Automated Test Generation
Taxonomy and Tool Applications

Eduard Enoiu[1][0000−0003−2416−4205], Nasir Mehmood Minhas[1][0000−0001−8177−4355], Michael Felderer[2,3][0000−0003−3818−4442], and Wasif Afzal[1][0000−0003−0611−2655]

[1] School of Innovation, Design, and Engineering, Mälardalen University, Västerås, Sweden
[2] Institute of Software Technology, German Aerospace Center (DLR), Oberpfaffenhofen, Germany
[3] Department of Mathematics and Computer Science, University of Cologne, Cologne, Germany
eduard.paul.enoiu@mdu.se, nasir.mehmood.minhas@mdu.se, michael.felderer@dlr.de, and wasif.afzal@mdu.se

Abstract. Automated test generation is an area that has seen a lot of research and development, resulting in many test automation methods and tools for test design. However, practitioners often face challenges in adopting these tools. This is not only due to the immaturity of some tools but also because of varying perspectives, confusing terminology, and, most importantly, the lack of a clear framework to guide the selection of the most suitable approach for their needs. We propose a taxonomy that characterizes the methods for automated test generation. The taxonomy was constructed using a process that involved analyzing secondary studies on automated test generation and existing taxonomies in the scientific literature. Direct observations and iterative refinements were included, followed by validation through conceptual evaluation and practitioner feedback. The resulting dimensions characterize automated test generation and its use in software testing. The taxonomy is organized into several dimensions: software artifact (i.e., type, notation, interfaces), test generation (i.e., objectives, methods, and monitoring), test execution, and test oracle. We demonstrate the taxonomy's use by applying it to several automated test-generation tools. This paper provides the necessary concepts and a generic process for categorizing and assessing automated test generation approaches.

Keywords: Test automation· automated test generation· taxonomy

1 Introduction

Software testing is a crucial phase of the software development life cycle; it ensures that the system under test meets the specified requirements and, under different circumstances, behaves as expected. Software testing is a repetitive, complex, and time-consuming activity, especially for large-scale systems with

complex requirements [16, 17]. Automated Test Case Generation (ATG) is a solution to address the complexity and time constraints of manual testing [5].

Despite the abundance of research on different testing techniques and tools, the results are difficult to adopt by practitioners [1]. In recent decades, software professionals or researchers have faced problems developing effective, applicable, and practically relevant test generation techniques and tools [6]. With so many approaches in automated test generation, there is a risk of not assessing and adopting these properly in practice, making it harder for practitioners to choose and use these tools. It is difficult for practitioners to adopt these techniques and tools, such as using different terminologies. Among many, one significant reason is the absence of a conceptual model to help practitioners choose a method or tool that fits their context [1].

Many software testing standards, such as ISO/IEC/IEEE 29119 [14], OMG UML Testing Profile [7], and TTCN-3 [27], aim to provide guidelines for the use of test techniques and test automation. However, these standards often lack the depth required to address the wide variety of techniques employed by automated test generation methods. For instance, ISO/IEC/IEEE 29119 has been critiqued for its generality and inability to accommodate automated testing practices' complexity and evolving nature fully [2]. This generality makes it challenging to address the specific needs of various testing approaches, including automated test generation. Similarly, the OMG UML Testing Profile and TTCN-3 have limitations in adapting to the particular needs of diverse test automation techniques [26]. Furthermore, despite the efforts to standardize software testing practices in automated test generation [1, 10, 26, 29], no comprehensive overview includes the many perspectives and techniques of automated test generation. While these taxonomies aim to categorize and organize the different techniques, methods, and approaches, they often focus on specific areas without providing a unified view of automated test generation. This gap highlights the need for a broader framework to integrate automated testing techniques' diverse and complex landscape.

This paper provides this overview as a taxonomy, how it can be used to categorize rather different automated test generation tools, and how to use it to identify the right tools.

The paper is organized as follows: Section 2 summarizes related work, Section 3 describes steps for creating the ATG taxonomy, and Section 4 introduces the generic test generation process. Section 5 details the ATG taxonomy, its application to tools in Section 6, and evaluation results in Section 7. Discussion and limitations in Section 8, and conclusions in Section 9.

