A Literature Survey of Assertions in Software Testing

Masoumeh Taromirad[1,2] [0000−0002−0838−928X] and
Per Runeson[1] [0000−0003−2795−4851]

[1] Lund University, SE-221 00 Lund, Sweden
[2] Jönköping University, SE-551 11 Jönköping, Sweden
{masoumeh.taromirad,per.runeson}@cs.lth.se

Abstract. Assertions are one of the most useful automated techniques for checking program's behaviour and hence have been used for different verification and validation tasks. We provide an overview of the last two decades of research involving 'assertions' in software testing. Based on a term–based search, we filtered the inclusion of relevant papers and synthesised them w.r.t. the problem addressed, the solution designed, and the evaluation conducted. The survey rendered 119 papers on assertions in software testing. After test oracle, the dominant problem focus is test generation, followed by engineering aspects of assertions. Solutions are typically embedded in tool prototypes and evaluated throughout limited number of cases while using large–scale industrial settings is still a noticeable method. We conclude that assertions would be worth more attention in future research, particularly regarding the new and emerging demands (e.g., verification of programs with uncertainty), for effective, applicable, and domain-specific solutions.

Keywords: assertions, testing, literature survey

1 Introduction

While there is abundance of research regarding the selection of test inputs and execution conditions, the assessment of expected results is less covered. Research on the expected results of test cases is often framed as "the oracle problem", with Weyuker as an early contributor, observing 1982 that "[a]lthough much of the testing literature describes methodologies which are predicated on both the theoretical and practical availability of an oracle, in many cases such an oracle is pragmatically unattainable" [82].

Barr et al. [6] surveyed the research literature related to oracles and classified oracles into specified, derived, implicit, and no automatable ones. Among the concepts identified in their survey are 'assertions', defined as "a boolean expression that is placed at a certain point in a program to check its behaviour at runtime". Despite being dated back to Turing and integrated into programming languages, testing tools and practices of today, they only found a few pieces of work specifically focused on assertions [16]. As our current research develops

around assertions, we decided to survey the existence of assertions, for testing purposes, in more recent research.

Our research goal is to provide an overview of existing research literature on assertions in software testing, to provide a basis for further research. As our research "aims to improve an area of practice", we choose the design science paradigm as a lens for this literature survey, as proposed by Engström et al. [25]. We search for literature that uses assertions or addresses problems with assertions in software testing. In line with design science elements, we catalogue the *problems* addressed in relation to assertions, the *solutions* designed to address the problems with or using assertions, and the types of *evaluation*, assessing the strength and relevance of the contributions.

We present existing literature surveys on testing in Section 2. Our methodology is outlined in Section 3, followed by the main results – the literature overview and synthesis in Section 4. We discuss our findings in Section 5, report limitations in Section 6, and conclude the paper in Section 7.

2 Background and Related work

Assertions are used to check program's behaviour at runtime: when an assertion evaluates to true (false), the program's behaviour is regarded "as intended" ("as erroneous") at the point of the assertion. They have gained significant attention and been used as a measure for code quality. Most dominantly, *program assertions* are used either to check the behaviour of the program, e.g., Blasi et al. [8], or to specify and check the contracts within the design by contract development.

Test oracle assertions (test assertions for short) are also used to specify and check the expected output of test cases [6]. Test assertions differ from program assertions as they check the expected output for one specific test case, while program assertions are typically located in the source code of the program, predicate on its variables, and return true or false throughout all its executions. Nevertheless, in many studies, program assertions and test oracle assertions are considered very closely or even interchangeably, e.g., Terragni et al. [74].

Specification assertions are also used to document programmers intent [16], i.e. modules are annotated with pre/post-conditions or invariants, e.g., JML. Specification assertions are basically non-executable and hence are inherently different from the other two types of assertions, although they seamlessly can be exploited at various stages of development for verification [54]. Our study basically focuses on test oracle assertions, yet designed to be inclusive of other types of assertions when they relate or contribute to testing.

Assertions (and their application) in software testing have been mostly studied under surveys on the test oracle problem, e.g., [6, 55, 61]. Among the 101 secondary studies, identified by Garousi and Mäntylä [32], only one is related to assertions, namely the one by Barr et al. [6] which reports on the roots of assertions, and existing support in languages and tools to use them for testing purposes. Surveys on automatic test generation techniques also consider assertions. Patel and Hierons [59] discuss the effectiveness and usability of assertions –

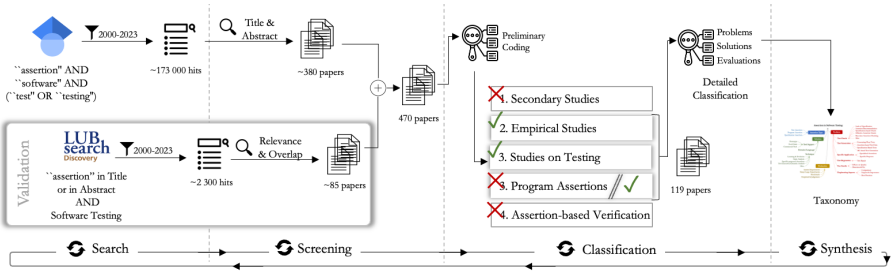

Fig. 1. Overview of the research method.

among others – in testing non-testable systems. In a mapping study on software test-code engineering, Garousi et al. [29] identify oracle assertion adequacy as a criterion of test-code quality assessment. In a survey on software testability [30], adding assertions is identified as an approach to improve testability. Winkler et al. [83] identify assertions as one of the factors affecting test code readability and understandability.

In summary, there are many secondary (and even tertiary) studies on software testing, but to our knowledge, there is no study specifically focusing on assertions used in software testing, and thus our survey fills a gap here.

3 Research method

This study provides an overview of research involving (different types of) *assertions* used in the context of software testing. We follow similar research procedures as used in the literature surveys conducted by Harman et al. [4,6], namely a term–based search in Google Scholar, followed by a filtering process, and finally synthesized in a qualitative analysis. This type of reviews, i.e., *semi-systematic reviews*, is proposed by Snyder [69], in particular, for a non-homogeneous concept (similar to the target of our survey), where systematic literature reviews would be too strict, and a narrative approach is more feasible. Kitchenham et al. [38] label a similar process *mapping study* that "may be auditable but not necessarily complete"; that they should have transparent procedures but the search scope may be limited. In this paper, we aim to "map a field of research, synthesize the state of knowledge, and create an agenda for further research" [69].

This survey was conducted in four major steps which were iterated in several cycles (demonstrated in Fig. 1). The first author was the main driver of the work, while the second author primarily took a validation role at each step.

1. Search To include also grey literature, Google scholar was used as the primary search engine [31], with a query defined as: "assertion" AND "software" AND ("test" OR "testing"). We limited the search in time to the 2000–2023 to get an overview of modern research on assertions, still partially overlapping with earlier surveys to ensure consistency (e.g. Barr et al. covered 1978–2012 [6]). The initial search rendered about 173 000 hits.

2. Screening The titles and abstracts were screened to find papers on assertions, although being inclusive when in doubt. After about 5 000 titles, no more relevant papers where found among the last dozens of titles. The screening resulted in a set of about 380 papers before further classification of the papers. To validate the search and screening, we used the same query in our university's library search portal, limiting the search to title and abstract within the context of software testing. The results were further screened for relevance and overlap resulting in 86 additional papers, and hence, the initial pool of about 470 papers.

