and Future Work

Software engineering, and particularly its sub-field of software testing, is still a maturing research domain. Because of this, the use of metrics, e.g. coverage metrics, is not well defined. This leads to researchers not defining, redefining or making up new metrics.

In this work, we have conducted a systematic literature review of the GUI-based software testing domain to identify and synthesise commonly-used coverage metrics. These metrics, and their definitions, have been merged into a taxonomy in an attempt to homogenize the use of these metrics in the GUI-based testing research domain.

55 metrics were identified, clustered into 17 categories belonging to 4 categories of GUI-based testing; Functional level metrics, GUI level metrics, Model-level metrics and Code-level metrics. Meta-analysis shows that out of the four categories, the GUI level metrics are used the least. However, the analysis also shows that the interest in GUI-based testing research, and mention of these metrics, is steadily growing. Thus, presenting an urgent need for a taxonomy of this type. We believe that our taxonomy can both help improve the quality of research in the area, enable better support for replication studies as well as macro-level research. In addition to providing guidance on which metrics are available and common definitions to use, we also urge researchers to use the hierarchical taxonomy to pick metrics on a suitable level of abstraction. Hence, utilize the most suitable metrics given the type and level of abstraction their research method, model or tool is applied upon.

However, as with any taxonomy, we do not claim this work to be comprehensive and therefore urge researchers to extend and complement the taxonomy in the future. Examples of improvement areas include metrics used in research on GUI-based testing with machine learning or artificial intelligence, which are not covered.

Furthermore, we see a need for more research into guidelines and ways of research in the GUI-based testing domain. This includes, but is not limited to, the design of controlled- and quasi-experiments, tool comparison studies, and replication studies. Experiments in this area generally require human involvement, which is notoriously difficult to control and borders on the lines of psychological- and sociological research. Mapping best practices from these domains into software engineering research is therefore of both interest and import.

Tool comparison studies are common in GUI-based testing research, but what subjects are used vary from commercial applications, open-source and even toy examples. Each of these has different benefits and drawbacks, but guidelines on how to leverage these benefits and drawbacks are missing. So are guidelines of when it is suitable to use one type of subject or another, e.g. based on research maturity.

Finally, replication studies in GUI-based research are uncommon, and how to perform these, utilizing tools and frameworks of other researchers, is mostly carried out ad hoc, with varying success. Once more, guidelines could be an aid for the research community, improving both the quality and success rate of such studies.

GUI-based testing is, according to this work, a growing field and we are excited to see what solutions the future will bring.

7. Acknowledgements

Emil Alégroth would like to acknowledge that this work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology.

References

[1] M. Usman, R. Britto, J. Börstler, E. Mendes, Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method, Information and Software Technology 85 (2017) 43–59.

[2] E. Engström, K. Petersen, Mapping software testing practice with software testing research—serp-test taxonomy, in: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2015, pp. 1–4.

[3] M. Azuma, F. Coallier, J. Garbajosa, How to apply the Bloom taxonomy to software engineering, in: Eleventh annual international workshop on software technology and engineering practice, IEEE, 2003, pp. 117–122.

[4] R. Roggio, J. Gordon, J. Comer, Taxonomy of common software testing terminology: Framework for key software engineering testing concepts, Journal of Information Systems Applied Research 7 (2) (2014) 4.

[5] K. Shaukat, U. Shaukat, F. Feroz, S. Kayani, A. Akbar, Taxonomy of automated software testing tools, International Journal of Computer science and innovation 1 (2015) 7–18.

[6] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Software testing, verification and reliability 22 (5) (2012) 297–312.

[7] M. Harman, P. McMinn, J. T. De Souza, S. Yoo, Search based software engineering: Techniques, taxonomy, tutorial, in: Empirical software engineering and verification, Springer, 2010, pp. 1–59.

[8] M. Nass, E. Alégroth, R. Feldt, Why many challenges with GUI test automation (will) remain, Information and Software Technology 138 (2021) 106625.