2 Related Work

Various authors have proposed software testing taxonomies across different testing categories, such as regression testing [1, 9], automated software testing [5], risk-based testing [10, 13], and model-based testing [16, 22, 26, 29]. Table 1 shows a comparative summary of these taxonomies that aim to categorize and organize

Table 1. Comparison of Testing Taxonomies Focused on Test Creation and Selection

Author(s)	Testing Category	Focus/Criteria	Key Contributions
Bin et al. [1]	Regression Testing	Industry Relevance	Proposed a taxonomy for regression testing.
Minhas et al. [16]	Model-Based Testing	Model, Test Type, Capabilities	Categorized test case generation techniques, providing an inventory and documentation of capabilities and limitations.
Felderer et al. (2016) [11]	Security Testing	Security Properties, Coverage Criteria	Developed a taxonomy to show insights into model-based security testing.
Felderer et al. (2014) [10]	Risk-Based Testing	Risk Assessment	Developed a taxonomy to understand, assess, and compare risk-based testing, applied in various phases of testing.
Saeed et al. [22]	Search-Based Techniques	Experimental Applications	Developed a taxonomy to categorize experimental applications of search-based techniques.
Zander et al. [29]	Model-Based Testing	Test Generation, Execution, Evaluation	Expanded on earlier taxonomies, adding a category for test generation tailored to embedded systems.
Ramli et al. [21]	Combinatorial Testing	Algorithmic Approaches	Reviewed algorithms and tools for combinatorial testing contributing to a structured understanding of the field.

various techniques, methods, and approaches in software testing to understand better and apply them in practice. Table 1 summarizes key taxonomies across different categories, highlighting their focus areas and contributions.

For example, Bin et al. [1] conducted reference searches and consulted industry experts to propose regression testing (RT) taxonomies that encompass aspects of industry relevance regarding regression testing techniques. They mapped 26 industry-relevant regression testing techniques to the proposed RT taxonomies. By categorizing techniques based on industry relevance, the authors aim to provide a structured framework to help practitioners select appropriate regression testing techniques for their specific contexts. Minhas et al. [16] categorized model-based test case generation techniques based on the underlying model used, test type, capabilities, and industry relevance. The authors aimed to provide practitioners with an inventory of results and documentation of the capabilities and limitations of these approaches.

Felderer et al. [11] proposed a taxonomy for model-based security testing methods, which includes filter and evidence criteria. The authors analyzed 119 publications on model-based security testing based on these criteria, offering insights into the current landscape. The analysis covers aspects such as security properties, coverage criteria, and the practicality and cost-effectiveness of such testing. In another work, Felderer et al. (2014) introduced a taxonomy of risk-based testing, which provides a framework for understanding, categorizing, assessing, and comparing risk-based testing approaches. The authors aligned the taxonomy with considering risks in all phases of the testing process and it has been applied to selected work on risk-based testing.

Saeed et al. [22] performed a systematic literature review that included 72 experimental studies on search-based techniques (SBTs) for model-based testing (MBT). Based on the results of the review, the authors developed a taxonomy to categorize the current experimental applications of SBTs. Saeed et al. concluded that this taxonomy would help researchers explore existing research efforts and identify limitations requiring further investigation.

Zander et al. [29] expanded on the model-based testing taxonomy introduced in Zander et al. [28], which itself built upon the taxonomy presented in Pretschner et al. [19]. In addition to the original categories of classes model, test generation, test execution, and test evaluation outlined in Zander et al. [28], the authors included a new category for the test generation class called "result of the generation´´.

Ramli et al. [21] presented a review of algorithms and tools on combinatorial testing proposed from 2010 to 2017. All these studies are intended to contribute to the structured understanding of the software testing landscape. Collectively, these works enrich the taxonomy landscape in software testing.

Although these taxonomies classify and organize various techniques, methods, and approaches in automated test generation, they focus on particular aspects, lacking a comprehensive, unified perspective on automated test generation. Table 1 shows a comparative summary of these taxonomies, demonstrating their diverse points and the existing gap in providing one view of the work done to categorize automated test generation and selection. This emphasizes the need for a more extensive taxonomy to integrate the varied landscape of automated test generation techniques.

3 Taxonomy Development Process

The automated test generation taxonomy development followed a structured and iterative approach, as shown in Figure 1, inspired by the methods used in [1, 15]. The process involves multiple stages, each contributing to refining the taxonomy and ensuring its applicability to real-world automated test generation tools.