3. Classification We then performed a preliminary coding of the papers, based on the type of the study and then the type of assertion, resulting in five categories: 1) secondary studies, 2) empirical studies, 3) studies explicitly on testing, 4) studies involving program assertions, and 5) studies on assertion-based verification. Firstly, we filtered out category 1 studies, as they were already considered under the related work. We also excluded the papers in category 5, since they are fundamentally related to hardware. Moreover, throughout the preliminary coding, we found out that the studies in category 4 are divided into two groups of 1) studies totally separate from testing, and 2) studies that are related to testing, and hence, we excluded the first group from the further classification. Accordingly, we came up with 119 papers on assertions related/contributing to testing. We further classified the remaining papers according to the design science elements of *problem*, *solution*, and *validation*.

4. Synthesis Finally, we synthesised the research from the perspectives of 1) the *problems* addressed, 2) the *solutions* presented, and 3) how they are *evaluated*. The design science perspectives are motivated by earlier research, concluding that this frame is feasible for software engineering research [25]. The results are presented in Section 4 accordingly. The complete listing of the synthesis (including 119 papers) is available as complementary material at `https://shorturl.at/ruCHL`.

4 Results

This section presents the results of reviewing the studies through characterising three aspects of each study: the addressed *problem* (Section 4.2), the main proposed *solution* (Section 4.3), and how the proposal was *evaluated* (Section 4.4). It also outlines the *type* of assertions considered in the studies (Section 4.1). Throughout a few iterations over the studies, these aspects were narrowed down using more fine-grained and consistent taxonomy (presented in Figure 2), that provides a comprehensive picture of the existing research on assertions in testing.

4.1 Assertion Types

Among our collection of studies, the three types of assertions are identified, which are considered for different purposes in the context of testing. Evidently, most

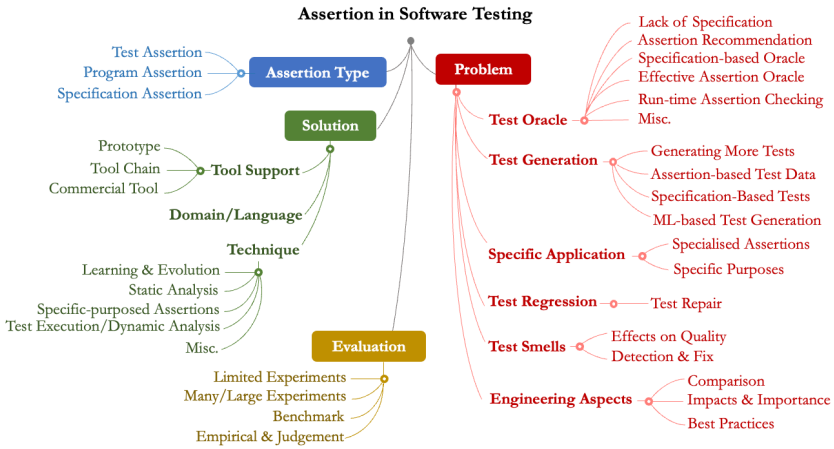

Fig. 2. Overview of the resulting taxonomy of the literature synthesis.

of the studies deal with *test assertions*, where assertions are manipulated as the result of performing other tasks, such as automatically generating assertions [90] or improving their effectiveness [18]. Empirical studies (e.g., [41,68]) also focus on test assertions investigating them from different perspectives. *Program assertions* are also found among the studies for testing purposes – rather than just program verification. In such studies, program assertions are employed as part of the *solution* in order to fulfill a goal, such as generating test data (e.g., [87, 92]). *Specification assertions* are also employed for generating tests (e.g., [23, 43]).

4.2 Assertion Problems

The *problem* aspect looks into the principal focus of the research. The problems, addressed by the collected studies, include test oracle, test generation, test regression, test smells, specific applications (e.g., Mobile Apps, GUI, ML), and test improvements. Note that these classes recognise the most distinguishing problem addressed by a piece of research, and hence, they are not necessarily disjoint.

Test Oracle While assertions are useful for specifying test oracles, writing and generating effective assertions are yet challenging [6]. Test oracle problem has been considered from different perspectives, including lack of specification, automatic generation of assertions statements, improving assertion oracles, and assertions based on specifications.

Specification of the intended behavior of the software under analysis is essential for assertion oracles. The *lack of such specification* has led to different techniques to capture the software behavior, and then generate assertions accordingly. Given an automatically generated test suite with no assertions, Ostra [86] collects objects' states, exercised by the test suite, and augments the test suite with new assertions specifying the behavior of a method. Zamprogno [91] propose

to automatically generate assertions for a given test case, based on its previous executions and feedback of the developer. EvoSpex [52] uses genetic algorithms to automatically produce a specification of the method's current behavior, in the form of postcondition assertions. Mesbah et al. [48] use a crawler to infer a state-flow graph of user interface states and then identify AJAX-specific faults and DOM-tree invariants that can serve as oracles. TOGA [22] is a unified transformer-based neural approach to infer both exceptional and assertion test oracles for a focal method, that in particular handles units with ambiguous or missing documentation.

Assertion recommendation focuses on automatic generation of candidate assertion statements. Agitator [9] applies software agitation to facilitate test automation and recommends assertions based on observations of a code's behaviour. DODONA [44] ranks program variables based on the interactions and dependencies, and accordingly proposes a set of variables to be monitored within test oracles. Pham et al. [33] generate candidate assertions based on test cases and then apply active learning techniques to iteratively refine them. DSpot [18] takes developer-written test cases as input and synthesizes improved versions of them by triggering new behaviors and adding new assertions. Valueian et al. [79] employ an Artificial Neural Network to construct automated oracles for low observable software based on tests inputs and verdict. Abdi et al. [1] address test amplification for dynamically typed languages (e.g., Pharo), and exploit profiling information to infer the necessary type information creating special test inputs with corresponding assertions.

OASIs [34] is a search-based tool for improving oracle, using test case generation and mutation testing to reveal false positives and false negatives, respectively. Given a set of assertions and a set of correct and incorrect program states, GAssert [74] employs a co-evolutionary algorithm that explores the space of possible assertions to identify oracle with fewer false positives and false negatives. Xie et al. [87] propose a mutation analysis approach for strengthening the assertions of parameterised unit tests. Fraser and Zeller [28] present a mutation-based assertion generation, within EvoSuite [27], optimised towards satisfying a coverage criterion. ATLAS [80] is a deep learning (DL)-based approach to generate meaningful assert statements for test methods based on existing unit tests. Yu et al. [90] introduce an IR-based assertion retrieval technique and a technique to adjust the assertions based on the context, that are more effective in generating a long sequence of tokens comparing to ATLAS. Tufano et al. [77] propose an approach to generate accurate and useful assertions using transformer model finetuned on the task of generating assert statements for unit tests.

Specification-based assertions can effectively reveal faults, up to their limit [17], and hence have been employed in specifying test oracle. Xie and Memon [85] automate GUI test oracles by inserting "assert" statements in test cases based on the formal specifications, i.e., pre/postconditions of GUI events. Zhao and Harris [94] introduce an approach to generate assertions directly from the natural language specifications employing semantic analysis of sentences in the specification document. Franke et al. [26] propose a method that identifies

life cycle dependent properties in the application specification, and derives test cases for validation. MeMo [8] automatically derives metamorphic equivalence relations from natural language documentation (given in Javadoc comments), which are then used as oracles in automatically generated test cases.