[9] J. Siegmund, N. Siegmund, S. Apel, Views on internal and external validity in empirical software engineering, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 9–19.

[10] J. Miller, Can results from software engineering experiments be safely combined?, in: Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403), IEEE, 1999, pp. 152–158.

[11] B. A. Kitchenham, T. Dyba, M. Jorgensen, Evidence-based software engineering, in: Proceedings. 26th International Conference on Software Engineering, IEEE, 2004, pp. 273–281.

[12] A. Fink, Conducting research literature reviews: From the internet to paper, Sage publications, 2019.

[13] C. Wohlin, A. Rainer, Challenges and recommendations to publishing and using credible evidence in software engineering, Information and Software Technology 134 (2021) 106555.

[14] T. Dybå, T. Dingsøyr, Strength of evidence in systematic reviews in software engineering, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 178–187.

[15] R. Van Solingen, V. Basili, G. Caldiera, H. D. Rombach, Goal question metric (gqm) approach, Encyclopedia of software engineering (2002).

[16] A. Jedlitschka, M. Ciolkowski, Towards evidence in software engineering, in: Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04., IEEE, 2004, pp. 261–270.

[17] L. Prechelt, On implicit assumptions underlying software engineering research, in: Evaluation and Assessment in Software Engineering, 2021, pp. 336–339.

[18] E. Alégroth, Z. Gao, R. Oliveira, A. Memon, Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–10.

[19] M. Linares-Vásquez, K. Moran, D. Poshyvanyk, Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 399–410.

[20] A. Bruns, A. Kornstadt, D. Wichmann, Web application tests with selenium, IEEE software 26 (5) (2009) 88–91.

[21] G. Nolan, Espresso, in: Agile Android, Springer, 2015, pp. 59–68.

[22] S. Di Martino, A. R. Fasolino, L. L. L. Starace, P. Tramontana, Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing, Software Testing, Verification and Reliability 31 (3) (2021) e1754.

[23] V. Garousi, M. V. Mäntylä, A systematic literature review of literature reviews in software testing, Information and Software Technology 80 (2016) 195–216.

[24] A. Saeed, S. H. Ab Hamid, M. B. Mustafa, The experimental applications of search-based techniques for model-based testing: Taxonomy and systematic literature review, Applied Soft Computing 49 (2016) 1094–1117.

[25] P. Tramontana, D. Amalfitano, N. Amatucci, A. R. Fasolino, Automated functional testing of mobile applications: a systematic mapping study, Software Quality Journal 27 (1) (2019) 149–201.

[26] V. Rechtberger, M. Bures, B. S. Ahmed, Overview of test coverage criteria for test case generation from finite state machines modelled as directed graphs, in: 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2022, pp. 207–214.

[27] S. Q. Pan, M. Vega, A. J. Vella, B. H. Archer, G. Parlett, A mini-delphi approach: An improvement on single round techniques, Progress in tourism and hospitality research 2 (1) (1996) 27–39.

[28] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering 2 (01 2007).

[29] S. Keele, et al., Guidelines for performing systematic literature reviews in software engineering, Tech. rep., Citeseer (2007).

[30] P. Ralph, Toward methodological guidelines for process theories and taxonomies in software engineering, IEEE Transactions on Software Engineering 45 (7) (2018) 712–735.

[31] M. T. T. Thai, L. C. Chong, N. M. Agrawal, Straussian grounded theory method: An illustration, The Qualitative Report 17 (5) (2012).

[32] S. Paydar, An empirical study on the effectiveness of monkey testing for Android applications, Iranian Journal of Science and Technology, Transactions of Electrical Engineering 44 (2) (2020) 1013–1029.

[33] Y. Li, Z. Yang, Y. Guo, X. Chen, Humanoid: A deep learning-based approach to automated black-box Android app testing, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 1070–1073.