Using Secondary Studies and Personal Experience. The taxonomy development process is based on the guidelines proposed by Ralph [20], using secondary studies and personal experiences as strategies for taxonomy generation (Step 1 in Figure 1). Since we were not limited to techniques generating test data fully automated, we focused on the only secondary study [5] on automated test generation techniques regardless of their input software artifacts. This study still remains a good reference in automated test generation because the techniques it categorizes are still fundamental to current practices. Their orchestrated survey of methodologies provides a broad study on automated test generation that continues to be relevant. Based on this study, we chose to scope our focus to the following test generation categories: structural testing, model-based testing, combinatorial testing, random testing, and search-based testing. We reflected on our own experience in automated test generation and used existing taxonomies for model-based testing [11, 26]. While there are common characteristics among

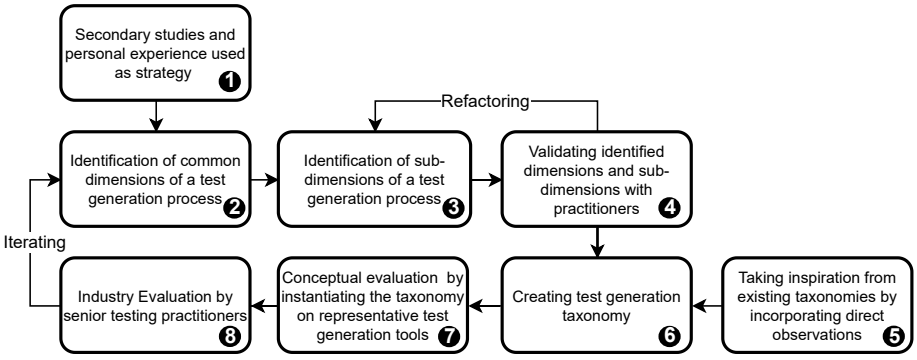

Fig. 1. Steps to create the automated test generation taxonomy

all these existing studies, we argue that these are insufficient to cover *generic* aspects of automated test generation regardless of their input software artifact and inner workings. We employed direct observation and reviewed these existing taxonomies and our chosen secondary study [5] to collect data and take notes on how a generic test generation process would look like (shown in Section 4).

Identification of Common Dimensions of Test Generation. We analyzed this process and built the taxonomy in step 2 (in Figure 1). We evolved the taxonomy by reviewing existing studies (mentioned in Section 5), identifying each dimension common to all test generation techniques.

Identification of Sub-Dimensions of Test Generation. After identifying the common dimensions, we moved on to identifying the sub-dimensions within each category (Step 3 in Figure 1). For example, within a top dimension, sub-dimensions were defined. This level of detail ensured the taxonomy could cover a wide range of test generation techniques and provide specific categories for each.

Validating Dimensions with Practitioners. To ensure the identified dimensions and sub-dimensions were relevant to real-world automated test generation tools, they were validated by practitioners in the field of software testing during a workshop with four engineers (Step 4 in Figure 1). This validation process involved obtaining qualitative feedback from these engineers who have experience with automated test generation. The identified dimensions and sub-dimensions were refactored based on practitioner feedback to address any gaps or inconsistencies. This allowed for fine-tuning the taxonomy.

Creating the Test Generation Taxonomy. After the validation and refactoring stages, the taxonomy for test generation was finalized for evaluation (Step 6 in Figure 1). The result is a categorization that captures the essential characteristics of test generation approaches, offering a unified view of the landscape, which can be applied to tools and methods across the field. This step involved drawing

inspiration from pre-existing taxonomies in software testing and automated test generation (Step 5 in Figure 1).

Conceptual Evaluation on Representative Tools. In step 7 in Figure 1, we performed an initial conceptual evaluation by instantiating the taxonomy on representative test generation tools covering all of the methodology categories outlined by Anand et al. [5].

Industry Evaluation by Senior Testing Practitioners. The final step in the process (step 8 in Figure 1) was an industry evaluation conducted by eight testing practitioners. This evaluation aimed to assess the practicality of the taxonomy when choosing a test generation tool. The feedback from this evaluation was used to further iterate on the taxonomy, ensuring it met the needs of both researchers and practitioners.

