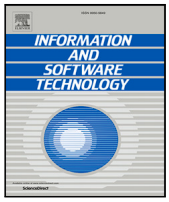
Information and Software Technology 138 (2021) 106625

Contents lists available at ScienceDirect

Information and Software Technology

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



# Why many challenges with GUI test automation (will) remain

Michel Nass [a,*], Emil Alégroth [b], Robert Feldt [b,c]

[a] H. SERL, Blekinge Institute of Technology, Sweden
[b] P. SERL, Blekinge Institute of Technology, Sweden
[c] Chalmers University of Technology, Sweden

ARTICLE INFO

*Keywords:*
System testing
GUI testing
Test automation
Systematic literature review

ABSTRACT

**Context:** Automated testing is ubiquitous in modern software development and used to verify requirement conformance on all levels of system abstraction, including the system's graphical user interface (GUI). GUI-based test automation, like other automation, aims to reduce the cost and time for testing compared to alternative, manual approaches. Automation has been successful in reducing costs for other forms of testing (like unit- or integration testing) in industrial practice. However, we have not yet seen the same convincing results for automated GUI-based testing, which has instead been associated with multiple technical challenges. Furthermore, the software industry has struggled with some of these challenges for more than a decade with what seems like only limited progress.

**Objective:** This systematic literature review takes a longitudinal perspective on GUI test automation challenges by identifying them and then investigating why the field has been unable to mitigate them for so many years.

**Method:** The review is based on a final set of 49 publications, all reporting empirical evidence from practice or industrial studies. Statements from the publications are synthesized, based on a thematic coding, into 24 challenges related to GUI test automation.

**Results:** The most reported challenges were mapped chronologically and further analyzed to determine how they and their proposed solutions have evolved over time. This chronological mapping of reported challenges shows that four of them have existed for almost two decades.

**Conclusion:** Based on the analysis, we discuss why the key challenges with GUI-based test automation are still present and why some will likely remain in the future. For others, we discuss possible ways of how the challenges can be addressed. Further research should focus on finding solutions to the identified technical challenges with GUI-based test automation that can be resolved or mitigated. However, in parallel, we also need to acknowledge and try to overcome non-technical challenges.

## 1. Introduction

Manual testing is time-consuming, repetitive, and error-prone to perform for a human tester [1,2]. Test automation, using techniques like unit testing [3] and record–replay [4], have been suggested as solutions to these challenges and proven successful in practice. Another reason for test automation is that the tests can be executed faster and more frequently and, therefore, deliver faster feedback about the quality of the software under development [5]. While automated testing is the state of practice for checking code and APIs (using unit- and integration testing), it is still common to manually test a system under test (SUT) through the graphical user interface (GUI) without any automation. That manual testing, using the GUI, is preferred before test automation indicates that there are challenges related to GUI-based test automation that reduces its applicability compared to manual GUI-based testing. This observation is supported by literature, which has frequently reported on challenges, problems, and limitations with GUI test automation.

This systematic literature review (SLR) [6] aims to take a longitudinal perspective to find the technical challenges reported by academic literature with GUI-based test automation during the last 20-years to determine how the most reported (key) challenges have evolved and if there have been attempts to mitigate them. In this SLR, the focus is mainly on testing the functionality of the SUT by using the GUI, not testing or checking that the user interface is presented, or rendered, correctly on the screen.

There have been previous attempts that try to identify the challenges (technical or otherwise) of software test automation. For instance, in 2012, Rafi et al. presented an SLR, which identifies 9 benefits and 7 limitations of automated software testing [7]. The literature

* Corresponding author.
*E-mail addresses:* michel.nass@bth.se (M. Nass), emil.alegroth@bth.se (E. Alégroth), robert.feldt@chalmers.se (R. Feldt).

https://doi.org/10.1016/j.infsof.2021.106625
Received 14 July 2020; Received in revised form 9 May 2021; Accepted 11 May 2021
Available online 14 May 2021

**Table 1**
Overview showing the number of included publications per step.

| Step | Description | No included |
|---|---|---|
| 4 | Search Relevant Publications | 185 |
| 5 | Screen of Relevant Publications | 95 |
| 8 | Include/Exclude Publications | 39 |
| 9 | Snowballing Selected Publications | 49 |

review identified several benefits: "Less human effort" and "Increased fault detection". However, "Automation cannot replace manual testing" and "False expectations" were identified as some of the limitations. A more recent SLR study by Wiklund et al. published in 2017, reported impediments, both technical and non-technical, provided similar results on software test automation [8].

In this SLR, unlike the previous that studied the entire field of test automation, the focus is on test automation performed using the GUI only. Hence, a subset of the software test automation field. Additionally, the SLR is delimited to the technical challenges of the approach that can be addressed using technical solutions, leaving out any non-technical perspectives such as behavioral, process, or business aspects. Furthermore, the SLRs published by Rafi et al. and Wiklund et al. present limitations and impediments on a relatively high abstraction level, an effect of their studies looking at such a large field of study. As an example, Rafi et al. reported the limitation: "Difficulty in maintenance of test automation" and Wiklund et al. the impediments: "Maintenance Costs" and "Fragile Test Scripts". In contrast, the goal of this SLR is to dig deeper into the technical challenges to identify their root causes to acquire further insights into their solution. Finally, the previous SLRs do not present solution attempts, how the challenges have evolved, and if the challenges are possible to solve or mitigate, thereby motivating the need for this SLR that investigates the complexities and longevity of each unresolved challenge in more detail. Knowing how consistently the challenges were described over a more extended time period, and any solution attempts, will make it possible to identify solutions or mitigation strategies for the key (most commonly reported) technical challenges.

The specific contributions of this literature review are:

- The technical challenges of test automation through the GUI reported by academic literature.
- A longitudinal perspective of how the key challenges have evolved.
- To what extent the key challenges can be resolved or mitigated using a technical solution.

This paper is structured as follows: The research process behind the systematic literature review is described in Section 2. Section 3.1 contains the results obtained by gathering information from the included publications. Section 4 discusses the results presented in Section 3.1 and synthesized in Section 3.2. We will give our conclusions and some work to consider for the future in Section 6.

## 2. Systematic literature review

The SLR presented in this work is based on the guidelines for conducting systematic literature reviews in software engineering, presented by Kitchenham et al. [6]. An overview of the literature review process is shown in Fig. 1. The number of included publications after each step is presented in Table 1.

**1. Establish Research Goal:** The research goal of this SLR is to find the key challenges, based on empirical evidence found in literature, associated with testing a system automatically through its GUI. Knowing the key challenges and their root-causes is crucial for future research to find effective solutions to mitigate them efficiently.

**2. Define Research Questions:** The research goal has been broken down into the following research questions:

**Table 2**
Search results.

| Scopus | IEEE Xplore | ACM | Wiley |
|---|---|---|---|
| 69 matches | 58 matches | 35 matches | 23 matches |

- RQ1: What are the academically reported technical challenges for automated testing of an application through its GUI?
- RQ2: How have key challenges for GUI-based test automation evolved in the last 20 years?
- RQ3: To what extent can the key challenges be resolved or mitigated using a technical solution?

**3. Identify Search Strings:** The search string sought to find publications that contain empirical results about the challenges of automated testing/checking of software applications/programs through the GUI. Keywords related to benefits were included in the search since a discussion about benefits is likely to contain statements concerning challenges as well.

The query string was constructed from keywords and synonyms to these keywords, which were formulated based on the authors' industrial experience and taking keywords used in previous literature reviews into consideration [7,8]. Below, we present a decomposition of the search string.

**Focus on software applications or programs:** (software OR application OR program) AND

**Focus on automated testing/checking:** ("test automation" OR "automatic testing" OR "automated testing" OR "automated test" OR "automatic checking" OR "automated checking" OR "automated regression") AND

**Focus on user-interfaces or human–machine interfaces:** ("user interface" OR GUI OR "human–machine interface" OR HMI) AND

**Contains benefits and/or challenges:** (challenge OR benefit OR advantage OR improvement OR limitation OR drawback OR problem OR pitfall) AND

**Contains empirical studies or experiments:** (empirical OR industry OR industrial OR practice OR "case study" OR survey OR experiment)

**Resulting in the final query string:** (software OR application OR program) AND ("test automation" OR "automatic testing" OR "automated testing" OR "automated test" OR "automatic checking" OR "automated checking" OR "automated regression") AND ("user interface" OR GUI OR "human–machine interface" OR HMI) AND (challenge OR benefit OR advantage OR improvement OR limitation OR drawback OR problem OR pitfall) AND (empirical OR industry OR industrial OR practice OR "case study" OR survey OR experiment)

**Exclude publication released before year 2000:** Due to the rapid evolution of the software engineering field, we decided to exclude all publications published before the year 2000. This choice was made to delimit the number of found papers and because papers older than this are considered out-of-date and not applicable to modern software systems.