[34] P. Wang, B. Liang, W. You, J. Li, W. Shi, Automatic android gui traversal with high coverage, in: 2014 Fourth International Conference on Communication Systems and Network Technologies, IEEE, 2014, pp. 1161–1166.

[35] H. N. Yasin, S. H. A. Hamid, R. J. Raja Yusof, Droidbotx: Test case generation tool for Android applications using Q-learning, Symmetry 13 (2) (2021) 310.

[36] J. Strecker, A. M. Memon, Accounting for defect characteristics in evaluations of testing techniques, ACM Transactions on Software Engineering and Methodology (TOSEM) 21 (3) (2012) 1–43.

[37] P. Li, T. Huynh, M. Reformat, J. Miller, A practical approach to testing GUI systems, Empirical Software Engineering 12 (4) (2007) 331–357.

[38] S. Huang, M. B. Cohen, A. M. Memon, Repairing GUI test suites using a genetic algorithm, in: 2010 Third International Conference on Software Testing, Verification and Validation, IEEE, 2010, pp. 245–254.

[39] W. Song, X. Qian, J. Huang, Ehbdroid: beyond GUI testing for Android applications, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 27–37.

[40] S. R. Choudhary, A. Gorla, A. Orso, Automated test input generation for Android: Are we there yet?(e), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 429–440.

[41] H. N. Yasin, S. H. A. Hamid, R. J. R. Yusof, M. Hamzah, An empirical analysis of test input generation tools for Android apps through a sequence of events, Symmetry 12 (11) (2020) 1894.

[42] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, B. Liang, Multiple-entry testing of Android applications by constructing activity launching contexts, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, 2020, pp. 457–468.

[43] S. Carino, J. H. Andrews, Dynamically testing GUIs using ant colony optimization (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 138–148.

[44] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, S. Chandra, Guided test generation for web applications, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 162–171.

[45] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, Z. Wang, Making system user interactive tests repeatable: When and what should we control?, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 55–65.

[46] I. M. Alsmadi, Using mutation to enhance GUI testing coverage, IEEE software 30 (1) (2012) 67–73.

[47] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, A. Usman, Amoga: A static-dynamic model generation strategy for mobile apps testing, IEEE Access 7 (2019) 17158–17173.

[48] L. Duan, A. Hofer, H. Hussmann, Model-based testing of infotainment systems on the basis of a graphical human-machine interface, in: 2010 Second International Conference on Advances in System Testing and Validation Lifecycle, IEEE, 2010, pp. 5–9.

[49] J. Itkonen, M. V. Mäntylä, Are test cases needed? replicated comparison between exploratory and test-case-based software testing, Empirical Software Engineering 19 (2) (2014) 303–342.

[50] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, J. Kazmeier, Automation of GUI testing using a model-driven approach, in: Proceedings of the 2006 international workshop on Automation of software test, 2006, pp. 9–14.

[51] H. Zhang, A. Rountev, Analysis and testing of notifications in Android wear applications, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 347–357.

[52] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, Y. Donmez, Qbe: Qlearning-based exploration of Android applications, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 105–115.

[53] T. Wetzlmaier, R. Ramler, Hybrid monkey testing: enhancing automated GUI tests with random test generation, in: Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing, 2017, pp. 5–10.

[54] R. C. Bryce, S. Sampath, A. M. Memon, Developing a single model and test prioritization strategies for event-driven software, IEEE Transactions on Software Engineering 37 (1) (2010) 48–64.

[55] A. Marchetto, R. Tiella, P. Tonella, N. Alshahwan, M. Harman, Crawlability metrics for automated web testing, International Journal on Software Tools for Technology Transfer 13 (2) (2011) 131–149.

[56] N. Beierle, P. M. Kruse, T. E. Vos, GUI-profiling for performance and coverage analysis, in: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2017, pp. 28–31.