4 A Generic Test Generation Process

The process of automated test generation aims to find suitable test cases using a description of the test objectives that guide towards a desirable property. These test cases could contain parameters to start the software, a sequence of steps and inputs and the timing when these steps should be supplied. In some cases, test cases might need to contain other information for a complete execution and evaluation of the system under test. In Figure 2, a typical setting for automated test generation is identified based on the testing process outlined by Utting et al. [26] and extended using the methodology categories for automated test generation [5]. A generic process of automated test generation proceeds as follows: (Step 1) A software artifact is used or created to guide the test generation. It is either a specification of what the System-under-Test (SUT) should do or the actual code of the SUT in different forms (e.g., source, executable code). (Step 2) A test objective and method formally encodes the test criteria and describes how the test generator should choose the resulting tests. This can relate to the structure of the software artifact (e.g., code or model coverage), random test goals, or fault-based objectives. (Step 3+4) A test suite is generated by running the software over many possible executions using a specific method. Each method needs to monitor if the test objectives are met, which can be achieved during test generation or just for test execution and evaluation of the

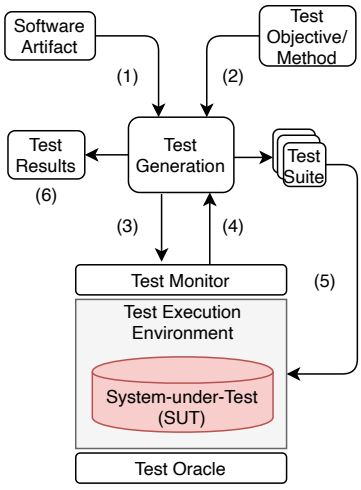

Fig. 2. A Test Generation Process.

test results. This can be invasive (e.g., at code level) or non-invasive by focusing on the available external interfaces. (Step 5) Once Steps 1 to 4 are completed, a test suite is executed by running the software online or offline. (Step 6) The test evaluation compares the actual outputs of the SUT with the expected outputs provided by the oracle so that the test results are generated.

5 The Automated Test Generation Taxonomy

Automated test generation approaches can be quite different, but all of them have common underlying dimensions that can be quite helpful when adopting automated test generation in a certain software development project. Given the generic test generation process shown in Figure 2, we identified several dimensions corresponding to these steps (i.e., Software Artifact (Step 1), Test Generation (Steps 2, 3 and 4), Test Execution and Evaluation (Step 5 and 6)). Even though there can be other steps in software testing used in practice, we argue that the identified steps are most commonly conducted when using automated test generation.

This taxonomy of automated test generation approaches has five categories shown in Figure 3. This gives an overview of the taxonomy where the tree leaves indicate options that are not incompatible (for example, some approaches may use more than one generation objectives). The initial taxonomy, which outlined dimensions such as software artifact, test generation methods, test execution, and evaluation, was validated by four experienced professionals in software quality assurance and test automation with careers ranging from 3 to over 20 years in roles such as software tester, test architect, QA coordinator, and consultant. Each practitioner provided insights into the use of automated test generation tools and the applicability of the taxonomy when choosing a tool. One significant addition was the dimension Test Oracle. Practitioners emphasized the need to clearly differentiate between the types of test objectives and test oracles. The taxonomy was updated to provide more granularity, separating oracles from test objectives.

5.1 Software Artifact

Creating the software artifact is reflected by the three dimensions within the software artifact category: type, notation, and interface. These are used in the software artifact type categorization of Pfeiffer [18] and the model-based testing taxonomy [26]. The first dimension refers to the type of software artifact used as input to the test generation process. Software is multidimensional and consists of a variety of artifacts related to data (i.e., test results[4], graphical user interface, database), code (source or executable) and documentation (i.e., requirement, design and test specifications) [18]. The second dimension refers to the notation style used to describe the software artifact. Many different notations have been used for representing the expected or the actual behavior of software

[4] Test results refer to the data as execution logs and evaluation outcomes.