Runtime assertion checkers transparently ensure that the specification assertions hold during program execution [16]. JML (or an extension of JML) and its runtime assertion checker(s) are notably employed for testing in different context, such as testing conformance of safety-critical systems [73], specifying metamorphic relations [54], testing services in the Home Automation System [62], testing concurrent object-oriented software [5]. Cheon and Leavens [13, 14] propose to use a specification language's runtime assertion checker (e.g., JML) to decide whether methods work correctly, and hence automating the test oracles. Pastore et al. [58] introduce CrowdOracles, exploiting CrowdSourcing idea in the context of test oracle problem, and demonstrate that CrowdOracles are a viable solution to automate the oracle problem, yet taming the crowd to get useful results is a difficult task.

In summary, the studies in the *test oracle* category focus on how to generate assertions, what (kind of) information can be used for generating assertions, how to automate or augment the assertion generation process to have more effective assertions.

Test Generation Assertions have been considered in the context of test generation addressing different challenges including generating either complete tests or part of a test, such as test data and input/output pair. Mirshokraie et al. [51] leverage existing DOM-dependent assertions in human-written UI-based test cases to automatically generate assertions for unit-level testing of JavaScript code. TESTILIZER [50] learns from existing human-written assertions to generate assertions for unchecked portions of the web application.

In *Assertion-based Testing*, program assertions are combined with automated test (data) generation in order to find assertion violations effectively. Zeng et al. [92], automatically convert program dynamic invariants into program assertions, which are then used to direct the test generation process. Mayer [47] develops an assertion-based testing framework and a tool to generate runtime checks based on the specification annotations, for the Go programming language.

Specification assertions are also used as the basis to automatically generate tests. Korat [10] uses a method precondition to automatically generate all test cases up to a given size and the method postcondition as a test oracle. Similarly, Jarteg [56] randomly generates test cases for Java classes specified in JML, which are used to eliminate irrelevant test cases and serve as a test oracle. Søndergaard et al. [73] use JML annotations to model conformance constraints – in a safety-critical system – in order to generate JUnit tests as well as runtime assertion checks. Higher-level specification languages (and their assertions) are also employed for test generation. Li and Sun [43] translate Z formal models into their UML/OCL counterparts and JUnit tests (containing assertions). TestEra [37] generates test inputs based on Alloy specifications using Alloy SAT solver. Stoyanova et al. [72] introduce a test generation process based on WS-BPEL, having

assertions at different levels (HTTP, SOAP and BPEL variable), for testing web services. Drusinsky et al. [23] propose an automatic, JUnit-based, white-box testing of statechart prototypes augmented with statechart assertions.

Using more recent ML-based techniques, A3Test [2] presents a DL-based test case generation approach that uses a pre-trained language model of assertions to improve test case generation from language models (e.g., AthenaTest).

In summary, different types of assertions have been basically employed to direct *test generation* in order to generate complete tests or part of them, such as test data and test oracle.

Specific Applications *Specialised assertions* – in contrast to general-purpose assertions – have been introduced addressing special requirements in particular domains. For multi-agent system development, Tiryaki et al. [75] introduce a specialized assertion method for agent level verification. Delamare et al. [21] extend JUnit with new types of assertions to specify the expected joinpoints in aspect-oriented programming using AspectJ. Chang et al. [76] introduce visual assertions to verify whether certain GUI interaction generates the desired visual feedback. Koesnander et al. [40] introduce web macro assertions to encode the expectations and assumptions of a website developed by non-technical users.

Verification and validation of applications with inherent, uncertain outcomes (e.g., machine learning programs) requires new types of assertions. Dutta et al. [24] present `FLEX` which uses *approximate assertions* to compare the actual and expected values, while systematically identify the acceptable bound between the actual and expected output which minimizes flakiness. Kang et al. [36] introduce model assertions – that could be 'exact' or 'soft', which adapts the classical use of program assertions as a way to monitor and improve ML models.

Assertions are also adapted for *specific purposes*, in addition to typical testing, such as fault localisation, detecting merge conflicts, and test-suite reduction. Salehi Fathabadi et al. [64] use a formal model of the APIs of independently developed components to generate a set of assertions embedded in the implementation. Xuan and Monperrus [88] present spectrum-driven test case purification for improving fault localization, that generates purified versions of failing test cases, which include only one assertion per test. Sequeira [66] provide an automated technique to determine the DOM dependencies for each test assertion (on DOM), so that assertion failures are connected to the underlying JavaScript code which help finding the cause of failures. Pariente and Signoles [57] propose a method to trigger security counter-measures, based on static detection and runtime assertion checking of program weaknesses. Knauth et al. [39] recommend assertion-driven development instead of test-driven development and introduce meta-mutations at the code level to simulate common programmer errors. An assertion-aware test-suite reduction technique has been proposed by Chen et al. [12]. Messaoudi et al. [49] use assertion-based backward slicing to decompose complex system test cases into smaller, separate ones. Petke and Blot [60] suggest to consider the output of test case assertions in fitness functions for test-based program repair using genetic algorithms. Fang and Lam [95] introduce assertion

fingerprint to identify suitable candidates in refactoring test suites. `TOM` [35] is a tool that detects merge conflicts with the help of assertions that are defined on the variables that have different values.

In summary, the *specific applications* category demonstrates that assertions are useful for many different purposes. With specialised syntax and semantics, assertions may support specific problems more effectively.

Test Regression Regression tests can fail not only due to faults in the program but also due to obsolete tests which do not reflect the behavior of the updated program. Moonen et al. [53] introduce "test-driven refactoring" in that general code refactorings are induced by (re)structuring tests, for example to remove assertion roulette. Sakakibara et al. [63] develop an assertion-based mechanism to eliminate unnecessary dependencies between test code and objects in order to decrease invalidated tests due to changes in a code. ReAssert [19] automatically repairs broken unit tests by for example changing assertion methods. ReAssert combines analysis of a test's dynamic execution with analysis and transformation of the static structure of test code. WATER [15] suggests repairs for web application test scripts (test assertions), employing differential testing in that the behavior of tests on two successive versions of the application are compared and analysed. Xu et al. [89] introduce TestFix to fix broken JUnit test cases by synthesizing new method calls. TestFix regards the assertion of a broken test as a constraint and relies on the information about changes between versions of the software to guide the search of method-call sequences that meet the constraint.

In summary, the studies in *test regression* category largely address test obsolescence as the most known reason for test evolution, and introduce automatic test repair techniques that mostly focus on changing assertions and use assertion-based mechanisms.

Test Smells Test smells, poorly designed tests, negatively affect the comprehensibility and the maintainability of the test code [7], and therefore, they have been investigated and considered in many studies, e.g, [71] [20]. Assertion Roulette (i.e., several assertions with no explanation within the same test method) is found as the most frequent and riskiest test smell [84]. RAID [65] provides automated detection of lines of code affected by test smells, namely Assertion Roulette and Duplicate Assert, and a semi-automated refactoring for Java projects using JUnit. Soares et al. [70] present a set of refactorings – exploiting specific features of JUnit 5 – that help to remove test smells.RTj [46] is a framework for detecting and refactoring rotten green test cases, i.e., tests that pass but contain assertions that are never executed, using static analysis and dynamic analysis. Vahabzadeh et al. [78] recognise incorrect and missing assertions as the dominant root cause of silent horror test bugs, i.e., those test that pass, while the production code is incorrect. Wei et al. [81] introduce an ML-based approach for labelling unit tests according to the AAA pattern (i.e., the Arrangement, Action, and Assertion), as a best practice towards better code comprehension and less maintenance effort.

In summary, the studies in the *test smells* category largely aim to prevent test quality degradation due to badly designed tests and hence, introduce techniques to automatically detect test smells, in particular assertion roulette.