[57] A. Oliveira, R. Freitas, A. Jorge, V. Amorim, N. Moniz, A. C. Paiva, P. J. Azevedo, Sequence mining for automatic generation of software tests from GUI event traces, in: International Conference on Intelligent Data Engineering and Automated Learning, Springer, 2020, pp. 516–523.

[58] A. Miniukovich, A. De Angeli, Computation of interface aesthetics, in: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, 2015, pp. 1163–1172.

[59] Q. Mayo, R. Michaels, R. Bryce, Test suite reduction by combinatorial-based coverage of event sequences, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2014, pp. 128–132.

[60] D. H. Nguyen, P. Strooper, J. G. Süß, Automated functionality testing through GUIs, in: Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102, 2010, pp. 153–162.

[61] H. Reza, S. Endapally, E. Grant, A model-based approach for testing GUI using hierarchical predicate transition nets, in: Fourth International Conference on Information Technology (ITNG'07), IEEE, 2007, pp. 366–370.

[62] Z.-W. He, C.-G. Bai, GUI test case prioritization by state-coverage criterion, in: 2015 IEEE/ACM 10th International Workshop on Automation of Software Test, IEEE, 2015, pp. 18–22.

[63] Y. Gao, C.-G. Bai, Selecting test cases by cluster analysis of GUI states, in: 2016 IEEE Chinese GUIdance, Navigation and Control Conference (CGNCC), IEEE, 2016, pp. 1024–1029.

[64] E. Habibi, S.-H. Mirian-Hosseinabadi, Event-driven web application testing based on model-based mutation testing, Information and Software Technology 67 (2015) 159–179.

[65] F. Gao, L. Zhao, C. Liu, GUI testing techniques based on event interactive graph tree model, in: The 2010 IEEE International Conference on Information and Automation, IEEE, 2010, pp. 823–827.

[66] S. Huang, R. Chen, Y. Jiang, Z. Hui, GUI testing based on function-diagram, in: 2010 International Conference on Computer Application and System Modeling (ICCASM 2010), Vol. 5, IEEE, 2010, pp. V5–353.

[67] T. Kushwaha, O. P. Sangwan, Prediction of usability level of test cases for GUI based application using fuzzy logic, in: Confluence 2013: The Next Generation Information Technology Summit (4th International Conference), IET, 2013, pp. 83–86.

[68] L. Novella, M. Tufo, G. Fiengo, Improving test suites via a novel testing with model learning approach, in: 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), IEEE, 2018, pp. 229–234.

[69] A. Memon, I. Banerjee, N. Hashmi, A. Nagarajan, Dart: a framework for regression testing" nightly/daily builds" of GUI applications, in: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., IEEE, 2003, pp. 410–419.

[70] Q. Xie, A. M. Memon, Studying the characteristics of a" good" GUI test suite, in: 2006 17th International Symposium on Software Reliability Engineering, IEEE, 2006, pp. 159–168.

[71] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su, Guided, stochastic model-based GUI testing of Android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 245–256.

[72] D. Adamo, D. Nurmuradov, S. Piparia, R. Bryce, Combinatorial-based event sequence testing of Android applications, Information and Software Technology 99 (2018) 98–117.

[73] H. Reza, K. Ogaard, A. Malge, A model based testing technique to test web applications using statecharts, in: Fifth International Conference on Information Technology: New Generations (itng 2008), IEEE, 2008, pp. 183–188.

[74] Z. Hui, R. Chen, S. Huang, B. Hu, GUI regression testing based on function-diagram, in: 2010 IEEE International Conference on Intelligent Computing and Intelligent Systems, Vol. 2, IEEE, 2010, pp. 559–563.

[75] A. Beer, S. Mohacsi, C. Stary, Idatg: an open tool for automated testing of interactive software, in: Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241), IEEE, 1998, pp. 470–475.