**4. Search Relevant Publications:** Four of the most widely used literature databases available today for the software engineering field were selected for the search: Scopus, IEEE Xplore, ACM, and Wiley. The final search string and the filter that excludes all publications published before the year 2000 were used in the selected databases. The database searches resulted in a total of 185 included publications distributed chronologically, as shown in Table 2.

**5. Screen of Relevant Publications:** All publications included in the search results were downloaded and imported into the tool Mendeley [9]. Mendeley was used to remove any duplicate publications found in more than one database. Non-research publications, publications that were not peer-reviewed, or not written in English were also excluded resulting in 95 remaining publications.

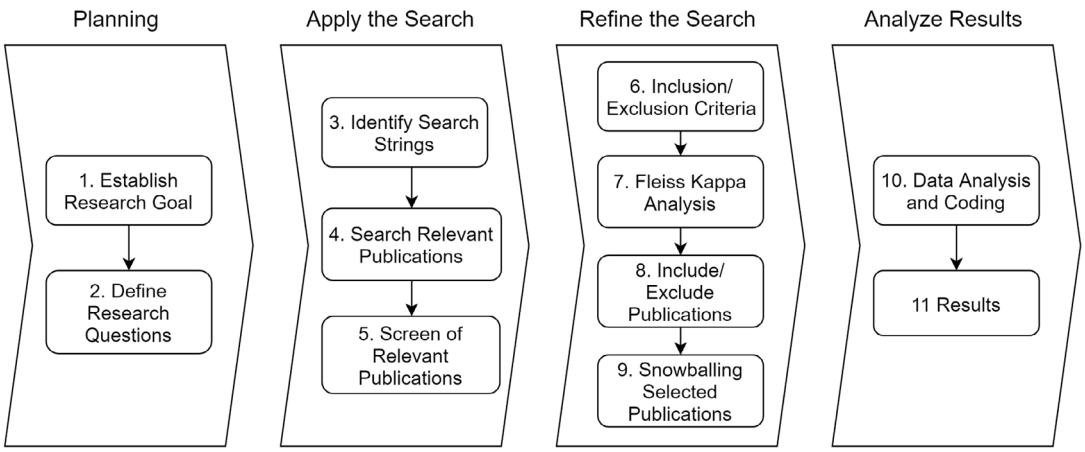



**Fig. 1.** Literature review process.

**Table 3**
Definitions.

| Term: | Definition: |
|---|---|
| Peer-reviewed article | Peer-reviewed articles are written by experts and are reviewed by several other experts in the field before the article is published to ensure the article's quality. |
| Empirical evidence | Any result acquired or observed by a human (researcher) in a reported experiment or case study (regardless of size). |
| Graphical User Interface (GUI) | A form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. |
| System Under Test (SUT) | The system that is being tested for correct operation. |
| Document Object Model (DOM) | A cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document. |
| Widget | A component of a GUI, that enables a user to perform a function or access a service. |
| Automated testing | Automated testing or test automation is a method in software testing that makes use of special software tools to control the execution of tests and then compares actual test results with predicted or expected results. Test execution is done automatically with little or no intervention from the test engineer. For this study, automated testing regards only tests of the functionality or features of the SUT, not its quality characteristics (like usability or performance) nor the visual appearance (that widgets are rendered correctly) on the SUT's user interface. |
| Automated testing through the GUI | Automated testing of a system through emulation of human/user usage of the SUT. The emulation can be performed using physical robotics (e.g. a robot arm) or purely through software (e.g. using OS-level events to type on the keyboard or move the mouse cursor), or a combination of these. Access to widgets is given by the accessibility API of the operating system, DOM objects in the SUT, internal GUI controls or other technical interfaces. These are all considered valid methods for emulating a human/user. |

**6. Inclusion/Exclusion Criteria:** We formulated three tiers of inclusion/exclusion criteria as shown in Tables 4–6. Publications that satisfied all of the inclusion criteria and did not match any of the exclusion criteria for a tier were included and evaluated by the criteria in the next tier until all the tiers were covered. Table 3 contains definitions for some of the terms that we used in the inclusion/exclusion criteria.

The three tiers were defined as:

- **Tier 0 - Check Preconditions:** Analysis of paper metadata to ensure that the paper followed the base criteria to be eligible for further analysis.
- **Tier 1 - Check Title and Abstract:** Analysis of the title and abstract of each paper to get an indication that it would include support evidence usable to meet the research objective.
- **Tier 2 - Read Full Text:** Analysis of the complete paper to ensure it would provide evidence to meet the research objective.

**7. Fleiss Kappa Analysis:** The leading researcher evaluated the remaining 95 publications using the inclusion/exclusion criteria for tier 0 and randomly selected 20 (more than 20 percent) of the included publications for Fleiss Kappa analysis [10]. The authors of this publication reviewed the randomly selected publications based on the inclusion/exclusion criteria for Tier 1 and Tier 2, marking papers that they believed should be included, excluding the rest. Fleiss Kappa analysis, a measure of the agreement between reviewers (inter-rater agreement), was then calculated. The result was 0.60, indicating a moderate or close to a substantial agreement between the reviewers [11]. This agreement among the three reviewers was considered good enough for the leading researcher to continue to review the remaining publications and gave us confidence that the inclusion/exclusion criteria were suitably defined. Of course, a threat remains that a few papers are overlooked, but we argue that the impact of a few incorrectly excluded or included publications would not likely have a significant impact on the results.

**Table 4**
Inclusion/Exclusion criteria tier 0.

| Inclusion criteria: | Exclusion criteria: |
|---|---|
| Included publications must be peer-reviewed. See the definition of peer-reviewed article in Table 3. | Exclude all publications published before year 2000 since they are considered out-of-date and not applicable to modern software systems. |
| Include workshop, conferences, book chapters, and journal publications. | Exclude short publications (less than 6 pages). |
| Include publications written in the English language only. | Exclude gray literature (non-peer-reviewed books, blogs, etc.) |
| | Exclude bachelor/master/Ph.D. theses. |

**Table 5**
Inclusion/Exclusion criteria Tier 1.

| Inclusion criteria: | Exclusion criteria: |
|---|---|
| Include publications that indicate that they report empirical evidence about benefits or challenges of automated testing/checking through the GUI. | Exclude publications that do not study automated testing/checking through the GUI. |
| | Exclude publications that do not cover benefits or challenges of automated testing/checking through the GUI. |
| | Exclude publications that do not contain any empirical evidence. |
| | Exclude publications where the empirical evidence does not report benefits or challenges of automated testing/checking through the GUI. |

**Table 6**
Inclusion/Exclusion criteria Tier 2.

| Inclusion criteria: | Exclusion criteria: |
|---|---|
| Include publications that contain empirical evidence about the benefits or challenges of automated testing/checking through the GUI. | Exclude publications that do not study automated testing/checking through the GUI. |
| | Exclude publications that do not cover benefits or challenges of automated testing/checking through the GUI. |
| | Exclude publications that do not contain any empirical evidence. |
| | Exclude publications where the empirical evidence does not report benefits and/or challenges of automated testing/checking through the GUI. |

**8. Include/Exclude Publications:** All publications that were not randomly selected and reviewed during the Fleiss Kappa analysis were reviewed, by the leading researcher, using the inclusion/exclusion criteria for tier 1 and 2. The review resulted in 39 included publications, counting also the publications there were included during the Fleiss Kappa analysis.

**9. Snowballing Selected Publications:** There are many possible reasons for not finding a relevant publication during the database searches. One is that the author of the publication uses a different vocabulary than the one used in the search string. Another reason is that the search keywords are not found since most literature databases only search a portion of the publication, like the title and abstract, but not the full text.

To verify that all relevant papers had been captured by the search, backward and forward snowballing, as described by Wohlin [12], was performed on the 39 included publications. Publications identified during the snowballing session that seemed relevant for the study, based on reading their title or abstract, were collected into a set of candidates. The leading researcher then assessed all the candidates using the inclusion/exclusion criteria to ensure that we analyzed all added or discarded candidates with the same rigor and systematic review as previously included publications. Using the existing inclusion/exclusion criteria is perceived to strengthen the approach's reliability since these criteria had been previously agreed upon by the researchers. This snowballing session resulted in five additional publications that passed the inclusion/exclusion criteria. Additionally, since performing an SLR is a lengthy process, another five publications were later discovered that had been published while writing this paper, resulting in a final total of 49 included publications.

**10. Data Analysis and Coding:** Analysis of the included publications was performed by the leading researcher using thematic analysis and coding [13]. 186 statements related to technical challenges about GUI-based test automation were identified and extracted from the 49 included publications. A statement is defined as a sentence, quote, or paragraph from the author of the publication based on experience, a conviction, or concluded from a result. All statements are extracted from publications that report empirical results making it more likely that the statements are relevant and valid. The codes (C1…Cn) were formulated iterative, i.e., without any starting set, from the statements, focusing on challenges. Each extracted statement was mapped to an existing code when the leading researcher believed that the statement related to the same challenge or mapped to a new code otherwise. After coding, the codes were analyzed for semantic equivalence and merged if such was found. The merger was done by formulating a new code that replaced both existing codes, or by removing one of the existing codes.