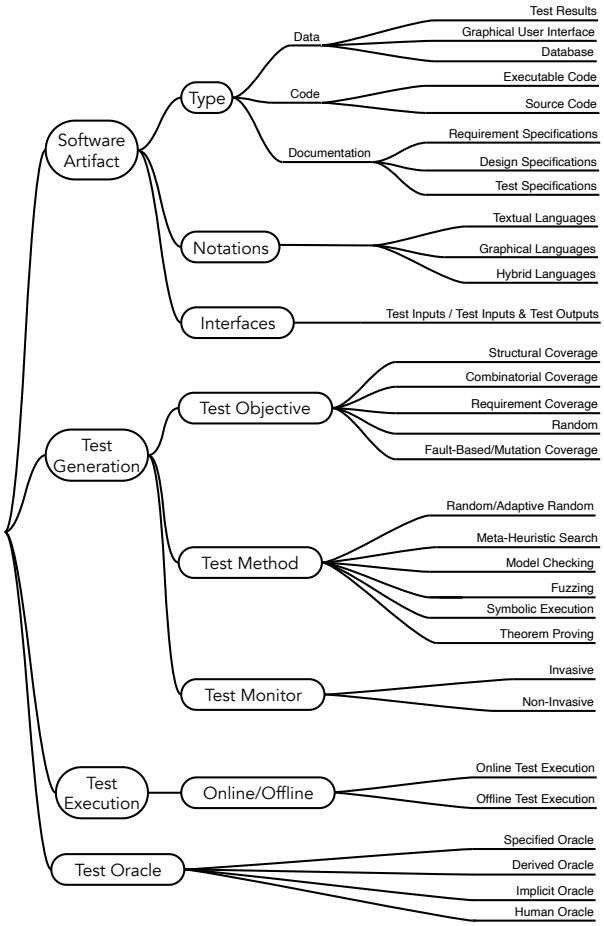

Fig. 3. Overview of the Test Generation Taxonomy.

artifacts. To differentiate between language styles [25], we group them into textual, graphical and hybrid languages. The last dimension relates to the input-output interfaces [26], which answers the following question: does the software artifact specify only the test inputs (or the sequence of test inputs), or does it specify the input-output behavior of the SUT? This is an "x/y" tree leaf alternative that indicates incompatible alternatives. For example, combinatorial test generation tools will be used to represent test inputs and do not specify the expected outputs.

5.2 Test Generation

This test generation category includes three dimensions defining test objectives for generating and monitoring test cases. Since it partially aligns with test generation dimensions in model-based testing taxonomies [12, 26], we use their categorization

of test objective criteria and monitors, which include structural, combinatorial, requirement, random, and fault-based/mutation coverage[5]. We classify test generation methods as graph search algorithms, meta-heuristic techniques, model checking, symbolic execution, theorem proving, and fuzzing. The test monitor dimension evaluates whether test objectives are met, referencing model-based testing processes [26] and automated test generation needs [5]. It includes two types: invasive and non-invasive. Invasive monitoring uses instrumentation at code and interface levels, and non-invasive monitoring is suitable for cases like embedded systems where invasive methods may alter behavior and obtain coverage information via static analysis or external interfaces. For example, code coverage data may be derived directly from source code without altering execution.

5.3 Test Execution

The test execution dimension defines how test case generation relates to their execution, a concept also employed in the model-based testing taxonomy [26]. Test execution can occur offline, where test cases are generated and executed later, or online, where test cases are dynamically generated and executed during runtime. Some tools, such as GraphWalker, support both modes, showing flexibility to adapt to specific testing scenarios.

5.4 Test Oracle

The last category relates to how automated test generation tools determine whether a given test case is acceptable. This should not be confused with the abstract information in the requirement and design specification. An oracle implements a specification and is used to judge the correctness of the generated test data. We use the categorization of test oracles of Barr et al. [8] in which the test generation tools can use specified oracles (formally specified models), derived oracles (derived from the software artifacts or system executions, e.g., metamorphic testing), implicit oracles (by relying on general, implicit knowledge to distinguish between a system's correct and incorrect behavior), and human oracles when a human being is checking the results of the generated test cases. For example, metamorphic testing [24] derives oracles based on metamorphic relations that must hold across different software executions.

6 Tool Classification via Test Generation Taxonomy

This section categorizes some typical automated test generation tools about the automated test generation taxonomy presented before. We show the characteristics of these tools and how the taxonomy can be used to differentiate between different automated test generation approaches by covering all the categories of test

[5] The objectives, based on a model-based testing taxonomy [26], align with automated test generation tools, focusing on specific criteria unlike broader classifications [4].

generation methodologies outlined in Anand et al. [5]. The results of applying the taxonomy on four tools (i.e., ACTS, Sapienz, SLDV, GraphWalker, and Randoop) are shown in Table 2.

Table 2. Results of applying the taxonomy on automated test generation tools.