[76] A. Kervinen, M. Maunumaa, M. Katara, Controlling testing using three-tier model architecture, Electronic Notes in Theoretical Computer Science 164 (4) (2006) 53–66.

[77] A. Rauf, M. Ramzan, Parallel testing and coverage analysis for context-free applications, Cluster Computing 21 (1) (2018) 729–739.

[78] Z. Yu, C. Bai, K.-Y. Cai, Mutation-oriented test data augmentation for GUI software fault localization, Information and Software Technology 55 (12) (2013) 2076–2098.

[79] C.-Y. Huang, J.-R. Chang, Y.-H. Chang, Design and analysis of GUI test-case prioritization using weight-based methods, Journal of Systems and Software 83 (4) (2010) 646–659.

[80] A. Marchetto, P. Tonella, Using search-based algorithms for ajax event sequence generation during testing, Empirical Software Engineering 16 (1) (2011) 103–140.

[81] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon, MobiGUItar: Automated model-based testing of mobile apps, IEEE software 32 (5) (2014) 53–59.

[82] S. Carino, J. H. Andrews, Evaluating the effect of test case length on GUI test suite performance, in: 2015 IEEE/ACM 10th International Workshop on Automation of Software Test, IEEE, 2015, pp. 13–17.

[83] A. M. Memon, M. L. Soffa, M. E. Pollack, Coverage criteria for GUI testing, in: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, 2001, pp. 256–267.

[84] L. Mariani, M. Pezzè, D. Zuddas, Augusto: Exploiting popular functionalities for the generation of semantic GUI tests with oracles, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 280–290.

[85] S. Majeed, M. Ryu, Model-based replay testing for event-driven software, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016, pp. 1527–1533.

[86] R. C. Bryce, S. Sampath, J. B. Pedersen, S. Manchester, Test suite prioritization by cost-based combinatorial interaction coverage, International Journal of System Assurance Engineering and Management 2 (2) (2011) 126–134.

[87] W. Choi, K. Sen, G. Necul, W. Wang, Detreduce: minimizing Android GUI test suites for regression testing, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 445–455.

[88] A. Rauf, S. Anwar, M. A. Jaffer, A. A. Shahid, Automated GUI test coverage analysis using ga, in: 2010 Seventh International Conference on Information Technology: New Generations, IEEE, 2010, pp. 1057–1062.

[89] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, S. Malek, Reducing combinatorics in GUI testing of Android applications, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 559–570.

[90] I. Alsmadi, M. Al-Kabi, The introduction of several user interface structural metrics to make test automation more effective, The Open Software Engineering Journal 3 (1) (2009).

[91] R. Michaels, D. Adamo, R. Bryce, Combinatorial-based event sequences for reduction of Android test suites, in: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2020, pp. 0598–0605.

[92] R. Michaels, M. K. Khan, R. Bryce, Test suite prioritization with element and event sequences for Android applications, in: 2021 IEEE

11th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2021, pp. 1326–1332.

[93] X. Yuan, M. Cohen, A. M. Memon, Covering array sampling of input event sequences for automated GUI testing, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 405–408.

[94] R. C. Bryce, A. M. Memon, Test suite prioritization by interaction coverage, in: Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting, 2007, pp. 1–7.

[95] A. C. Paiva, L. Vilela, Multidimensional test coverage analysis: Paradigm-cov tool, Cluster Computing 20 (1) (2017) 633–649.

[96] J. Yan, T. Wu, J. Yan, J. Zhang, Widget-sensitive and back-stack-aware GUI exploration for testing Android apps, in: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2017, pp. 42–53.

[97] A. M. Adeniyi, A. S. Olalekan, An improved genetic algorithm-based test coverage analysis for graphical user interface software, American Journal of Software Engineering and Applications 5 (2) (2016) 7–14.

[98] A.-J. Molnar, Live visualization of GUI application code coverage with GUItracer, in: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), IEEE, 2015, pp. 185–189.