The resulting set of codes and statements were then used as evidence to draw the study's conclusions and answer its research questions.

## 3. Results and synthesis

Here we present an overview of the results and answer the first research question (RQ1) in Section 3.1. The answers to the remaining research questions (RQ2 and RQ3) are presented in Section 3.2 after further analysis and synthesis of the reported challenges.

### 3.1. Results

Table 7 contains a list of the 49 included publications that contain one or more statements related to challenges associated with GUI-based test automation. The coded and mapped challenges from the reviewed literature are listed in Table 8 with references to this paper's reference list. This table thereby represents a summary of all the currently known challenges explicit to automated testing through an application's GUI, representing an answer to RQ1.

To provide an overview, Fig. 2 visualizes the key (most commonly reported) challenges on a timeline sorted on the year that the publication, which reports the challenge, was published. The criteria for a challenge to be considered "key" is that four or more publications can triangulate it. Each dot in the timeline represents one, or more, publications that were published during that year. An index number specifies the number of publications when more than one. The figure shows that several of the challenges have existed for almost 20 years but that the majority of challenges are newer, i.e., published within the last seven years.

To gain further insights, Fig. 3 also visualizes what application platform (desktop, web, or mobile), the empirical evaluation was performed on and how these platforms appear over time. As shown in the figure, the first period, from the year 2000 to 2004, only contains two empirical evaluations, both on desktop applications. In the second and third periods (2005–2015), research on desktop applications is dominant, while in the fourth period, web-applications become dominant, and research in mobile applications are on the rise. We also note that the number of published papers increase over time, indicating a growing interest in empirical studies of GUI-based test automation.

### 3.2. Synthesis

To answer RQ2, the SLR results were analyzed further and synthesized to identify how consistent the challenges remained (e.g., how consistently they were described) over the studied time period. Four of the challenges were reported repeatedly during 17 years or more, as can be seen in Fig. 2, and one (C5) was only reported during a four-year-long time-period.

**Table 7**
Included publications.

| No: | Title: | Year: |
|---|---|---|
| 1 | Hierarchical GUI Test Case Generation Using Automated Planning [14] | 2001 |
| 2 | Effective Automated Testing: A Solution of Graphical Object Verification [15] | 2002 |
| 3 | Experimental Assessment of Manual Versus Tool-Based Maintenance of GUI-Directed Test Scripts [16] | 2009 |
| 4 | Automated GUI Testing for J2ME Software Based on FSM [17] | 2009 |
| 5 | Repairing GUI Test Suites Using a Genetic Algorithm [18] | 2010 |
| 6 | Debug Support for Model-Based GUI Testing [19] | 2010 |
| 7 | Experiences of System-Level Model-Based GUI Testing of an Android Application [20] | 2011 |
| 8 | AutoBlackTest: Automatic Black-Box Testing of Interactive Applications [21] | 2012 |
| 9 | Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study [22] | 2013 |
| 10 | Graphical User Interface Testing Using Evolutionary Algorithms [23] | 2013 |
| 11 | Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing [24] | 2014 |
| 12 | On the Industrial Applicability of TextTest: An Empirical Case Study [25] | 2016 |
| 13 | An analysis of automated tests for mobile Android applications [26] | 2016 |
| 14 | Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice [27] | 2018 |
| 15 | Introducing automated GUI testing and observing its benefits: an industrial case study in the context of law-practice management software [28] | 2018 |
| 16 | Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles [29] | 2018 |
| 17 | Using a Pilot Study to Derive a GUI Model for Automated Testing [30] | 2008 |
| 18 | A "More Intelligent" Test Case Generation Approach through Task Models Manipulation [31] | 2017 |
| 19 | Evaluating the TESTAR tool in an industrial case study [32] | 2014 |
| 20 | Comparing Automated Visual GUI Testing Tools: An Industrial Case Study [33] | 2017 |
| 21 | Robust Test Automation Using Contextual Clues [34] | 2014 |
| 22 | Obstacles and opportunities in deploying model-based GUI testing of mobile software: a survey [35] | 2012 |
| 23 | PESTO: Automated migration of DOM-based Web tests towards the visual approach [36] | 2018 |
| 24 | Design and industrial evaluation of a tool supporting semi-automated website testing [5] | 2014 |
| 25 | Automatic testing of GUI-based applications [37] | 2014 |
| 26 | An event-flow model of GUI-based applications for testing [38] | 2007 |
| 27 | Pattern-based GUI testing: Bridging the gap between design and quality assurance [39] | 2017 |
| 28 | Model-based Approach to Assist Test Case Creation, Execution, and Maintenance for Test Automation [40] | 2011 |
| 29 | Efficient and Change-Resilient Test Automation: An Industrial Case Study [41] | 2013 |
| 30 | Reuse of model-based tests in mobile apps [42] | 2017 |
| 31 | Automated Testing of Software-as-a-Service Configurations using a Variability Language [43] | 2015 |
| 32 | Reducing GUI Test Suites via Program Slicing [44] | 2014 |
| 33 | Improved GUI Testing using Task Parallel Library [45] | 2016 |
| 34 | Introducing Model-Based Testing in an Industrial Scrum Project [46] | 2012 |
| 35 | Development and Maintenance Efforts Testing Graphical User Interfaces: A Comparison [47] | 2016 |
| 36 | Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case? [48] | 2016 |
| 37 | Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation [49] | 2011 |
| 38 | Using combinatorial testing to build navigation graphs for dynamic web applications [50] | 2016 |
| 39 | Automating Web Application Testing from the Ground Up: Experiences and Lessons Learned in an Industrial Setting [51] | 2016 |
| 40 | Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning [52] | 2019 |
| 41 | Using Multi-Locators to Increase the Robustness of Web Test Cases [53] | 2015 |
| 42 | Visual GUI Testing in Practice: limitations, Problems and Limitations [54] | 2015 |
| 43 | Call Stack Coverage for GUI Test-Suite Reduction [55] | 2006 |
| 44 | A Black-Box Based Script Repair Method for GUI Regression Test [56] | 2018 |
| 45 | A New Algorithm for Repairing Web-Locators using Optimization Techniques [57] | 2018 |
| 46 | Apply computer vision in GUI automation for industrial applications [58] | 2019 |
| 47 | ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing [59] | 2016 |
| 48 | Offline Oracles for Accessibility Evaluation with the TESTAR Tool [60] | 2019 |
| 49 | Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study [61] | 2015 |

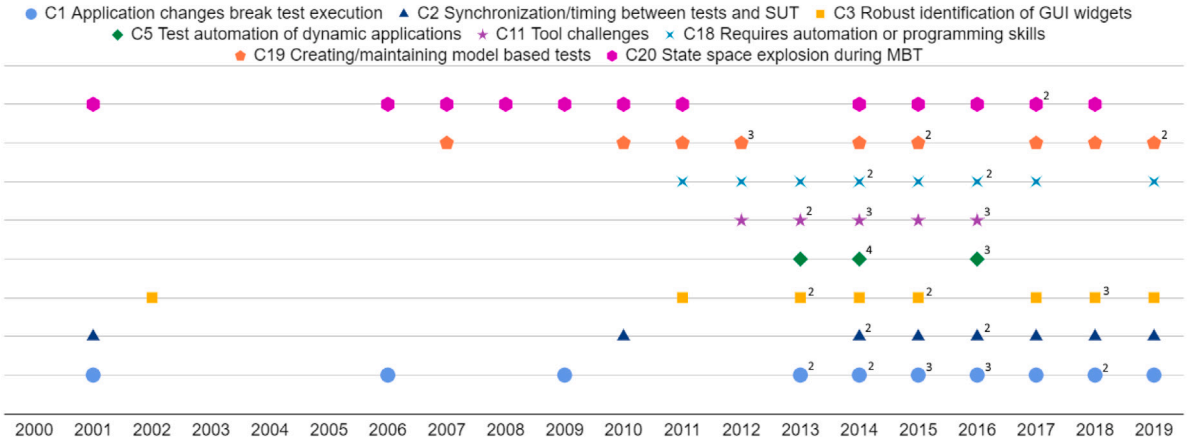



**Fig. 2.** The key challenges mapped on a timeline. Each dot represents a statement from one or more publications published during the year.