Engineering Aspects There are many studies that focus on, so-called, engineering aspects of using assertions in software development, including the impact of using assertions, comparison between different techniques or types of assertions, and good practices in using assertions. These studies consider assertions in a more general context in comparison to the aforementioned problems.

The application of assertions as test oracles is empirically investigated by Shrestha and Rutherford [68]. Li and Offutt [42] investigate the ability of test oracles (that vary in amount and frequency of program state checked) to reveal failures. The adequacy of assertions in test suite, particularly in the context of automated test generation has been investigated in several studies, e.g., [96] [67] [3]. Zhang and Mesbah [93] find a strong correlation between the number of assertions in a test suite with its effectiveness. The relation between developers' experience and assertion density is then investigated by Catolino et al. [11], showing that such experience is a significant factor in effective testing.

The effect of fluent assertions on comprehensibility of tests is investigated by Leotta et al. [41], demonstrating that adopting AssertJ (a fluent assertion library in JUnit) has no significant effect on the level of comprehension, though it significantly improves the efficiency in their comprehension. Ma'ayan [45] studied the quality of real world unit tests and reported that they don't follow the well-known good patterns (in particular using the right assertions) for writing tests.

In summary, the studies of the *engineering aspects* category tend to empirically investigate the application of assertions in software development in order to provide rigorous evidence of the benefits developers gain by using assertions and/or discover the best practices in the context.

4.3 Solutions

In order to a have an expressive view over the proposals in our collection, the solution of each study is characterised by 1) the main *technique(s)* that specifies the essence of the proposal, 2) the target *domain/language* for that the solution is ultimately actualised and implemented (if applicable), and 3) the *tooling* support which could be either a prototype implementation or within an existing tool. Note that the studies considering the engineering aspects are excluded herein, since they inherently do not provide any particular solution, in the way it is investigated in this section, except very few of them. Also, the information was collected based on the papers as the only source of our survey, and is hence limited to what is explicitly provided.

Technique By technique, the very core idea of the proposed solution is determined. While the technique(s) are (have to be) eventually implemented and hence, shaped within a context (e.g., language and domain) considering all of

its restrictions and capabilities, herein we abstract from such details and tend to provide a high-level view of the techniques within limit. The main classes of techniques, identified throughout our survey, are summarised in this section.

Learning and evolutionary algorithms have been used in several studies, particularly among the ones on assertion generation. Pham et al. [33] use active learning techniques to generate assertions. A combination of evolutionary and learning based techniques have been applied in EvoSpex [52] to automatically generate specifications. GASSERT [74] applies a co-evolutionary algorithm that explores the space of possible assertions to improve test oracles. Valueian et al. [79] employ an Neural Network algorithm to construct automated test oracles for low observable software. A3Test [2] uses a pre-trained language model of assertions to generate assertions in test case generation process.

The application of *static analysis* is considered as a promising technique in the literature, in different context. Zeng et al. [92] automatically generate assertions based on program invariants. Pariente and Signoles [57] generate runtime assertions checks based on static detection of CWEs[3].

A number of studies *exploit test execution* in generating or improving test oracles. Xie [86] adds assertions based on the object states collected throughout previous test executions. Employing a search-based algorithm for improving assertions, Jahangirova [34] combine test case generation to reveal false positives and mutation testing to reveal false negatives. Test case execution logs are used in DS3 [49] to determine dependencies among test slices. Mutation analysis has been also used by Fraser and Zeller [28] to improve the fault detection capability of test oracles, by Knauth et al. [39] to assess the quality of the assertions, and by Xie et al. [87] for analyzing PUTs written by developers and identifying likely locations in PUTs for improvement.

In several studies, a *specific-purpose assertion* is introduced, that is typically defined on top of an existing assertion language/construct, through an extended syntax and semantics, and a novel assertion evaluation technique. Corduroy [54] introduces metamorphic assertions, built on top of Java Modelling Language (JML). Model assertions [36] adapt the classical use of program assertions, tailored to the specific needs of ML programs, in particular uncertainty in output.

Domain/Language A wide range of domains and languages are considered by the collected papers, though with different density. In addition to solutions for general and typical programs, that are the target of many studies, the proposed solution in many studies are applicable to specific types of programs, e.g., Machine Learning programs [24] [36], web/mobile applications [26], and GUI [85].

The solutions can also be characterised regarding the language for which the solution is introduced. While the most common language is Java (e.g., [86] [34] [33] [74] [52]), a variety of other general-/specific-purpose languages have been covered, including JavaScript/TypeScript [91], Go [47], and Pharo Smalltalk [1]. Other solutions (e.g., [79] [49]) are not limited to a specific programming language and are applicable to programs in different languages. For example, Val-

[3] Common Weakness Enumerations – `https://cwe.mitre.org`

Table 1. Evaluation Methods vs. Assertion Problems

Problem	Evaluation Method			
	Limited	**Many/Large**	**Benchmark**	**Empirical**
Test Oracle	16	18	2	3
Test Generation	12	4	1	-
Specific Application	6	5	1	3
Test Regression	4	1	-	-
Test Smells	2	1	-	13
Engineering Aspects	3	1	1	17
Total	**43**	**30**	**5**	**36**

ueian et al. [79] demonstrate the application of their solution on programs in Java, C, C++, Verilog, and VHDL. There are also a number of studies that consider a higher level of abstraction and introduce their solutions for specific types of models, such as UML statecharts [23], Alloy models [37], Z Specification [43], WS-BPEL [72], and Machine Learning models [36].

Tool Support Most of the solutions are embedded in and supported by *tool prototypes* that are typically available online. A number of studies use a chain of available tools to implement and demonstrate their solutions (e.g., [17] [77]). One study [9] introduces its solution as part of a commercial tool (Agitator). Studies in the engineering aspects category and the empirical studies are exempted to have prototypes or any other implementation support and few papers (e.g., [92] [60]) have not explicitly mentioned how the solution is implemented.

4.4 Evaluation

Looking into how the proposals of the collected studies have been evaluated, we identified four main classes of the evaluation methods, namely *limited experiments, many/large experiments, benchmarks, and empirical & judgement*, that are described in the following. Note that most of the studies, excluding the ones looking into the engineering aspects, provide a proof of concept through developing a prototype of the tooling support for their proposed solutions, which is not considered herein as evaluation. There are few papers that do not present any evaluation which is however compatible to their types of study, such as short paper (e.g., [60]) or report on ongoing study (e.g., [87]). Table 1 summarises evaluation methods w.r.t. the assertion problems.

Limited Experiments This type of evaluation provides preliminary and limited evidence of the application of the proposed techniques or tools, in that, for example, the effectiveness of the proposals and how the proposal meets its goal(s), is demonstrated throughout a limited number of case studies (e.g., up

to 10 cases), e.g., [94] [64], or by limited artificial experiments (e.g., by manually generating or adding required information [68] [39]). In our collection of 119 papers, the evaluation of 43 studies fall into this category; the studies focusing on *test generation* and *test oracle/assertion generation* have the main portion among this group (28 studies in total).

Many/Large Experiments Several studies provide more convincing evaluation results by assessing their solutions on many cases (e.g. > 10) or throughout one or more experiments in an industrial setting. Large, open-source or public projects or repositories, for example on GitHub, have been used in evaluation experiments (e.g., [86] [33] [12]), that is, mostly used in the studies that address *assertion generation*. Some of the studies use real systems/applications that are under operation to demonstrate the usefulness and/or the cost-effectiveness of their proposals, such as using an Aircraft e-Maintenance application [57].