Dimensions/Tools	***ACTS***	***Sapienz***	***SLDV***	***Randoop***	***GraphWalker***
Software Artifact Type	Test Specification	Source/Exec. Code	Design Specification, Source Code	Source Code	Requirement Specification, Test Specification, Design Specification
Software Artifact Notation	Graphical, Textual	Textual	Graphical, Textual	Textual	Graphical
Software Artifact Interface	Inputs	Inputs & Outputs	Inputs	Inputs	Inputs & Outputs
Test Generation Objective	Combinatorial	Structural	Structural, Requirement	Random	Structural, Requirement, Random
Test Generation Method	Model Checking	Meta-Heuristic Search	Theorem Proving, Model Checking	Random	Graph-Based
Test Generation Monitor	Non-Invasive	Invasive/Non-Invasive	Invasive	Non-Invasive	Non-Invasive
Test Execution	Offline	Offline/Online	Offline	Offline	Offline/Online
Test Oracle	Human	Implicit	Specified, Derived, Implicit, Human	Implicit, Specified	Specified, Derived

6.1 ACTS

ACTS [6] is a tool designed to generate test input that ensures t-way coverage of input parameters with support for constraints. ACTS works with test specifications, focusing on input parameter combinations. The tool operates with input-only interfaces, meaning test inputs are generated without expected output specifications. Driven by the goal of combinatorial coverage, ACTS systematically explores input combinations through model-checking. It follows a noninvasive approach for test monitoring; it does not require instrumentation or alteration to the system under test. Tests are generated offline, requiring manual or semi-automated execution, and based on the provided input combinations, ACTS typically relies on a human oracle to validate test results.

6.2 Sapienz

Sapienz[7] is a test generation tool using a search-based evolutionary algorithm, guided by a fitness function. Sapienz supports both offline and online test ex-

[6] `https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software`
[7] `https://github.com/Rhapsod/sapienz`

ecution with an implicit test oracle. It can generate tests dynamically during execution (online) or analyze tests post-execution (offline). Sapienz works with source and binary code, specifically Android Package (APK) files. It builds its internal model of the system through textual and graphical UI interactions, using both input-output interfaces to handle user actions and system responses. For test generation, Sapienz focuses on structural coverage, ensuring that the code paths are explored. It employs a meta-heuristic search, with invasive and non-invasive monitoring at the UI and method levels for coverage tracking.

6.3 SimuLink Design Verifier

Simulink Design Verifier (SLDV [8]) is the de-facto standard for validating and verifying Simulink-described systems. SLDV works with Simulink and C/C++ specifications, supporting graphical and textual notations via input-output interfaces to ensure coverage objectives like statement coverage and requirements. SLDV's test generation leverages theorem-proving and model-checking with an invasive monitoring approach that instruments models or code for coverage tracking. SLDV supports both offline and online execution and runs tests in environments such as X-in-the-Loop (XIL). It supports specified, derived, and implicit oracles for validation through formal specifications, inferred properties, or system behavior monitoring.

6.4 Randoop

Randoop[9] is a feedback-directed random test generation tool. It inputs a set of programs and generates random test cases and assertions. Randoop works with source and binary code and targets Java and .NET environments. It generates a sequence of method calls using textual notations at the method level. It also uses input-output interfaces, where inputs are method invocations and outputs are system states or exceptions. Using random and structural coverage, Randoop generates method call sequences through feedback-directed random testing, using prior successes for future cases. Its non-invasive monitoring requires no code instrumentation. Randoop executes tests online, dynamically generating and running tests with oracles that detect exceptions and allow user-specified contracts to define expected behavior.

6.5 GraphWalker

GraphWalker[10] is a model-based testing tool that reads models as directed graphs (finite state machines) and generates test paths based on user-defined generator rules (e.g., structural, requirement, or random objectives) and stop conditions. GraphWalker models the system using requirement, test, and design

[8] `https://www.mathworks.com/products/simulink-design-verifier.html`
[9] `https://randoop.github.io/`
[10] `https://graphwalker.github.io/`

specifications, employing graphical and textual notations to depict states and transitions. Operating with input-output interfaces, it traverses the model to define paths representing action-state sequences. For test generation, GraphWalker focuses on structural and requirement-based coverage. It can also produce random paths based on the specified objective. It identifies paths through the system model using graph traversal algorithms like A star. Its non-invasive approach requires no system instrumentation. GraphWalker supports online and offline test execution, allowing for real-time path generation or pre-execution analysis. Test oracles in GraphWalker include specified oracles (for predefined behaviors) and derived oracles, inferring correct behavior based on state transitions during execution.