The analysis also showed that the level of description of the challenges varied and that some reported challenges were so generally defined that it is unlikely that their description can aid in finding any candidate solutions. Such challenges must first be broken down,

**Table 8**
Reported challenges related to GUI-based test automation.

| Challenge: | Reported by publication no: |
|---|---|
| C1 Application changes break execution | 1, 3, 10, 12, 14, 21, 24, 27, 29, 31, 35, 39, 41, 43, 44, 46, 49 |
| C2 Synchronizing/timing between tests and SUT | 1, 6, 19, 20, 32, 35, 39, 40, 42, 45 |
| C3 Robust identification of GUI widgets | 2, 7, 9, 20, 23, 24, 29, 42, 44, 45, 46, 49 |
| C4 Changed screen resolution | 13, 29, 40 |
| C5 Test automation of dynamic applications | 11, 13, 19, 21, 29, 32, 38, 39 |
| C6 Fails for unknown reason | 14, 24 |
| C7 Non-determinism | 40, 49 |
| C8 Sensitivity to business logic | 20 |
| C9 Cascading error | 29 |
| C10 Environment/Hardware configuration | 13, 21, 29 |
| C11 Tool challenges | 9, 11, 12, 13, 19, 22, 24, 29, 36, 42 |
| C12 High maintenance cost of test cases | 10, 24 |
| C13 Long setup time | 22, 24 |
| C14 HMI must be available | 10, 34 |
| C15 Hard to reproduce the error | 6, 19, 24 |
| C16 Less effective in detecting faults | 15 |
| C17 Requires workflow changes | 22 |
| C18 Requires automation or programming skills | 7, 9, 12, 19, 22, 24, 30, 31, 39, 46 |
| C19 Creating/maintaining model based tests | 6, 8, 11, 16, 22, 26, 27, 28, 31, 34, 40, 48, 49 |
| C20 State space explosion during MBT | 1, 4, 5, 16, 17, 18, 25, 26, 27, 32, 37, 38, 43, 49 |
| C21 Low test coverage during MBT | 19, 36 |
| C22 Identify less faults when using an oracle | 19 |
| C23 Limited applicability of models | 26, 49 |
| C24 Slow test execution during MBT | 19, 30 |

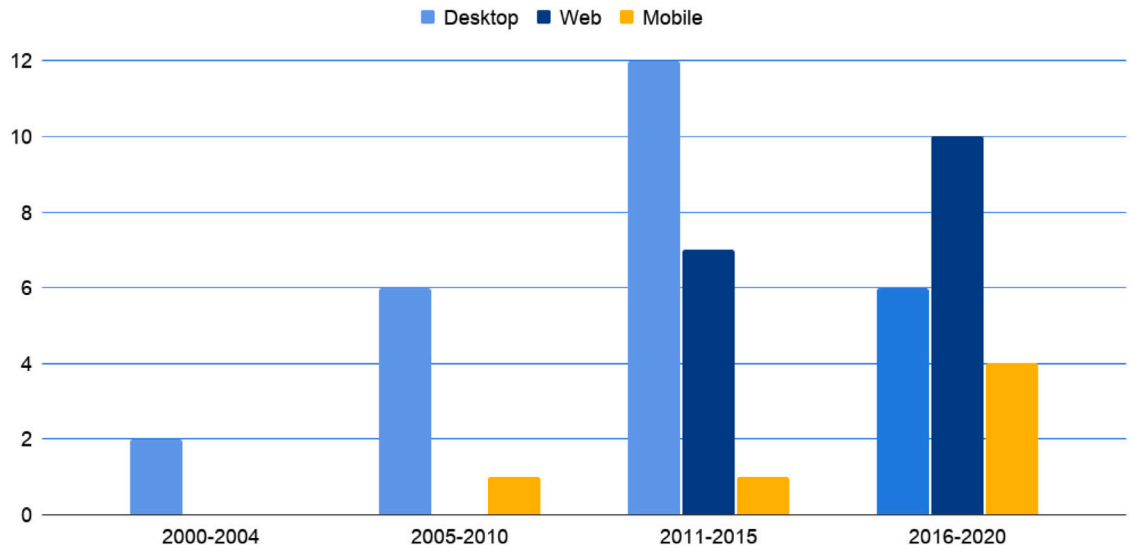



**Fig. 3.** The type of software application selected for the empirical evaluation distributed over four time-periods.

i.e., further detailed, to be adequately understood. As an example, Mahmud et al. reported in 2014, that practitioners with test automation experience mentioned that automated tests had to be rewritten or re-recorded when the application changed. Mahmud et al. did not document what explicit changes the practitioners perceived to be the underlying reason for tests to be rewritten or re-recorded or if all changes had the same impact. Such information would be valuable in guiding both development and research efforts towards finding candidate solutions to aid the practitioners.

To answer RQ3 and better understand which challenges to address with more focus and effort and which challenges to defer from or focus less effort on, we need to distinguish different challenges. In the book "No silver bullet" by Brooks et al. software engineering challenges are classified as essential or accidental [62]. We argue that a similar analysis can add value here. Essential challenges are inherent in the nature of software, whilst accidental challenges are challenges today but not inherent. This classification implies that accidental challenges can be resolved or mitigated through technical, or other, means. In contrast, essential challenges are not necessarily solvable because they are so fundamental, or significant, that no general solution can be perceived. In the following descriptions of found challenges, we use Brooks et al. definitions to classify the identified key challenges in an attempt to provide guidance for future research. However, we decided to limit the definition of accidental challenges to challenges that can be mitigated using a technical solution to match the scope of this SLR. Therefore, our definition of an accidental challenge is that it can be solved using a technical solution. Also, our definition of an essential challenge is that any technological solution cannot completely resolve it. Hence, excluding challenges related to human, process, or organizational aspects of GUI-based test automation.

**Application changes break test execution (C1):** The most reported challenge for automated GUI-based tests is that application changes cause tests to fail, stated by seventeen publications between 2001 to 2018. Memon et al. wrote in 2001 that regression testing of GUI applications is a challenge since the user interface does not remain constant across successive versions of the software [14]. A similar conclusion came from Jiang et al. in 2018, that scripts made for the previous version of the SUT do no longer work well when the GUI layouts change [56].

This challenge is mainly essential since a significant change of the SUT will, and should, break the test execution. Hence, a prerequisite of any technical solution that seeks to eliminates this challenge must fail the test when the test execution is expected to break, but otherwise pass. In particular, this challenge relates to the oracle problem, i.e. what information, and how much information, to use to determine correct SUT behavior [60].

As an essential challenge, we can conclude that this challenge is present today and likely will not be entirely removed in the future. However, when discussing this challenge, we must also consider the background to why the SUT was changed. Some changes to the SUT are intentional, and the tests might need to be adjusted accordingly to prevent the tests from failing during execution. However, other changes are unintentionally caused by intentional modifications to the SUT and might break the test execution since the tests are not adjusted according to the unintentional changes. Furthermore, different changes range in ease of detection. Significant changes, like a missing widget, are simple for a human tester to spot, while minor changes, like the size or position of GUI widgets, might be impossible for a human to notice. Unintentional minor changes that cause the test execution to fail can be frustrating for the human tester since the automated test scripts seem to break for no apparent reason. Since the human tester would expect an intentional change to impact the tests, further research should focus on finding solutions that reduce failures caused by unintentional, and primarily minor, changes. For example, a technical solution that provides a more robust identification of GUI widgets (C3) might prevent minor changes to the SUT from breaking the test execution.

**Synchronizing/timing between tests and SUT (C2):** The challenge of keeping the automated test scripts, or states in a model, in sync with the tested application was reported by ten publications published between 2001 and 2019. This challenge stems from the need for tests to wait until the SUT is ready to receive the next action, or event, to avoid the risk that the action does not have the desired effect on the SUT. As an example, let us study the following pseudo test script:

```
Step 1: Open the Report Dialog.
Step 2: Click on the Create Report button in the Report Dialog.
Step 3: Check that the Report has been created.
```

Since it may take some time to open the Report Dialog (step 1), there is a risk that step 2 is performed before the Report Dialog and Create Report button are visible. The Report might not be created if step 2 is performed before the Create Report button is ready to receive the click action, causing step 3 to fail. A human, replicating the failure, would not perceive the same challenge, and we observe that the script, in this case, would have reported a "false positive" test result. However, and regardless, the test result still required investigation, leading to unnecessary root-cause analysis costs. The solution to making the test more robust would be to add synchronization, like static or dynamic delays, between the steps.

However, as stated by Heiskanen et al. in 2010, a common cause of failure in test runs are the delays between executing certain keywords on the SUT [19]. This statement is supported by Cheng et al. [58], in 2019, which stated that "The timing to trigger a series of events may not always be the same between two different runs. So, a straightforward replay is often infeasible".

The synchronization challenge has been reported repeatedly, from empirical evaluations on all types of applications (desktop, web, and mobile), during an 18-year time-period and is likely still a challenge today, supported by two publications published as recently as 2019. Since human testers do not seem to experience this challenge when testing manually, but machines and software often face issues, it is perceivable that this is a purely accidental, and technical, challenge. Debroy et al. [51] suggest that static (or fixed) delays, e.g., a delay of a few seconds, will solve this challenge but at the cost of making the script execution unnecessarily slow. For instance, assume that the transition between GUI states is 1.9 s. A delay of 2 s would then be sufficient to synchronize the test execution and ensure that the next script action would happen after the GUI state change, with only a penalty of 100-millisecond unnecessary delay. Of course, these penalties should be kept as small as possible, but (1) fine-tuning synchronization is time-consuming [25], and (2) having minor penalties increases the risk that performance degradation in the application leads to new false positives. For instance, in our example above that, the state transition time increases from 1.9 to 2.1 s. Debroy et al. therefore instead suggests the use of dynamic waits, e.g., waiting for a specific page title, or until the title changes from a particular title. Another option is to wait for GUI widgets to appear, for example, a headline, a text field, or a button.