[99] L. Zhao, K.-Y. Cai, Event handler-based coverage for GUI testing, in: 2010 10th International Conference on Quality Software, IEEE, 2010, pp. 326–331.

[100] S. McMaster, A. Memon, Call-stack coverage for GUI test suite reduction, IEEE Transactions on Software Engineering 34 (1) (2008) 99–115.

[101] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2) (2009) 131–164.

Table A.8: Statistics for Functional-level metrics

Sub-Category	Metric	Total mentions	Unique names	Total definitions	Unique definitions	Name Variability	Undefined mentions	Definition variability
Issues	Number of Failures	2	2	0	0	0%	100%	-
Issues	Number of Faults	24	9	2	1	33.3%	91.7%	0%
Issues	Number of Crashes	6	3	4	2	233.3%	33.3%	25%
Issues	Number of Exceptions	1	1	1	1	0%	0%	0%
Specification	Business rule Coverage	1	1	1	1	0%	0%	0%
Specification	Requirements Coverage	1	1	0	0	0%	100%	-
Specification	Feature Coverage	2	2	1	1	0%	0%	0%
Test Verification	Invariant Detection	1	1	1	1	0%	0%	0%
Test Verification	Mutation Score	3	3	2	2	0%	33.3%	50%
Others	Usage-Oriented Coverage	1	1	1	1	0%	0%	0%
Others	Application-oriented Coverage	1	1	1	1	0%	0%	0%
Others	Perceived Coverage	1	1	1	1	0%	0%	0%
Others	Data Coverage	1	1	1	1	0%	0%	0%
Others	Notification Sites	1	1	1	1	0%	0%	0%

Table A.9: Statistics for GUI-level metrics

Sub-Category	Metric	Total mentions	Unique names	Total definitions	Unique definitions	Name Variability	Undefined mentions	Definition variability
Screen Coverage	Page Coverage	23	8	10	2	30.4%	56.2%	10%
Screen Coverage	PCOV	1	1	1	1	0%	0%	0%
Screen Coverage	Nested Pages	1	1	1	1	0%	0%	0%
Widget Coverage	Simple Widget Coverage	4	4	4	4	75%	0%	75%
Widget Coverage	Parameter-based Widget Coverage	1	1	1	1	0%	0%	0%
Others	Action Coverage	1	1	1	1	0%	0%	0%
Others	Cell Coverage	1	1	1	1	0%	0%	0%

Table A.10: Statistics for Model-level metrics

Sub-Category	Metric	Total mentions	Unique names	Total definitions	Unique definitions	Name Variability	Undefined mentions	Definition variability
GUI State	Single State	14	6	8	4	35.7%	42.9%	37.5%
GUI Event	Single Event	57	25	29	5	43.9%	49.1%	13.8%
GUI Path	Multiple State	2	1	2	1	0%	0%	0%
GUI Path	Multiple Event	31	26	26	5	80.6%	16.1%	15.4%
GUI Path	Prime Path Coverage	2	1	2	1	0%	0%	0%
GUI Path	Tree Paths Number	1	1	0	0	0%	100%	-
GUI Path	Tree depth	2	2	2	1	50%	0%	0%
T-Way	Combinatorial Coverage (CCov)	5	1	3	1	0%	40.0%	0%
T-Way	Sequence-Based Combinatorial Coverage (SCov)	3	1	3	1	0%	0%	0%
T-Way	Consecutive-Sequence Combinatorial Coverage (CSCov)	3	1	3	1	0%	0%	0%
Link-based criteria	Action-one-link coverage	1	1	1	1	0%	0%	0%
Link-based criteria	Action-all-link coverage	1	1	1	1	0%	0%	0%
Link-based-criteria	Action n-way coverage	1	1	1	1	0%	0%	0%
Interaction Criteria	Single event and single outcome	1	1	1	1	0%	0%	0%
Interaction Criteria	Many events and single outcome	1	1	1	1	0%	0%	0%
Interaction Criteria	Event to component	1	1	1	1	0%	0%	0%
Interaction Criteria	Component to event	1	1	1	1	0%	0%	0%
Interaction Criteria	Component-to-component	1	1	1	1	0%	0%	0%
Others	Condition coverage	2	1	2	1	0%	0%	0%
Others	Invocation coverage	1	1	1	1	0%	0%	0%
Others	Invocation-termination coverage	1	1	1	1	0%	0%	0%
Others	Action coverage	1	1	1	1	0%	0%	0%
Others	TGConfs executed	1	1	1	1	0%	0%	0%

