Who Tests the Tests? Model-Based Validation of AI-Generated Test Cases

Eduard Paul Enoiu
Department of Computer Science and Engineering, Mälardalen University, Västerås, Sweden

Abstract

AI agents can increasingly generate executable tests from code, requirements, and human-written natural-language prompts, but the resulting tests often remain difficult to trust. They may compile and run, yet remain arbitrary or inconsistent with the intended system behavior. This paper proposes using Model-Based Testing (MBT) to check and classify AI-generated tests. We treat these generated software artifacts as candidate tests that must be checked against an MBT model. We define checks for the model path, guards, inputs, adequacy, and MBT logs. One can then classify candidate tests as model-backed, model-extending, or model-inconsistent. The contribution is a conceptual framework for making AI-assisted test generation more reviewable and reusable in quality assurance.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases model-based testing, AI-based test generation, model coverage, test evaluation

Funding *Eduard Paul Enoiu* : This work was supported by Software Center project 68 (TRACE), Vinnova-funded ITEA projects MONA LISA and MATISSE (101140216), and the AI and Society Fellowship.

1 Introduction

AI-assisted testing [6] is changing the cost of producing tests and is increasingly capable of generating abstract and executable tests from source code, documentation, requirements, and natural-language prompts. Recent studies [11, 15, 1] show that such systems can produce tests that achieve high test coverage and fault-detection scores. At the same time, these studies show that generated tests still suffer from correctness issues, execution failures, and assertion errors. This raises a practical question: if AI can now generate many candidate tests, how can Model-Based Testing (MBT) [13] help improve the checking of such tests?

Instead of using MBT only as another generator, we use it as a checking step for AI-assisted software testing. The idea is to treat each generated test as a candidate that must be interpreted against an explicit behavioral model. The assumption is that a test is better supported by evidence when it follows and is consistent with a valid model, satisfies a meaningful adequacy criterion, and is supported by evidence from MBT execution logs.

We present a conceptual workflow for AI-generated tests, along with an initial classification scheme that shows how MBT can accept, reject, or trigger revisions of generated test candidates. The idea is to help teams use AI to produce tests without compromising consistency with the intended system behavior as understood when creating an MBT model.

2 Motivation and related work

We build on the established view of MBT [13] and model testing [2] as a test design and execution approach in which models of the system or its environment are used to derive,

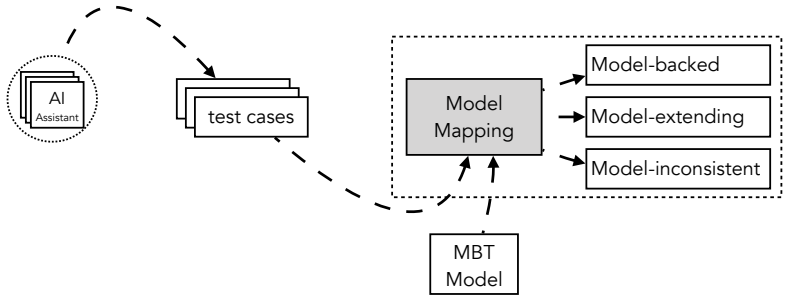

Figure 1 AI generates candidate tests; MBT checks and classifies them.

Class	MBT interpretation	Action
MODEL-BACKED	The test maps to an existing model path, respects guards, and satisfies an adequacy criterion.	Accept and reuse.
MODEL-EXTENDING	The test appears relevant but cannot be mapped to the current model, suggesting missing behavior.	Review and evolve.
MODEL-INCONSISTENT	The test contradicts the model, e.g., invalid transitions, violated guards, unsupported outputs, or wrong expected states.	Reject and revise.

Table 1 Classification of AI-generated tests after MBT checking.

execute, and evaluate tests.

Recently, Mondal et al. [11] showed that LLMs can support MBT by helping build models from natural-language specifications. Recent model-driven approaches [1] have begun using LLMs for test scenario generation, demonstrating that models can guide the production of AI-generated tests. The growing body of work on LLMs for software testing [14] shows that AI-assisted testing is becoming a major research direction, but also motivates the need for systematic validation of generated test artifacts. Our work takes a complementary approach, using MBT to validate and classify tests generated by AI agents.