The synchronization challenge is related to, or partly dependent on, the challenge of robust identification of GUI widgets (C3) since the test execution needs to be synchronized by waiting for correctly identified widgets. Therefore, we pose that these two challenges (C2 and C3) are connected and that a solution to C3 might also provide a possible solution, or mitigation, for the synchronization challenges (C2).

**Robust identification of GUI widgets (C3):** As stated, this challenge is possibly related, or even the leading cause of many of the other reported challenges. It might be one of the challenges that cause the test execution to fail when the SUT changes (C1). A more robust widget detection would, per definition, be more tolerant and possibly able to prevent unexpected failure. We differentiate between expected and unexpected failures here since some changes to the application are expected to cause failure while other changes are not. For example, a missing widget is expected to fail the test execution, but a minor change in a widget's position should not cause the script to fail. Robust widget identification of this type would possibly also mitigate the synchronization challenge (C2) since dynamic waits rely on waiting for located widgets. The challenges related to a changed screen resolution (C4) might also be reduced since a reliable widget identification technique might be able to find the correct widgets even if the widgets have been rearranged. Thummalapenta et al. [41] give the example that some elements (widgets) might not appear in the same location when the screen resolution changes, thus preventing the test execution from continuing. Even tests that fail due to unknown reasons (C6) might be partly caused by non-robust identification of widgets since failure due to timing and widget identification can be hard to understand and explain. Hence, challenges reported in research may be incorrectly classified even though caused by this underlying challenge.

The correct widget, like an input field or a button, first needs to be identified before verifying some of its properties or triggering an action. Failing to identify the correct widget will most likely result in a failed script. The correct widget is, in this case, defined as the widget that the human tester that created the script or model would select in a given situation. Note that the automated test execution should fail when the correct widget is not present since that would have also prevented the human tester from continuing. Widgets can, however, be located in several ways. Alégroth et al. [63] describe three generations of techniques for automated GUI-based testing: coordinate-based, component/widget-based, and Visual GUI Testing (image recognition). These techniques are fundamentally different and, as such, have various benefits and drawbacks. Research has shown that combinations of these techniques can have positive net effects, which represent one possible solution to the problem [61].

Twelve publications reported challenges with reliable identification of widgets from 2002 to 2019. Two of the papers bring up the challenge of using screen coordinates. Four publications mention the challenge with component/widget-based identification, seven of the publications talk about robustness issues related to image recognition, and one does not go into any technical details. Solving or significantly reducing the challenges with robust identification of widgets could have a notable impact on the success of the GUI-based test automation field. There have been many attempts to solve or mitigate this challenge over

the years. Leotta et al. proposed, in 2015, a new type of locator, named multi-locator, that uses a voting procedure for selecting the best candidate from a set of locators [53]. Multi-locators might be one solution for reducing some of the fragility issues when trying to locate widgets when testing an application through the GUI. Another exciting technique, proposed by Yandrapally et al. for addressing the fragility challenge by improving the reliability of widget localization, is using contextual clues [34]. The idea is to mimic how humans perceive an application by evaluating the target widget and the surrounding widgets. Taking advantage of the surrounding widgets would, in theory, make it possible to locate a widget even in the worst case when all of the locators, associated with a widget, failed by using contextual information about the widget's neighbors. To automatically compensate for frequent changes and be able to identify widgets in a continually changing GUI would perhaps increase the stability of the automated tests and lower the maintenance cost of automated tests. Therefore, we perceive this challenge to be accidental since it should be possible to come up with a technical solution that can identify widgets with an accuracy comparable to human testers. We base this assumption on recent advancements in oracle and GUI locator research, where significant improvements to test execution robustness can be observed [53].

**Test automation of dynamic applications (C5):** Controlling the dynamic aspects of an application is not only a challenge for GUI-based test automation but rather a challenge shared by the entire field of test automation and all types of applications. Dynamic aspects include, continuously changing databases, dynamic GUI contents, application state transitions, and dependency of external dynamic applications. The challenge is mentioned by eight publications during a four-year-long time-period between 2013 and 2016. Thummalapenta et al. [41] give an example: "if test setup does not remove such entities created during the previous test run, scripts in the new run fail while attempting to create an existing entity". Hence, an additional complexity with testing dynamic applications is that the tests themselves may also be affecting the system state when tests are executed. This implies that tests might need to be executed in a specific order since the state change of previous states set the preconditions for later tests.

Automating a dynamic application is an essential challenge compared to a static application, placing significantly higher demands on the time and resources needed to handle and prepare tests to deal with dynamic data in a robust, deterministic way. For instance, the application database might need to be set to a specific state before starting the test execution, external systems might need to be mocked, and test cases might need to be executed in a particular order to lay the foundation for stable test automation. While some of these issues might be mitigated through technological advances, the core challenge of non-deterministic SUT behavior will remain, making this a lasting challenge.

**Tool challenges (C11):** Tool related challenges were reported, for all platforms, by ten publications published between 2012 and 2016. These challenges are however different from each other, ranging from difficult installation [5,32] or lacking documentation [54] to technical issues like the one mentioned by Alégroth et al. in 2015 [25]: "The main drawbacks are that the tool requires hooks into the AUT's GUI library which means TextTest is currently restricted to Java and Python.".

These challenges range from minor drawbacks to severe maturity or stability issues with the software products. While challenges with unique tools might be accidental and possible to solve, the higher-level concept of "tool challenges", in general, is an essential challenge since it is difficult, or even impossible, to find one technical solution that addresses them all. For example, general solutions how to make the user interface more user-friendly or how to make the application more responsive.

**Requires automation or programming skills (C18):** Successful test automation requires knowledge, experience and skills to master. Between 2011 and 2019, ten publications mentioned the lack of skills as a challenge in regards to either test automation, test automation tools, or programming when automating an application through the GUI. The need for development skills were further explicitly specified by eight of the included papers. The need for programming skills were highlighted by five papers [5,32,42,43,51] and two papers stated a need for experience in writing regular expressions [25] or XPath locators [59].

Minor skills in test automation or programming can be sufficient when only creating a few test cases but, as with the case of software development, more skills are needed when creating and maintaining an extensive suite of tests as complexity increases with size. Many tools have targeted this challenge during the years because skilled resources are typically hard to find and are more expensive to hire. Capture/replay (CR), or record/replay, tools is one approach that attempts to simplify and speed up the process of creating and running automated GUI-based test cases. However, as explained by Moreira et al. the main drawback with CR tools is the cost of script maintenance [39]. Previously recorded tests break easily when the GUI changes and this leads to considerable manual labor to repair them. According to Moreira et al. this is often the reason the CR methods are abandoned after a few software releases. Another challenge emerges with CR when newly recorded test cases need to be merged with existing test suites since this generally requires programming and skills to deal with growing code complexity. The lack of skill is mostly an accidental challenge. It might be possible to find a technical solution that makes CR tools capable of creating and maintaining stable test suites that can be used by testers with less programming skills than today. One possible solution could be to create and maintain the tests using an Augmented GUI proposed by Nass et al. [64]. Lack of skills could also be seen as an essential challenge since test automation skills will always be required regardless of any technical advances. Hence, even if user-friendly tools can be developed, the purpose of the tools, and their contextual use, must still be taught to the users.

**Creating/maintaining model based tests (C19):** The time and cost to create and keep the model up to date with a continually changing GUI application resemble the challenge of maintaining the test scripts to prevent application changes from breaking the test execution (C1). This challenge is probably the main reason why the industry has not yet embraced MBT. Memon et al. wrote, in 2007, that models used for automated GUI testing are expensive to create [38] and was confirmed by Aho et al. [24], in 2014, Patel et al. [43], in 2015, and Moira et al. [39], in 2017. That model-based tests also require maintenance effort, to keep the models up to date when the SUT changes, is also mentioned by three publications [19,35,46], from 2010 to 2012.

One option for avoiding the time-consuming work of manually creating the models is to generate the models by traversing the SUT automatically. While this approach makes it possible to generate thousands of test cases quickly, they might not be effective or efficient in finding defects or providing adequate coverage of the SUT. The dilemma is especially valid for GUI-testing since it takes many magnitudes more time to execute tests on a GUI-level than on a unit-level. Gupta et al. claimed, in 2011, that model-based approaches mainly focus on test case generation but that it is not effective in real-life industry scenarios [40]. Aho et al. provide supporting statements to Gupta, in 2014, that industry adoption of model-based approaches for GUI-based test automation has been insignificant despite many academic approaches to target the challenge [24].