Benchmarks Few studies have used benchmarks to evaluate and demonstrate properties of their solutions. Different sets of benchmarks (e.g., regarding size, application, and domain) were used depending on the target and context of a study. Messaoudi et al. [49] use a proprietary benchmark of 30 complex system test cases to assess the effectiveness and efficiency of their solution in slicing system test cases. The quality of EvoSpex [52] was assessed on a benchmark of open source Java projects in SF110[4]. Alagarsamy et al. [2] use Defects4J repository to evaluate A3Test's performance. Ji et al. [35] firstly design the benchmark MCon4j and then use it to evaluate the effectiveness of their solutions.

Empirical & Judgement Some of the studies investigate and demonstrate *empirical evidence* regarding a particular research question or of the use of a technique or tool in practice. They may use surveys or interview among a certain number of participants (e.g., [91]), or use more formal experimental methods (e.g., controlled experiment [41]). Most of the studies in this category, look into the *engineering aspects* of the use of assertions, that is however obvious considering their intention.

5 Discussion

This section summarizes the research findings following the same structure we used to review our collection of studies, and synthesise the results.

Assertion Problems. The dominant problem focus is the oracle problem. About 34% of the studies (41) address the substantial challenge of specifying the expected output or behaviour in tests using assertions. They largely investigate different types of information that can be used for generating or defining

[4] https://www.evosuite.org/experimental-data/sf110/

test oracle (assertions) and how to automate or augment the assertion generation process to improve effectiveness, efficiency, and practicality.

Engineering aspects is the second premier focus. About 20% of studies provide empirical evidence of the benefits to gain by using assertions and also point out challenges and obstacles in effective application of assertions in practice.

The third group of studies (about 15%) employ assertions to direct test generation tasks, such as generating test data. The use of assertions for specific applications, addressed in 16 studies, demonstrates that assertions are useful and could support specific problems more effectively. Among different specific domains, limited studies address uncertainty in outputs, which however, considering the emerging use of ML, require more research. The same of number of studies focus on poorly designed tests. Most of these studies investigate how test smells affect test quality, whereas few of them introduce techniques to detect and fix test smells. Finally, few studies address test regression due to program evolution which mostly introduce automatic test repair techniques.

Solutions. Most of the solutions are embedded in and supported by tool prototypes that are generally available online. About 85% of the studies excluding those considering the engineering aspects, since they inherently do not provide any particular solution.

As described in Section 4.3, many and various techniques have been previously introduced in the literature and therefore, they are not completely categorised. However, a number of techniques and ideas are more visible among others. Learning and evolutionary algorithms have been used as a promising technique in many recent studies (20 out of 119 papers), particularly among the ones focusing on test oracle and test generation. Nearly the same amount of papers suggest integrating static analysis and dynamic testing to improve the effectiveness of either testing and/or static program analysis. Defining a specific-purpose assertion language, including syntax, semantics, and possibly a new assertion checking method, is a common proposal among the studies, e.g., the studies addressing uncertainty in output.

While a wide range of domains and languages are considered in the collected papers, general software programs and C/C++ and Java programming languages are the target of the most of the studies (about 60%). While Java is a broadly used programming language, it is important for the assertions research to also take other languages into account. For example, in machine learning applications, Python is frequently used, which may be a specific target for assertions.

Evaluation. As demonstrated in Table 1, the largest set of studies have been evaluated throughout limited number of cases. The evaluation of 43 studies, out of 119 papers, fall into this category, where the studies focusing on test oracle and test generation have the main portion among this group (28 studies in total). Empirical and judgement is the next more common evaluation method, that is obviously used in the studies that focus on engineering aspects and also the studies on test smells. A quarter of the studies, largely on test oracle, evaluate their proposals using many experiments or within large–scale industrial cases. Benchmarks are used in five studies.

To ensure the relevance for practice, research has to go beyond small scale proofs of concept. Among the surveyed studies, one third are evaluated in more realistic cases, which is promising. However, for future research, we would like to see even more focus on the scaling and relevance aspects.

6 Limitations

The main issues related to threats to validity of this survey are incomplete set of studies in our collection and imprecise data extraction that are fundamentally because of the researcher bias in choosing search terms, the search engine, and the targeted databases, as well as, the exclusion/inclusion criteria. A very basic method to address these issues is to conduct a survey in a structured way; we accordingly carried out a semi-systematic review throughout four major steps, which were iterated in several cycles and carefully defined and reported.

To reduce the risk of incomplete set of primary sources, Google Scholar was used with a general search query which would render a large amount of studies, including grey literature, as the initial pool. To minimise researchers' bias, the second author took a validation role and double checked the work done by the first author. Design science paradigm was used as a lens for this survey, that was motivated by earlier research concluding that this frame is feasible for software engineering research. In order to ensure conclusion validity, the classification and synthesis were performed repeatedly, and the outcome of each turn was discussed between the authors to avoid any misunderstanding.

7 Conclusion

In this survey, we provide an overall picture of research work on assertions in software testing, within the last two decades of research. Using a term–based search, a collection of relevant papers was selected and then the papers were reviewed and synthesised with respect to the design science elements, namely the problem addressed, the solution proposed, and the evaluation method. The synthesis demonstrated that test oracle is the dominant problem focus, followed by engineering aspects of assertions and assertions in test generation. Solutions include a wide range of techniques and are typically embedded in tool prototypes. They are mostly consider general applications and languages, e.g., Java. This however, suggest to consider other languages that are getting attention more recently (e.g., Python). The proposals are by large evaluated within a limited number of cases while using large–scale industrial settings is also visible. Nevertheless, in order to support practice, research has to go beyond small scale experiments since scaling up analyses to large code bases is an essential challenge. We conclude that assertions would be worth more attention in future research, particularly regarding the new and emerging demands (e.g., wide-spread applications of software, verification of applications with uncertain outputs), for effective, applicable, and domain-specific solutions, as well as more focus on the scaling and relevance aspects.

Acknowledgements. This work is funded by the ELLIIT strategic research area (`https://elliit.se`), project 'A19 – Software Regression Testing with Near Failure Assertions'.

References

1. Abdi, M., Rocha, H., Demeyer, S., Bergel, A.: Small-amp: Test amplification in a dynamically typed language. Empir. Softw. Eng. **27**(6), 128 (2022). https://doi.org/10.1007/s10664-022-10169-8
2. Alagarsamy, S., Tantithamthavorn, C., Aleti, A.: A3test: Assertion-augmented automated test case generation (2023). https://doi.org/10.48550/ARXIV.2302.10352
3. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: Finding real faults in a financial application. In: IEEE/ACM Int. Conf. on Software Engineering: Software Engineering in Practice Track. pp. 263–272 (2017). https://doi.org/10.1109/ICSE-SEIP.2017.27
4. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013). https://doi.org/10.1016/j.jss.2013.02.061
5. Araujo, W., Briand, L., Labiche, Y.: On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In: Int. Symposium on Empirical Software Engineering and Measurement. pp. 10–19 (2011). https://doi.org/10.1109/ESEM.2011.9
6. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785
7. Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D.: Are test smells really harmful? an empirical study. Empir. Softw. Eng. **20**(4), 1052–1094 (2015). https://doi.org/10.1007/s10664-014-9313-0
8. Blasi, A., Gorla, A., Ernst, M.D., Pezzè, M., Carzaniga, A.: MeMo: Automatically identifying metamorphic relations in javadoc comments for test automation. J. of Sys. & Softw. **181**, N.PAG–N.PAG (2021). https://doi.org/10.1016/j.jss.2021.111041
9. Boshernitsan, M., Doong, R., Savoia, A.: From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In: ACM Int. symposium on Software testing and analysis. pp. 169–180. ISSTA '06, ACM (2006). https://doi.org/10.1145/1146238.1146258
10. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. ACM SIGSOFT Software Engineering Notes **27**(4), 123–133 (2002). https://doi.org/10.1145/566171.566191
11. Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F.: How the experience of development teams relates to assertion density of test classes. In: IEEE Int. Conf. on Software Maintenance and Evolution. pp. 223–234 (2019). https://doi.org/10.1109/ICSME.2019.00034, ISSN: 2576-3148
12. Chen, J., Bai, Y., Hao, D., Zhang, L., Zhang, L., Xie, B.: How do assertions impact coverage-based test-suite reduction? In: IEEE Int. Conf. on Software Testing, Verification and Validation. pp. 418–423 (2017). https://doi.org/10.1109/ICST.2017.45
13. Cheon, Y., Kim, M., Perumandla, A.: A complete automation of unit testing for java programs. Tech. Rep. UTEP-CS-05-05, University of Texas at El Paso (2005), `https://scholarworks.utep.edu/cs_techrep/234`

14. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP — Object-Oriented Programming. pp. 231–255. Lecture Notes in Computer Science, Springer (2002). https://doi.org/10.1007/3-540-47993-7_10
15. Choudhary, S.R., Zhao, D., Versee, H., Orso, A.: WATER: Web application TEst repair. In: ACM Int Workshop on End-to-End Test Script Engineering. pp. 24–29. ETSE '11, ACM (2011). https://doi.org/10.1145/2002931.2002935
16. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes **31**(3), 25–37 (2006). https://doi.org/10.1145/1127878.1127900
17. Coppit, D., Haddox-Schatz, J.: On the use of specification-based assertions as test oracles. In: IEEE/NASA Software Engineering Workshop. pp. 305–314 (2005). https://doi.org/10.1109/SEW.2005.33, ISSN: 1550-6215
18. Danglot, B., Vera-Pe´rez, O., Baudry, B., Monperrus, M.: Automatic test improvement with DSpot: a study with ten mature open-source projects. Empir. Softw. Eng. **24**(4), 2603–2635 (2019). https://doi.org/10.1007/s10664-019-09692-y
19. Daniel, B., Gvero, T., Marinov, D.: On test repair using symbolic execution. In: ACM International Symposium on Software Testing and Analysis. pp. 207–218. ISSTA '10, ACM (2010). https://doi.org/10.1145/1831708.1831734
20. De Stefano, M., Pecorelli, F., Di Nucci, D., De Lucia, A.: A preliminary evaluation on the relationship among architectural and test smells. In: IEEE Int. Working Conf. on Source Code Analysis and Manipulation. pp. 66–70 (2022). https://doi.org/10.1109/SCAM55253.2022.00013, ISSN: 2470-6892
21. Delamare, R., Baudry, B., Ghosh, S., Le Traon, Y.: A test-driven approach to developing pointcut descriptors in AspectJ. In: IEEE Int. Conf. on Software Testing Verification and Validation. IEEE Comput. Soc. (2009). https://doi.org/10.1109/ICST.2009.41
22. Dinella, E., Ryan, G., Mytkowicz, T., Lahiri, S.K.: TOGA: A neural method for test oracle generation. In: IEEE/ACM Int. Conf. on Software Engineering. pp. 2130–2141. ACM (2022). https://doi.org/10.1145/3510003.3510141
23. Drusinsky, D., Shing, M.T., Demir, K.: Creation and validation of embedded assertion statecharts. In: IEEE International Workshop on Rapid System Prototyping. pp. 17–23 (2006). https://doi.org/10.1109/RSP.2006.12, ISSN: 1074-6005
24. Dutta, S., Shi, A., Misailovic, S.: FLEX: fixing flaky tests in machine learning projects by updating assertion bounds. In: ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering. pp. 603–614. ESEC/FSE 2021, ACM (2021). https://doi.org/10.1145/3468264.3468615
25. Engström, E., Storey, M., Runeson, P., Höst, M., Baldassarre, M.T.: How software engineering research aligns with design science: A review. Empir. Softw. Eng. **25**, 2630–2660 (2020). https://doi.org/10.1007/s10664-020-09818-7
26. Franke, D., Kowalewski, S., Weise, C., Prakobkosol, N.: Testing conformance of life cycle dependent properties of mobile applications. In: IEEE Int. Conf. on Software Testing, Verification and Validation. pp. 241–250 (2012). https://doi.org/10.1109/ICST.2012.104, ISSN: 2159-4848
27. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: ACM SIGSOFT symposium and European conference on Foundations of software engineering. pp. 416–419. ESEC/FSE '11, ACM (2011). https://doi.org/10.1145/2025113.2025179
28. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Trans. Softw. Eng. **38**(2), 278–292 (2012). https://doi.org/10.1109/TSE.2011.93

29. Garousi, V., Amannejad, Y., Betin Can, A.: Software test-code engineering: A systematic mapping. Inf. Softw. Technol. **58**, 123–147 (2015). https://doi.org/10.1016/j.infsof.2014.06.009
30. Garousi, V., Felderer, M., Kılıçaslan, F.N.: A survey on software testability. Inf. Softw. Technol. **108**, 35–64 (2019). https://doi.org/10.1016/j.infsof.2018.12.003
31. Garousi, V., Felderer, M., Mäntylä, M.V.: Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf. Softw. Technol. **106**, 101–121 (2019). https://doi.org/10.1016/j.infsof.2018.09.006
32. Garousi, V., Mäntylä, M.V.: A systematic literature review of literature reviews in software testing. Inf. Softw. Technol. **80**, 195–216 (2016). https://doi.org/10.1016/j.infsof.2016.09.002
33. H. Pham, L., Tran Thi, L.L., Sun, J.: Assertion Generation Through Active Learning. In: Duan, Z., Ong, L. (eds.) Formal Methods and Software Engineering. pp. 174–191. LNCS, Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_11
34. Jahangirova, G., Clark, D., Harman, M., Tonella, P.: OASIs: oracle assessment and improvement tool. In: ACM SIGSOFT Int. Symposium on Software Testing and Analysis. ISSTA 2018, ACM, New York, NY, USA (Jul 2018). https://doi.org/10.1145/3213846.3229503
35. Ji, T., Chen, L., Mao, X., Yi, X., Jiang, J.: Automated regression unit test generation for program merges (2020), `http://arxiv.org/abs/2003.00154`
36. Kang, D., Raghavan, D., Bailis, P., Zaharia, M.: Model assertions for monitoring and improving ML models (2020), `http://arxiv.org/abs/2003.01668`
37. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of java programs using SAT. Autom. Softw. Eng. **11**(4), 403–434 (2004). https://doi.org/10.1023/B:AUSE.0000038938.10589.b9
38. Kitchenham, B.A., Budgen, D., Brereton, O.P.: Using mapping studies as the basis for further research – a participant-observer case study. Inf. Softw. Technol. **53**(6), 638–651 (Jun 2011). https://doi.org/10.1016/j.infsof.2010.12.011
39. Knauth, T., Fetzer, C., Felber, P.: Assertion-driven development: Assessing the quality of contracts using meta-mutations. In: IEEE Int. Conf. on Software Testing, Verification, and Validation Workshops. pp. 182–191 (2009). https://doi.org/10.1109/ICSTW.2009.40
40. Koesnandar, A., Elbaum, S., Rothermel, G., Hochstein, L., Scaffidi, C., Stolee, K.T.: Using assertions to help end-user programmers create dependable web macros. In: ACM SIGSOFT Int. Symposium on Foundations of software engineering. pp. 124–134. SIGSOFT '08/FSE-16, ACM (2008). https://doi.org/10.1145/1453101.1453119
41. Leotta, M., Cerioli, M., Olianas, D., Ricca, F.: Fluent vs basic assertions in java: An empirical study. In: Int. Conf. on the Quality of Information and Communications Technology (QUATIC). pp. 184–192 (2018). https://doi.org/10.1109/QUATIC.2018.00036
42. Li, N., Offutt, J.: Test oracle strategies for model-based testing. IEEE Trans. on Softw. Eng. **43**(4), 372–395 (2017). https://doi.org/10.1109/TSE.2016.2597136
43. Li, P., Sun, J., Wang, H.: Formal approach to assertion-based code generation. Int. J. of Softw. Eng. and Knowl. Eng. **27**(9), 1637–1662 (2017). https://doi.org/10.1142/S0218194017400162, publisher: World Scientific Publishing Co.
44. Loyola, P., Staats, M., Ko, I., Rothermel, G.: Dodona: automated oracle data set selection. In: ACM Int. Symposium on Software Testing and Analysis. pp. 193–203. ISSTA 2014, ACM (2014). https://doi.org/10.1145/2610384.2610408