Table A.11: Statistics for Code-level metrics

Sub-Category	Metric	Total mentions	Unique names	Total definitions	Unique definitions	Name Variability	Undefined mentions	Definition variability
Unit-based	Class coverage	4	2	1	1	25%	75%	0%
Unit-based	Method coverage	9	3	3	1	22.2%	66.7%	0%
Unit-based	Block coverage	4	2	1	1	25%	75%	0%
Path-based	Path coverage	2	1	1	1	0%	50%	0%
Path-based	Branch coverage	16	2	0	0	6.25%	100%	-
Path-based	Condition coverage	1	1	0	0	0%	100%	-
Line-based	Statement coverage	40	4	3	1	7.5%	92.5%	0%
Line-based	Line coverage	24	3	2	1	8.3%	66.7%	0%
Others	Event-handler coverage	1	1	1	1	0%	0%	0%
Others	Call-stack coverage	1	1	1	1	0%	0%	0%
Others	Event call-graph coverage	1	1	1	1	0%	0%	0%

Appendix A. Statistics about individual metrics

Table A.8 provides the statistics for Functional-level metrics. As an example, the most mentioned metric is the one that we named *Number of Faults*, with 24 mentions. For this metric, we grouped several synonyms of the same concept (e.g., *Fault Revealing Ability*, *Fault Coverage*) or the noun fault (e.g. *Defects*, *Bugs*), resulting in a name variability of 33.3%. The metric was explicitly defined in only two sources (percentage of undefined mentions of 91.7%) and the definition was the same in both cases (0% definition variability).

Table A.9 provides the statistics for GUI-level metrics. The most mentioned metric is the Page Coverage metric, with 23 mentions. With this code, we identified a set of 8 different metrics in the papers. This code had an important name variability and definition variability especially because of different names the screens are called based on the application domain (e.g., *Activities* for Android applications, *Windows* for desktop applications). Few mentions at the GUI Level were not related to page coverage, with the second most-mentioned metric (*Simple Widget Coverage*) being used in just four different papers.

Table A.10 provides the statistics for Model-level metrics. In this category, we identify a clear tendency to utilize base metrics for the evaluation of model coverage (14 mentions for *Single Coverage*, 57 for *Single Event Coverage*) and very high variability of the names and the specific definitions in the manuscripts. This variability is mainly due to the fact that the manuscript

utilizes different structures to model the GUIs of the tested SUTs, which results in different definitions and mathematical formulations of very similar concepts. We found many mentions (with, again, very high variability in names) for metrics that aim to analyze the exploration of paths in the GUI model through the evaluation of sequences of two or more events (*Multiple Event Coverage*). Many metrics (e.g., all in the categories *Link-based criteria* and *Interaction Criteria*) were defined in a single manuscript and did not share commonalities with other works in the literature.

Table A.11 provides the statistics for Code-level metrics. Code-level metrics were widely used in the sample of literature items that we analyzed, with a predominant utilization of Line-based coverage metrics (40 mentions for *Statement Coverage*, 24 mentions for *Line Coverage*). Metrics of this category have a very high percentage of mentions where no definition is provided (with a top 100% for *Branch Coverage* and *Condition Coverage*), but at the same time no variability in the definitions, mainly because they can build upon decades of utilization in any software testing domain.