The challenge reported on all application platforms, associated with the creation/maintenance of model-based tests (C19), is accidental since automated ways of generating models, like GUI crawling, lower model development cost. It should be possible to find new or improve existing technical solutions that target this challenge through further research. Instead, especially for GUI-based test automation, the underlying challenge is that the industry cannot afford to maintain and run

several test technologies at the same time, e.g., script-based and model-based tests. Thus, although the technologies have different benefits and drawbacks, script-based testing is considered less costly and therefore comes up on top. As such, there is still a need to find solutions for efficient modeling of GUI applications to make MBT attractive for the industry.

**State space explosion during MBT (C20):** There are close to infinite paths or scenarios through a typical GUI application. This presents a core challenge for model-based testing in terms of possible state-space to explore during testing (also known as state-space explosion). Memon et al. [14] reported in 2001 that the space of possible interactions with a GUI is high, and twelve other publications have provided supporting observations of the challenge for both desktop and web applications from 2006 to 2018.

As such, the state-space explosion challenge is essential. Any technical mitigation attempt to it must be able to navigate the state-space in such a manner that it only finds scenarios that within the state-space that, for instance, provide the best test coverage of the SUT or the best defect finding ability. However, these solutions are workarounds to the problem since they still do not cover the entire state-space, which, in theory, could be infinite and, therefore, never completely covered. Mariani et al. proposed a machine-learning based solution, called Augusto, that can efficiently and effectively test application-independent functionalities (AIFs) while the rest of the SUT needs to be covered using existing approaches [29]. Examples of AIFs are common user interface patterns, like authentication operations and CRUD (Create, Read, Update, Delete) operations. While it is still too early to know if the approach, proposed by Mariani et al. works for the industry, it implies that machines can learn from human behavior how to efficiently and effectively navigate through GUI applications and thus avoid the essential state space explosion challenge. However, as the challenge is considered essential, we perceive that even advanced solutions will only avoid the challenge, never solve it entirely.

## 4. Discussion

While it might be tempting to find a solution to a challenge, we should first determine if the challenge is possible to solve or if the challenge is inherent. Fig. 4 contains the key challenges identified in our study related to GUI-based test automation, arranged from essential to accidental difficulties. Challenges are considered as essential if they are inherent to a specific technology or approach and as accidental when it is possible to eliminate the challenge through technology or improved ways of working [62]. However, as discussed in the synthesis of the results, the view of the challenge as accidental or essential also relates to from what perspective we view the challenge. Therefore, some challenges are arranged in Fig. 4 in between essential and accidental, for example, tool challenges.

In the figure, we also attempt to cluster the challenges into three different groups related to (1) test execution fragility, (2) tools and automation skills, and (3) MBT challenges. This grouping is based on the semantic similarities between the challenges of each group, as presented in the literature, e.g., in what papers the challenges are presented. In the following sections, we discuss these groups in more detail.

**Test execution fragility:** Three of the challenges have been grouped together as they are related to the fragility of test execution. As discussed, they may be all caused by the challenge to reliably identify widgets when running automated tests, i.e. (C3). There have been, at least, nine academic papers that attempt to solve or mitigate the challenge with robust widget identification in the past two decades [34,41,53,57,59,65–68], and publications still report the same challenge. Despite these efforts, the challenges remain, possibly because the academic solutions have been insufficient in solving the challenges faced by the industry. Another possible explanation is that academia has found possible solutions but not communicated these in a suitable way to industry. In the latter case, academia might have failed in transferring the knowledge to the industry, or the industry has not yet been able to incorporate the solution into their products or processes.

**Tools and automation skills:** Three of the challenges form a group related to test automation tools or skills required for efficient/effective test automation or programming (C5, C11, and C18). The challenge that testers need automation and programming skills to succeed with GUI-based testing (C18) is the only challenge, in this group, that we defined as mainly accidental and, therefore, the challenge that should be devoted to the most research efforts in the future. The current solution to the challenge is training the team members or by hiring someone with the required skills. However, people with in-depth knowledge of test automation and programming are difficult to find, and training is both time-consuming and costly. Therefore, the industry would benefit from a GUI-based test automation approach that does not require the same amount of skills as needed today.

CR tools was a good attempt for such an approach since it allowed anyone with domain knowledge to record valid test scripts. However, the challenge with CR tools is that the recorded scripts are fragile and need programming skills to be enhanced or merged into reliable test suites. Also, CR tools can only record new test scripts and require programming or scripting knowledge when maintaining existing test scripts at a high cost. This first challenge, related to the script fragility, can be partly mitigated with a more robust identification of GUI widgets (C3). The remaining challenges could be addressed by a possible future solution for reducing the need for automation and programming skills. It could enhance CR tools with functionality, methods, or processes that can aid the manual tester when merging and maintaining the recorded tests.

Aho et al. divide GUI-based test automation into three levels: Script-based, Model-based, and Scriptless [69]. According to Aho et al. the benefits of scriptless testing are that the initial investment and maintenance effort is low compared to the other approaches. Still, a challenge is effective action selection since actions are typically selected randomly. Combining scriptless with the recording functionality from CR tools would perhaps be an interesting approach to investigate in further research since recorded actions from real test scenarios could provide useful hints when selecting actions instead of picking actions randomly. Hence, a more machine-learning based approach based on models of the system to be tested, recorded from human users.

**MBT challenges:** The final group is about challenges with model-based testing. Since the state-space explosion challenge (C20) is essential, research should concentrate on possible solutions for creating and maintaining model-based tests (C19). Many of the reported MBT challenges are caused by the large number of possible ways to traverse a GUI application and to be able to extract the paths that provide the best test coverage and the highest chance of finding a defect. Instead of trying to extract high-quality test scenarios by randomly traversing the application, the solution might instead be similar to the one suggested by Mariani et al. that we need high-level guidance from typical patterns of working with GUI applications [29]. Perhaps it is even possible to create the patterns, or the models themselves, by recording or observing actual end-users or testers of the SUT. Recording the models from the users would be a feasible solution to handle both the state space challenge and the high cost of creating the models. Theoretically, these models would include meta information of what the primary states of the application under test are, assuming that they are covered by the human that recorded them. This information would provide guidance for heuristics of how to generate suitable test cases, which could trump existing random approaches.

The challenge that models need maintenance when the SUT changes (C19) are very similar to the challenge that application changes break the test execution (C1), which in turn depends a lot on robust identification of GUI widgets (C3). Finding a solution to C3 might, consequently, have a positive impact on C19.

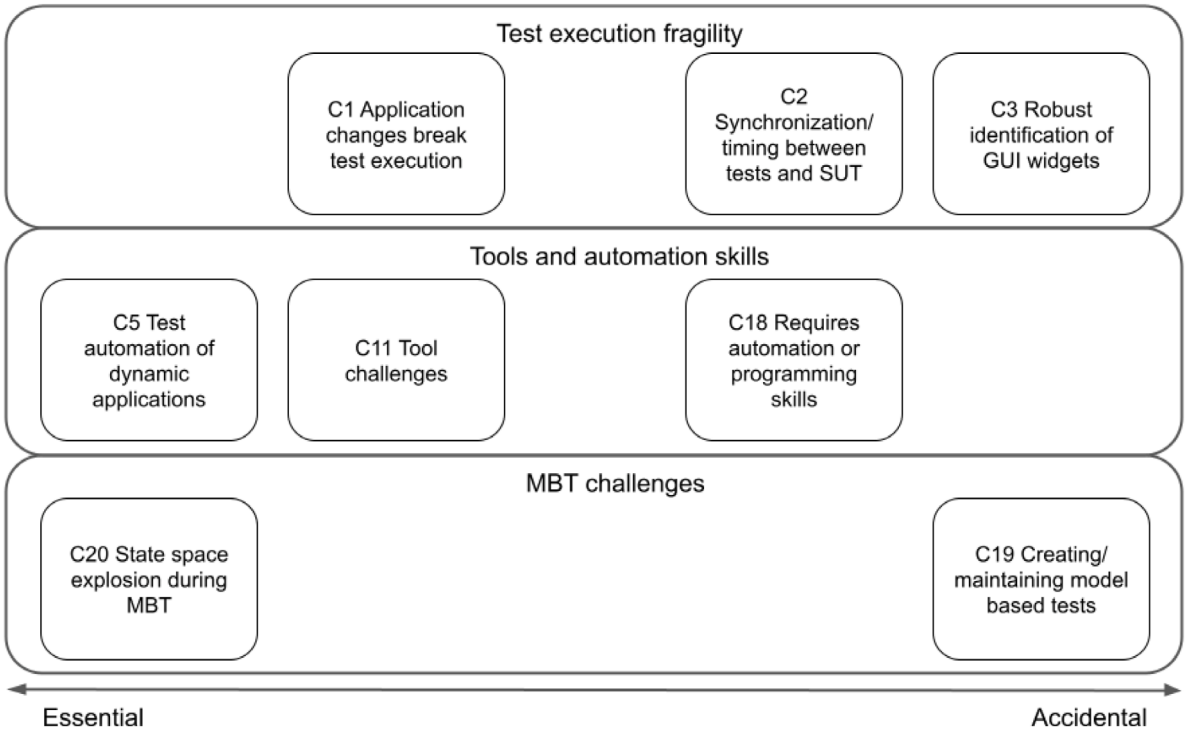



**Fig. 4.** The key challenges related to GUI-based test automation arranged from essential to accidental difficulties.

**Summary:** Synthesis of the acquired results, and grouping of the identified challenges, reveal some challenges to be accidental (solvable) and some essential (only possible to mitigate or solvable in specific contexts). Additionally, many challenges share the common issue of a lack of robust identification of GUI widgets, implying that this is a central and critical challenge to solve.