45. Ma'ayan, D.D.: The quality of junit tests: An empirical study report. In: IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies. pp. 33–36 (2018)
46. Martinez, M., Etien, A., Ducasse, S., Fuhrman, C.: RTj: a java framework for detecting and refactoring rotten green test cases. In: IEEE/ACM Int. Conf. on Software Engineering: ICSE-Companion. pp. 69–72 (2020), publisher: ACM
47. Mayer, E.C.: Assertion-based testing of go programs. Master thesis, Technical University Munich (2020)
48. Mesbah, A., van Deursen, A., Roest, D.: Invariant-based automatic testing of modern web applications. IEEE Trans. on Softw. Eng. **38**(1), 35–53 (2012). https://doi.org/10.1109/TSE.2011.28
49. Messaoudi, S., Shin, D., Panichella, A., Bianculli, D., Briand, L.C.: Log-based slicing for system-level test cases. In: Cadar, C., Zhang, X. (eds.) ACM SIGSOFT Int. Symposium on Software Testing and Analysis. pp. 517–528. ACM (2021). https://doi.org/10.1145/3460319.3464824
50. Milani Fard, A., Mirzaaghaei, M., Mesbah, A.: Leveraging existing tests in automated test generation for web applications. In: ACM/IEEE Inte. Conf. on Automated Software Engineering. pp. 67–78. ASE '14, ACM (2014). https://doi.org/10.1145/2642937.2642991
51. Mirshokraie, S., Mesbah, A., Pattabiraman, K.: Atrina: Inferring unit oracles from GUI test cases. In: IEEE Int. Conf. on Software Testing, Verification and Validation (ICST). IEEE Computer Society (2016). https://doi.org/10.1109/ICST.2016.32
52. Molina, F., Ponzio, P., Aguirre, N., Frias, M.: EvoSpex: An Evolutionary Algorithm for Learning Postconditions (artifact). In: IEEE/ACM Int. Conf. on Software Engineering: ICSE-Companion. pp. 185–186 (May 2021). https://doi.org/10.1109/ICSE-Companion52605.2021.00080, iSSN: 2574-1926
53. Moonen, L., van Deursen, A., Zaidman, A., Bruntink, M.: On the interplay between software testing and evolution and its effect on program comprehension. In: Mens, T., Demeyer, S. (eds.) Software Evolution, pp. 173–202. Springer (2008). https://doi.org/10.1007/978-3-540-76440-3_8
54. Murphy, C., Shen, K., Kaiser, G.: Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In: IEEE Int. Conf. on Software Testing Verification and Validation. pp. 436–445 (2009). https://doi.org/10.1109/ICST.2009.19, ISSN: 2159-4848
55. Oliveira, R.A.P., Kanewala, U., Nardi, P.A.: Chapter three - automated test oracles: State of the art, taxonomies, and trends. In: Memon, A. (ed.) Advances in Computers, vol. 95, pp. 113–199. Elsevier (2014). https://doi.org/10.1016/B978-0-12-800160-8.00003-6
56. Oriat, C.: Jartege: A tool for random generation of unit tests for java classes. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) Quality of Software Architectures and Software Quality. pp. 242–256. Lecture Notes in Computer Science, Springer (2005). https://doi.org/10.1007/11558569_18
57. Pariente, D., Signoles, J.: Static analysis and runtime-assertion checking : Contribution to security counter-measures (2017), `https://zenodo.org/record/820856`
58. Pastore, F., Mariani, L., Fraser, G.: CrowdOracles: Can the crowd solve the oracle problem? In: IEEE Int. Conf. on Software Testing, Verification and Validation. pp. 342–351 (2013). https://doi.org/10.1109/ICST.2013.13, publisher: IEEE
59. Patel, K., Hierons, R.M.: A mapping study on testing non-testable systems. Software Quality Journal **26**(4), 1373–1413 (2018). https://doi.org/10.1007/s11219-017-9392-4

60. Petke, J., Blot, A.: Refining fitness functions in test-based program repair. In: IEEE/ACM Int. Conf. on Software Engineering Workshops. pp. 13–14. ICSEW'20, ACM (2020). https://doi.org/10.1145/3387940.3392180
61. Pezzè, M., Zhang, C.: Chapter one - automated test oracles: A survey. In: Memon, A. (ed.) Advances in Computers, vol. 95, pp. 1–48. Elsevier (2014). https://doi.org/10.1016/B978-0-12-800160-8.00001-2
62. Rajan, A., du Bousquet, L., Ledru, Y., Vega, G., Richier, J.L.: Assertion-based test oracles for home automation systems. In: ACM Int. Workshop on Model-Based Methodologies for Pervasive and Embedded Software. p. 45–52. MOMPES '10, ACM (2010). https://doi.org/10.1145/1865875.1865882
63. Sakakibara, M., Sakurai, K., Komiya, S.: An assertion mechanism for software unit testing to remain unaffected by program modification - the mechanism to eliminate dependency from/to unnecessary object. Knowledge-Based Software Engineering pp. 125–134 (2008). https://doi.org/10.3233/978-1-58603-900-4-125
64. Salehi Fathabadi, A., Dalvandi, M., Butler, M., Al-Hashimi, B.M.: Verifying cross-layer interactions through formal model-based assertion generation. IEEE Embedded Systems Letters **12**(3), 83–86 (2020). https://doi.org/10.1109/LES.2019.2955316
65. Santana, R., Martins, L., Rocha, L., Virgínio, T., Cruz, A., Costa, H., Machado, I.: RAIDE: a tool for assertion roulette and duplicate assert identification and refactoring. In: Brazilian Symposium on Software Engineering. pp. 374–379. SBES '20, ACM (2020). https://doi.org/10.1145/3422392.3422510
66. Sequeira, S.: Understanding web application test assertion failures. Ph.D. thesis, University of British Columbia (2014). https://doi.org/10.14288/1.0167024
67. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 201–211 (2015). https://doi.org/10.1109/ASE.2015.86
68. Shrestha, K., Rutherford, M.J.: An empirical evaluation of assertions as oracles. In: IEEE Int. Conf. on Software Testing, Verification and Validation. pp. 110–119 (2011). https://doi.org/10.1109/ICST.2011.50, ISSN: 2159-4848
69. Snyder, H.: Literature review as a research methodology: An overview and guidelines. J. of Business Res. **104**, 333–339 (2019). https://doi.org/10.1016/j.jbusres.2019.07.039
70. Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., Santos, A.: Refactoring test smells with JUnit 5: Why should developers keep up-to-date? IEEE Trans. on Softw. Eng. **49**(3), 1152–1170 (2023). https://doi.org/10.1109/TSE.2022.3172654
71. Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A.: On the relation of test smells to software code quality. In: IEEE Int. Conf. on Software Maintenance and Evolution. pp. 1–12 (2018). https://doi.org/10.1109/ICSME.2018.00010, ISSN: 2576-3148
72. Stoyanova, V., Petrova-Antonova, D., Ilieva, S.: Automation of test case generation and execution for testing web service orchestrations. In: IEEE Int. Symposium on Service-Oriented System Engineering. pp. 274–279 (2013). https://doi.org/10.1109/SOSE.2013.9
73. Søndergaard, H., Korsholm, S., Ravn, A.: Conformance test development with the java modeling language. Concurrency and Computation: Practice and Experience **29**(22), (32 pp.) (2017). https://doi.org/10.1002/cpe.4071
74. Terragni, V., Jahangirova, G., Tonella, P., Pezzè, M.: GAssert: A Fully Automated Tool to Improve Assertion Oracles. In: IEEE/ACM Int. Conf. on Software