7 Evaluation of the Taxonomy

The participants in this static evaluation have extensive software industry experience, with backgrounds spanning 10 to over 20 years in roles including developer, tester, and test manager, specializing in embedded systems, automated testing, and sectors such as railway and telecommunications. Based on this exploratory evaluation conducted by the eight testing practitioners, feedback revealed insights into the practical application of the developed taxonomy for selecting automated test generation tools. Several themes were identified through thematic analysis of notes taken during the workshop, where the taxonomy was presented, applied, and discussed. Practitioners agreed that the taxonomy offers a structured view of various test generation tools **(1)**, particularly highlighting relevant characteristics like monitoring techniques, test generation objectives, artifact type, and test oracles. These features were beneficial for understanding test generation functionality and selecting tools suited to specific software testing contexts and different project scopes. Practitioners noted that the taxonomy's dimensions were broad **(2)**, especially for categorizing tools based on the software artifact dimension. A recurring observation was that specific terms within the taxonomy, such as invasive/non-invasive monitoring or implicit oracle, were challenging due to subtle differences in industry versus academic contexts **(3)**. Practitioners recommended more precise definitions to bridge this gap, particularly for terms with varying practical applications. The taxonomy was found well-suited for classifying tools like Pynguin and EvoMaster **(4)**. Practitioners emphasized the importance of understanding specific test generation characteristics, such as fault types detected, practical coverage levels, and resource requirements **(5)**. Expanding the taxonomy to include dimensions related to effectiveness and cost could enhance its practical relevance.

8 Discussions and Limitations

A practitioner interested in identifying a certain tool for generating test cases in a certain project can use this taxonomy to show and compare the characteristics of these approaches to target various application domains and testing aspects. The

results in Table 2 indicate that the provided taxonomy can be useful for choosing between different automated test generation tools. We see a variety of ways in which this taxonomy can be used. Depending on the individual characteristics of a project, engineers can determine the automated test generation tool of interest. Based on these selections, relevant attributes can be analyzed in terms of the software artifact, test generation, test execution, and test oracle categories. More work is needed to refine and evaluate this taxonomy by applying it to more tools and industrial settings. The result of using this taxonomy can be enhanced by coupling it with a measurement framework for test efficiency and effectiveness analysis as mentioned also by the practitioners in the industrial evaluation.

Generative AI has recently been applied to test generation and augmentation [3, 23]. These tools demonstrate how LLMs can enhance automated test generation by iterative refining and creating test cases to cover previously untested software. In our taxonomy, Generative AI-based tools would fit well under the Test Generation Method category, with a potential new subcategory for LLM-based approaches that leverage iterative test generation capabilities. We have not yet included this, as it is a rather recent development requiring more established research before proper integration.

Based on secondary studies and personal experience, the taxonomy development may introduce bias and limit comprehensiveness. Built on a secondary study [5], it systematically classifies current software testing methods yet risks omitting emerging techniques. Practitioner input was gathered in two phases to address potential bias and ensure relevance. Additionally, the taxonomy was mapped to five test generation tools to showcase relevance, though this may not include all relevant tools. A further limitation is that practitioners evaluated the taxonomy statically, which may affect assessing its practical applicability; a larger, more varied group is needed for broader validation.

9 Conclusions and Reflections

The generic test generation process and the presented taxonomy help to clarify the main characteristics of the automated test generation area and show the possible alternatives and directions. This information can be used to classify test generation tools and to help testers or users of these tools understand which approaches fit their specific needs most closely. Automated test case generation has matured, and large-scale deployments of this technology are underway in many industries. Given the variety of approaches available, a taxonomy like this could be valuable for researchers and practitioners. Applying it in research and practice may lead to ongoing validation and refinement.

10 Acknowledgements

This work has received funding from the MATISSE project, an EU-funded initiative under Horizon Europe GA no. 101056674, and support from the SmartDelta project funded by Vinnova and the Software Center project.