Furthermore, the grouping of essential and accidental challenges helps us explain why certain challenges are chronologically reoccurring, and cannot be solved, in research on different test techniques and platforms. This conclusion provides interesting insights for both industry and academia. It should influence our mindset regarding what challenges we target and how we reason about the proposed solution- or mitigation strategies in the future.

## 5. Threats to validity

This section covers some of the threats to the validity of this literature review.

**Construct validity:** The selection of databases and construction of the query string for the literature search has a significant impact on the publications included in this study and is a threat to be considered. We divided the query string into several parts and tested each part separately to avoid missing any relevant publications that contained benefits or challenges about GUI-based test automation. A search tool was created to perform full-text searches inside the downloaded publications to make sure that they contained all the keyword combinations. This was complemented by manual review to ensure the inclusion of semantically suitable papers. However, four publications were included during the snowballing procedure after checking the inclusion/exclusion criteria and suggests that our search query string was not perfect. However, we decided that it was good enough to continue with the study since we argue that a few missed publications will likely not have a significant impact on the results. We argue that in the worst case, we may have missed some challenge of importance, but given the authors' extensive empirical understanding of the area, we perceive it as unlikely. However, it cannot be ruled out since the challenges were identified and formulated by the leading researcher based on the statements extracted from the included publications and is therefore affected by the leading researcher's knowledge and perception. However, the acquired challenges were reviewed by the other authors, and no explicit challenges have been observed as left out.

**Internal validity:** The extensive industrial experience held by the authors of this study is a threat to internal validity. Bias from industrial experience can undoubtedly affect the synthesis and coding of challenges, even though we were aware of this challenge and actively tried to make unbiased judgments. There is also a risk that the challenges extracted from the literature are biased or that our interpretation of them are affected by our own experience from the field and that it might have an impact on the synthesis.

**External validity:** We decided to exclude publications that did not contain any empirical evidence to try to reduce the positive-results bias, a type of publication bias that occurs when the authors are more likely to submit positive results than negative or inconclusive results [70].

Also, a well-defined methodology, like a systematic literature review in our case, makes it less likely that the literature results are biased even though it does not protect against publication bias in the primary studies, according to Kitchenham et al. [6]. To eliminate possible bias from our own analysis, the results and conclusions were verified within the research group through continuous discussion throughout the study process. Although not eliminating analysis bias, this approach mitigates the personal bias of one or several of the authors and thereby improves the validity of the results.

**Reliability:** Data analysis and coding of challenges are affected by the author's experience and knowledge in the GUI-based test automation field. They are a threat to the reliability and reproducibility of this literature review. Repeating this literature review would likely result in slightly different descriptions of the challenges, but we consider these potential deviations small and not significantly affect our results. Even though our definitions of essential and accidental challenges were based on the definition by Brooks et al. we decided to limit the definition of accidental challenges as something that can be mitigated using a technical solution to get a clear definition and match the scope of this SLR. Our redefinition and interpretation of essential and accidental challenges impact the classification of the reported challenges.

However, we note that it does not affect essential challenges as these, per definition, cannot be completely solved with technical solutions, or otherwise.

## 6. Conclusions

Test automation through the GUI is still a challenge despite many emerging tools and technologies during the last two decades. This SLR pinpoints 24 challenges of GUI-based test automation synthesized from 49 publications that build on empirical evidence. Eight key challenges were mapped on a timeline to determine how the challenges have evolved and if there have been attempts to mitigate them.

A novel contribution of this work is that all identified key challenges have been classified as essential or accidental in an attempt to provide guidance for future research and to avoid time expenditure on solutions for inherent challenges that cannot be solved entirely. Instead, we urge a focus on the challenges of a more accidental nature that we can solve by further research. Additionally, we urge a change in mindset, with this new understanding in mind, to approach proposed solutions or mitigation strategies in the future.

We might never find any solutions to four of the key challenges since they are essential. The remaining four are accidental challenges that are possible to mitigate and therefore deserving further attention. The challenge of robust identification of GUI widgets (C3) could be the root cause of the synchronization challenge (C2) and many of the other challenges reported. A solution, to C3, could make the test execution less fragile and reduce the maintenance time and cost of both test scripts and models. Both the challenge that GUI-based test automation requires automation or programming skills (C18) and the challenge of creating/maintaining model-based tests (C19) could be targeted by a tool, method, or process that provide scriptless creation and maintenance of the tests instead of recording and maintaining the tests by coding or scripting like a conventional CR tool. A scriptless approach could provide a more efficient and effective way of creating and maintaining automated GUI tests without extensive programming skills.

Future work should be conducted to find solutions that enhance script execution robustness (C2 and C3), reduces the skills required for successful GUI-based test automation (C18), and improves the efficiency and effectiveness of creating/maintaining model-based tests (C19).

## CRediT authorship contribution statement

**Michel Nass:** Leading researcher, Wrote the paper, Conceived and designed the analysis, Collected the data, Performed the analysis. **Emil Alégroth:** Assisted in writing and reviewing the paper, Assisted in designing the analysis, Assisted in performing the analysis. **Robert Feldt:** Assisted in writing and reviewing the paper, Assisted in designing the analysis, Assisted in performing the analysis.

## Declaration of competing interest

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

## Acknowledgment

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

## References

[1] M. Grechanik, Q. Xie, C. Fu, Maintaining and evolving GUI-directed test scripts, in: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 408–418.
[2] M. Grechanik, Q. Xie, C. Fu, Creating GUI testing tools using accessibility technologies, in: Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on, IEEE, 2009, pp. 243–250.
[3] M. Olan, Unit testing: test early, test often, J. Comput. Sci. Colleges 19 (2) (2003) 319–328.
[4] A. Adamoli, D. Zaparanuks, M. Jovic, M. Hauswirth, Automated GUI performance testing, Softw. Qual. J. 19 (4) (2011) 801–839.
[5] J. Mahmud, A. Cypher, E. Haber, T. Lau, Design and industrial evaluation of a tool supporting semi-automated website testing, Softw. Test. Verif. Reliab. 24 (1) (2014) 61–82.
[6] B. Kitchenham, R. Pretorius, D. Budgen, O.P. Brereton, M. Turner, M. Niazi, S. Linkman, Systematic literature reviews in software engineering–a tertiary study, Inform. Softw. Technol. 52 (8) (2010) 792–805.
[7] D.M. Rafi, K.R.K. Moses, K. Petersen, M.V. Mäntylä, Benefits and limitations of automated software testing: Systematic literature review and practitioner survey, in: Proceedings of the 7th International Workshop on Automation of Software Test, IEEE Press, 2012, pp. 36–42.
[8] K. Wiklund, S. Eldh, D. Sundmark, K. Lundqvist, Impediments for software test automation: A systematic literature review, Softw. Test. Verif. Reliab. 27 (8) (2017) e1639.
[9] H. Zaugg, R.E. West, I. Tateishi, D.L. Randall, Mendeley: Creating communities of scholarly inquiry through research collaboration, TechTrends 55 (1) (2011) 32–36.
[10] J.L. Fleiss, J. Cohen, The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability, Educ. Psychol. Meas. 33 (3) (1973) 613–619.
[11] J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data, Biometrics (1977) 159–174.
[12] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, Citeseer, 2014, p. 38.
[13] D.S. Cruzes, T. Dyba, Recommended steps for thematic synthesis in software engineering, in: 2011 International Symposium on Empirical Software Engineering and Measurement, IEEE, 2011, pp. 275–284.
[14] A.M. Memon, M.E. Pollack, M.L. Soffa, Hierarchical GUI test case generation using automated planning, IEEE Trans. Softw. Eng. 27 (2) (2001) 144–155.
[15] J. Takahashi, Y. Kakuda, Effective automated testing: a solution of graphical object verification, in: Proceedings of the 11th Asian Test Symposium, 2002.(ATS'02)., IEEE, 2002, pp. 284–291.
[16] M. Grechanik, Q. Xie, C. Fu, Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 9–18.
[17] Y. Hou, R. Chen, Z. Du, Automated GUI testing for J2me software based on FSM, in: 2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing, IEEE, 2009, pp. 341–346.
[18] 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.
[19] H. Heiskanen, A. Jääskeläinen, M. Katara, Debug support for model-based GUI testing, in: 2010 Third International Conference on Software Testing, Verification and Validation, IEEE, 2010, pp. 25–34.
[20] T. Takala, M. Katara, J. Harty, Experiences of system-level model-based GUI testing of an android application, in: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, IEEE, 2011, pp. 377–386.
[21] L. Mariani, M. Pezze, O. Riganelli, M. Santoro, Autoblacktest: Automatic black-box testing of interactive applications, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, 2012, pp. 81–90.
[22] E. Alegroth, R. Feldt, H.H. Olsson, Transitioning manual system test suites to automated testing: An industrial case study, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, IEEE, 2013, pp. 56–65.
[23] G.I. Laţiu, O. Creţ, L. Văcariu, Graphical user interface testing using evolutionary algorithms, in: 2013 8th Iberian Conference on Information Systems and Technologies (CISTI), IEEE, 2013, pp. 1–6.
[24] P. Aho, M. Suarez, T. Kanstrén, A.M. Memon, Murphy tools: Utilizing extracted gui models for industrial software testing, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2014, pp. 343–348.
[25] E. Alegroth, G. Bache, E. Bache, On the industrial applicability of texttest: An empirical case study, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–10.