Engineering: Companion Proceedings (ICSE-Companion). pp. 85–88 (May 2021). https://doi.org/10.1109/ICSE-Companion52605.2021.00042, iSSN: 2574-1926
75. Tiryaki, A., Öztuna, S., Dikenelli, O., Erdur, R.: SUNIT: A unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (eds.) Agent-Oriented Software Engineering VII. pp. 156–173. LNCS, Springer (2007). https://doi.org/10.1007/978-3-540-70945-9_10
76. Tsung-Hsiang, C., Yeh, T., Miller, R.C.: GUI testing using computer vision. In: SIGCHI Conf. on Human Factors in Computing Systems. pp. 1535–1544. ACM (2010). https://doi.org/10.1145/1753326.1753555
77. Tufano, M., Drain, D., Svyatkovskiy, A., Sundaresan, N.: Generating accurate assert statements for unit test cases using pretrained transformers. In: ACM/IEEE International Conf. on Automation of Software Test. pp. 54–64. AST '22, ACM (2022). https://doi.org/10.1145/3524481.3527220
78. Vahabzadeh, A., Milani Fard, A., Mesbah, A.: An empirical study of bugs in test code. In: IEEE Int. Conf. on Software Maintenance and Evolution. pp. 101–110 (2015). https://doi.org/10.1109/ICSM.2015.7332456
79. Valueian, M., Attar, N., Haghighi, H., Vahidi-Asl, M.: Constructing automated test oracle for low observable software. Scientia Iranica **27**(3), 1333–1351 (2020). https://doi.org/10.24200/sci.2019.51494.2219
80. Watson, C., Tufano, M., Moran, K., Bavota, G., Poshyvanyk, D.: On learning meaningful assert statements for unit test cases. In: ACM/IEEE Int. Conf. on Software Engineering. pp. 1398–1409. ICSE '20, ACM (2020). https://doi.org/10.1145/3377811.3380429
81. Wei, C., Xiao, L., Yu, T., Chen, X., Wang, X., Wong, S., Clune, A.: Automatically tagging the "AAA" pattern in unit test cases using machine learning models. In: IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 1–3. ASE '22, ACM (2023). https://doi.org/10.1145/3551349.3559510
82. Weyuker, E.J.: On testing non-testable programs. Comput. J. **25**(4), 465–470 (1982). https://doi.org/10.1093/comjnl/25.4.465
83. Winkler, D., Urbanke, P., Ramler, R.: What do we know about readability of test code? - a systematic mapping study. In: IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER). pp. 1167–1174 (2022). https://doi.org/10.1109/SANER53432.2022.00135, ISSN: 1534-5351
84. Wu, H., Yin, R., Gao, J., Huang, Z., Huang, H.: To what extent can code quality be improved by eliminating test smells? In: Int. Conf. on Code Quality. pp. 19–26 (2022). https://doi.org/10.1109/ICCQ53703.2022.9763153
85. Xie, Q., Memon, A.M.: Designing and comparing automated test oracles for GUI-based software applications. ACM Trans. on Softw. Eng. Methodol. **16**(1), 4–es (2007). https://doi.org/10.1145/1189748.1189752
86. Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas, D. (ed.) ECOOP – Object-Oriented Programming. pp. 380–403. LNCS, Springer (2006). https://doi.org/10.1007/11785477_23
87. Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Mutation analysis of parameterized unit tests. In: IEEE Int. Conf. on Software Testing, Verification, and Validation Workshops. pp. 177–181 (2009). https://doi.org/10.1109/ICSTW.2009.43
88. Xuan, J., Monperrus, M.: Test case purification for improving fault localization. In: ACM/SIGSOFT Int. Symp. on Foundations of Software Engineering. pp. 52–63. ACM (2014). https://doi.org/10.1145/2635868.2635906
89. Y. Xu, B. Huang, G. Wu, M. Yuan: Using genetic algorithms to repair JUnit test cases. In: Asia-Pacific Software Engineering Conf. vol. 1. IEEE Computer Society (2014). https://doi.org/10.1109/APSEC.2014.51

90. Yu, H., Lou, Y., Sun, K., Ran, D., Xie, T., Hao, D., Li, Y., Li, G., Wang, Q.: Automated assertion generation via information retrieval and its integration with deep learning. In: IEEE/ACM Int. Conf. on Software Engineering. pp. 163–174 (2022). https://doi.org/10.1145/3510003.3510149, publisher: ACM
91. Zamprogno, L., Hall, B., Holmes, R., Atlee, J.M.: Dynamic human-in-the-loop assertion generation. IEEE Trans. on Softw. Eng. **49**(4), 2337–2351 (2023). https://doi.org/10.1109/TSE.2022.3217544
92. Zeng, F., Deng, C., Yuan, Y.: Assertion-directed test case generation. In: World Congress on Software Engineering. pp. 41–45 (2012). https://doi.org/10.1109/WCSE.2012.16
93. Zhang, Y., Mesbah, A.: Assertions are strongly correlated with test suite effectiveness. In: ACM Joint Meeting on Foundations of Software Engineering. pp. 214–224. ESEC/FSE 2015, ACM (2015). https://doi.org/10.1145/2786805.2786858
94. Zhao, J., Harris, I.G.: Automatic Assertion Generation from Natural Language Specifications Using Subtree Analysis. In: Design, Automation Test in Europe Conf. Exhibition (DATE). pp. 598–601 (Mar 2019). https://doi.org/10.23919/DATE.2019.8714857, iSSN: 1558-1101
95. Zheng, F., Lam, P.: Identifying test refactoring candidates with assertion fingerprints. In: ACM Principles and Practices of Programming on The Java Platform. pp. 125–137. PPPJ '15, ACM (2015). https://doi.org/10.1145/2807426.2807437
96. Zhi, J., Garousi, V.: On adequacy of assertions in automated test suites: An empirical investigation. In: IEEE Int. Conf. on Software Testing, Verification and Validation Workshops. pp. 382–391 (2013). https://doi.org/10.1109/ICSTW.2013.49