References

[1] bin Ali, N., Engström, E., Taromirad, M., Mousavi, M.R., Minhas, N.M., Helgesson, D., Kunze, S., Varshosaz, M.: On the search for industry-relevant regression testing research. Empirical Software Engineering pp. 1–36 (2019)
[2] Ali, S., Yue, T.: Formalizing the iso/iec/ieee 29119 software testing standard. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 396–405. IEEE (2015)
[3] Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., Marginean, A., Sengupta, S., Wang, E.: Automated unit test improvement using large language models at meta. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering. pp. 185–196 (2024)
[4] Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2017)
[5] Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8), 1978–2001 (2013)
[6] Arcuri, A.: An experience report on applying software testing academic results in industry: we need usable automated test generation. Empirical Software Engineering **23**(4), 1959–1981 (2018)
[7] Baker, P.: Model-Driven Testing: Using the UML Testing Profile. Springer-Verlag, Berlin, Heidelberg (2009)
[8] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE transactions on software engineering **41**(5), 507–525 (2014)
[9] Felderer, M., Fourneret, E.: A systematic classification of security regression testing approaches. International Journal on Software Tools for Technology Transfer **17**, 305–319 (2015)
[10] Felderer, M., Schieferdecker, I.: A taxonomy of risk-based testing. International Journal on Software Tools for Technology Transfer **16**, 559–568 (2014)
[11] Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschner, A.: Model-based security testing: a taxonomy and systematic classification. Software Testing, Verification and Reliability **26**(2), 119–148 (2016)
[12] Fraser, G., Rojas, J.M.: Software testing. In: Handbook of Software Engineering, pp. 123–192. Springer (2019)
[13] Großmann, J., Felderer, M., Viehmann, J., Schieferdecker, I.: A taxonomy to assess and tailor risk-based testing in recent testing standards. IEEE Software **37**(1), 40–49 (2019)
[14] ISO/IEC/IEEE: Iso/iec/ieee international standard - software and systems engineering –software testing –part 1:general concepts. ISO/IEC/IEEE 29119-1:2022(E) pp. 1–60 (2022). https://doi.org/10.1109/IEEESTD.2022.9698145

[15] Minhas, N.M., Börstler, J., Petersen, K.: Checklists to support decision-making in regression testing. Journal of Systems and Software **202**, 111697 (2023)
[16] Minhas, N.M., Masood, S., Petersen, K., Nadeem, A.: A systematic mapping of test case generation techniques using uml interaction diagrams. Journal of Software: Evolution and Process **32**(6), e2235 (2020)
[17] Minhas, N.M., Petersen, K., Börstler, J., Wnuk, K.: Regression testing for large-scale embedded software development – exploring the state of practice. Information and Software Technology **120**, 106254 (2020)
[18] Pfeiffer, R.H.: What constitutes software? an empirical, descriptive study of artifacts. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 481–491 (2020)
[19] Pretschner, A., Utting, M., Legeard, B.: A taxonomy of model-based testing. Department of Computer Science, University of Waikato, Tech. Rep (2006)
[20] Ralph, P.: Toward methodological guidelines for process theories and taxonomies in software engineering. IEEE Transactions on Software Engineering **45**(7), 712–735 (2018)
[21] Ramli, N., Othman, R.R., Khalib, Z.I.A., Jusoh, M.: A review on recent t-way combinatorial testing strategy. In: MATEC Web of Conferences. vol. 140, p. 01016. EDP Sciences (2017)
[22] Saeed, A., Ab Hamid, S.H., Mustafa, M.B.: The experimental applications of search-based techniques for model-based testing: Taxonomy and systematic literature review. Applied Soft Computing **49**, 1094–1117 (2016)
[23] Schäfer, M., Nadi, S., Eghbali, A., Tip, F.: An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering (2023)
[24] Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortés, A.: A survey on metamorphic testing. IEEE Transactions on software engineering **42**(9), 805–824 (2016)
[25] Tse, T., Pong, L.: An examination of requirements specification languages. The Computer Journal **34**(2), 143–152 (1991)
[26] Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software testing, verification and reliability **22**(5), 297–312 (2012)
[27] Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., Schulz, S.: An introduction to TTCN-3. John Wiley & Sons (2011)
[28] Zander, J., Schieferdecker, I.: Model-based testing of embedded systems exemplified for the automotive domain. In: Behavioral modeling for embedded systems and technologies: Applications for design and implementation, pp. 377–413. IGI Global (2010)
[29] Zander, J., Schieferdecker, I., Mosterman, P.J.: A taxonomy of model-based testing for embedded systems from multiple industry domains. (2011)