[26] D.B. Silva, A.T. Endo, M.M. Eler, V.H. Durelli, An analysis of automated tests for mobile android applications, in: 2016 XLII Latin American Computing Conference (CLEI), IEEE, 2016, pp. 1–9.

[27] E. Alégroth, A. Karlsson, A. Radway, Continuous integration and visual GUI testing: Benefits and drawbacks in industrial practice, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 172–181.

[28] V. Garousi, E. Yildirim, Introducing automated GUI testing and observing its benefits: an industrial case study in the context of law-practice management software, in: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2018, pp. 138–145.

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

[30] Q. Xie, A.M. Memon, Using a pilot study to derive a GUI model for automated testing, ACM Trans. Softw. Eng. Methodol. (TOSEM) 18 (2) (2008) 7.

[31] J.C. Campos, C. Fayollas, M. Gonçalves, C. Martinie, D. Navarre, P. Palanque, M. Pinto, A more intelligent test case generation approach through task models manipulation, Proc. ACM Human-Comput. Interact. 1 (EICS) (2017) 9.

[32] S. Bauersfeld, T.E. Vos, N. Condori-Fernandez, A. Bagnato, E. Brosse, Evaluating the TESTAR tool in an industrial case study, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2014, p. 4.

[33] V. Garousi, W. Afzal, A. Çağlar, İ.B. Işık, B. Baydan, S. Çaylak, A.Z. Boyraz, B. Yolaçan, K. Herkiloğlu, Comparing automated visual GUI testing tools: an industrial case study, in: Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing, ACM, 2017, pp. 21–28.

[34] R. Yandrapally, S. Thummalapenta, S. Sinha, S. Chandra, Robust test automation using contextual clues, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, 2014, pp. 304–314.

[35] M. Janicki, M. Katara, T. Pääkkönen, Obstacles and opportunities in deploying model-based GUI testing of mobile software: a survey, Softw. Test. Verif. Reliab. 22 (5) (2012) 313–341.

[36] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Pesto: Automated migration of DOM-based web tests towards the visual approach, Softw. Test. Verif. Reliab. 28 (4) (2018) e1665.

[37] L. Mariani, M. Pezzè, O. Riganelli, M. Santoro, Automatic testing of GUI-based applications, Softw. Test. Verif. Reliab. 24 (5) (2014) 341–366.

[38] A.M. Memon, An event-flow model of GUI-based applications for testing, Softw. Test. Verif. Reliab. 17 (3) (2007) 137–157.

[39] R.M. Moreira, A.C. Paiva, M. Nabuco, A. Memon, Pattern-based GUI testing: Bridging the gap between design and quality assurance, Softw. Test. Verif. Reliab. 27 (3) (2017) e1629.

[40] P. Gupta, P. Surve, Model based approach to assist test case creation, execution, and maintenance for test automation, in: Proceedings of the First International Workshop on End-To-End Test Script Engineering, ACM, 2011, pp. 1–7.

[41] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D.D. Nagaraj, S. Kumar, S. Kumar, Efficient and change-resilient test automation: An industrial case study, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 1002–1011.

[42] G. de Cleva Farto, A.T. Endo, Reuse of model-based tests in mobile apps, in: Proceedings of the 31st Brazilian Symposium on Software Engineering, ACM, 2017, pp. 184–193.

[43] S. Patel, V. Shah, Automated testing of software-as-a-service configurations using a variability language, in: Proceedings of the 19th International Conference on Software Product Line, ACM, 2015, pp. 253–262.

[44] S. Arlt, A. Podelski, M. Wehrle, Reducing GUI test suites via program slicing, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, 2014, pp. 270–281.

[45] M.M. Öztürk, A. Zengin, Improved GUI testing using task parallel library, ACM SIGSOFT Softw. Eng. Notes 41 (1) (2016) 1–8.

[46] V. Entin, M. Winder, B. Zhang, S. Christmann, Introducing model-based testing in an industrial scrum project, in: Proceedings of the 7th International Workshop on Automation of Software Test, IEEE Press, 2012, pp. 43–49.

[47] A. Kresse, P.M. Kruse, Development and maintenance efforts testing graphical user interfaces: a comparison, in: Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, ACM, 2016, pp. 52–58.

[48] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, T. Xie, Automated test input generation for android: Are we really there yet in an industrial case?, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 987–992.

[49] S. Arlt, C. Bertolini, M. Schäf, Behind the scenes: an approach to incorporate context in GUI test case generation, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2011, pp. 222–231.

[50] W. Wang, S. Sampath, Y. Lei, R. Kacker, R. Kuhn, J. Lawrence, Using combinatorial testing to build navigation graphs for dynamic web applications, Softw. Test. Verif. Reliab. 26 (4) (2016) 318–346.

[51] V. Debroy, L. Brimble, M. Yost, A. Erry, Automating web application testing from the ground up: Experiences and lessons learned in an industrial setting, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 354–362.

[52] D. Amalfitano, V. Riccio, N. Amatucci, V. De Simone, A.R. Fasolino, Combining automated GUI exploration of android apps with capture and replay through machine learning, Inf. Softw. Technol. 105 (2019) 95–116.

[53] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Using multi-locators to increase the robustness of web test cases, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–10.

[54] E. Alégroth, R. Feldt, L. Ryrholm, Visual gui testing in practice: challenges, problems and limitations, Empir. Softw. Eng. 20 (3) (2015) 694–744.

[55] S. McMaster, A. Memon, Call-stack coverage for GUI test suite reduction, IEEE Trans. Softw. Eng. 34 (1) (2008) 99–115.

[56] W. Jiang, X. Li, X. Wang, A black-box based script repair method for GUI regression test, in: 2018 7th International Conference on Digital Home (ICDH), IEEE, 2018, pp. 148–153.

[57] H.M. Eladawy, A.E. Mohamed, S.A. Salem, A new algorithm for repairing web-locators using optimization techniques, in: 2018 13th International Conference on Computer Engineering and Systems (ICCES), IEEE, 2018, pp. 327–331.

[58] Y.-P. Cheng, C.-W. Li, Y.-C. Chen, Apply computer vision in GUI automation for industrial applications, Math. Biosci. Eng. MBE 16 (6) (2019) 7526–7545.

[59] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Robula+: An algorithm for generating robust xpath locators for web testing, J. Softw. Evol. Process 28 (3) (2016) 177–204.

[60] F. de Gier, D. Kager, S. de Gouw, E.T. Vos, Offline oracles for accessibility evaluation with the TESTAR tool, in: 2019 13th International Conference on Research Challenges in Information Science (RCIS), IEEE, 2019, pp. 1–12.

[61] 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.

[62] F. Brooks, H. Kugler, No Silver Bullet, 1987, April.

[63] E. Alégroth, Visual Gui Testing: Automating High-Level Software Testing in Industrial Practice, Chalmers University of Technology, 2015.

[64] M. Nass, E. Alégroth, R. Feldt, Augmented testing: Industry feedback to shape a new testing technology, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2019, pp. 176–183.

[65] I. Aldalur, O. Diaz, Addressing web locator fragility: a case for browser extensions, in: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2017, pp. 45–50.

[66] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, X. Li, ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2017, pp. 161–171.

[67] S.R. Choudhary, D. Zhao, H. Versee, A. Orso, Water: Web application test repair, in: Proceedings of the First International Workshop on End-To-End Test Script Engineering, 2011, pp. 24–29.

[68] Y. Zheng, S. Huang, Z.-w. Hui, Y.-N. Wu, A method of optimizing multi-locators based on machine learning, in: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2018, pp. 172–174.

[69] P. Aho, T. Vos, Challenges in automated testing through graphical user interface, in: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2018, pp. 118–121.

[70] D.L. Sackett, Bias in analytic research, in: The Case-Control Study Consensus and Controversy, Elsevier, 1979, pp. 51–63.