Other prior work shows that LLMs can improve automated test generation. For example, CodaMOSA [9] uses LLMs to help search-based testing escape coverage plateaus, while empirical work [15] on LLM-based software testing reports that passing generated tests can achieve coverage and readability comparable to manually written tests. However, there is a gap between generating and trusting the generated tests. Such generated tests can still suffer from correctness issues, including execution failures, mostly due to incorrect assertions. Recent oracle-generation studies [10] support the claim that LLMs can generate useful oracles. Others have proposed a model-driven testing framework to ensure LLM quality [3], in which models guide test case generation. These findings suggest a practical view for engineers and researchers. AI-generated tests can be treated as test candidates rather than as direct, usable assets, and MBT can be used to evaluate them.

Existing work has predominantly used model analysis to generate, optimize or select tests. We propose that a test model becomes an independent checking artifact. This shifts model coverage from a primary generation-stopping criterion to one source of evidence about the quality of an AI-generated test suite.

3 Checking AI-generated tests using MBT

We propose a workflow applicable to AI-generated candidate tests (as shown in Figure 1). The proposed model mapping checks if the test case candidates correspond to a valid path in the model. In addition, one would need to check if the chosen inputs respect the model's guards. Also, one would need to check whether the expected result is justified by the model. The adequacy check should verify that the test covers some model elements required by the model. Finally, an MBT log check should verify that model executions produce evidence consistent with the stated test generation goals. MBT execution and model testing are especially important because they link this workflow to evidence of how the model executes. As a vision paper, we focus on the conceptual workflow rather than a tool implementation or empirical evaluation. Earlier work on runtime verification and passive testing [8] shows that traces can be meaningfully checked against specifications and models.

In a graph-based MBT setting [12], for example, the actions and reactions in an AI-generated test can be mapped to a sequence of model elements. The test is checked against the model to determine if its sequence of actions is possible. This view is related to earlier uses of model-based testing and model checking for test generation and analysis [5]. For example, a model can be dynamically monitored, and then used to check whether the proposed test is behaviorally consistent, incomplete with respect to the model, or inconsistent with the modeled behavior.

The model mapping is relative to the model type and a mapping from test actions, concrete inputs and assertions to model elements. Behavioral model mapping starts from a specified initial state and input values with the model replay recording the traversed elements. A failed or inconsistent mapping should be referred for review. Input validity, behavioral consistency, oracle support and test suite adequacy should be recorded since a redundant test case may still be backed by a model. On the other hand, a combinatorial model cannot be used directly for oracle correctness.

4 Classifying AI-generated tests

The outcome of checking is a classification (as shown in Figure 1 and detailed in Table 1). A test is MODEL-BACKED when it maps to an existing model path supported by the MBT execution, satisfies a meaningful adequacy criterion, and is consistent with the MBT logs. A test is MODEL-EXTENDING when it appears valuable but does not fit the current model, suggesting that the model may be incomplete. A test is MODEL-INCONSISTENT when it contradicts the modeled behavior, for example, by violating guards, expecting outputs outside a certain range, or asserting state changes that the model does not allow. Table 1 shows the possible outcomes of the MBT checking step and shows how each class of AI-generated test is interpreted and acted upon.

Consider a controller with Locked and Unlocked states and an unlock transition enabled only when a signal is valid. Starting from Locked, a candidate that provides a valid signal and expects Unlocked is model-backed when its inputs and assertions match this transition. A candidate that expects this transition when the signal is invalid is model-inconsistent. When another emergency signal that is supported by requirements is absent from the model, the test case is model-extending and requires review before acceptance.

5 Proposed Tool Support

For model-based validation of AI-generated test cases, existing generation and execution tools can be used. For example. SEAFOX provides a starting point for combinatorial models [4]. A combinatorial model specifies input parameters, their finite value domains or equivalence partitions and constraints on certain combinations. Fifo et al. [7] measured combinatorial coverage of manually created industrial tests. Similarly, for AI-generated tests, SEAFOX can identify which interactions are covered, which are missing, and which proposed combinations fall outside the modeled input space. Different test suites can cover the same interactions, so acceptance should depend on the input model and constraints. The inputs selected by the AI are mapped to the combinatorial model, checked for valid parameter values and constraints, and analyzed to determine which interactions they cover. The resulting coverage information can reveal missing combinations, redundant tests and candidates that use unsupported or infeasible inputs. In this way, combinatorial modeling provides an adequacy criterion against which AI-generated tests can be evaluated.

Another tool, TIGER, provides behavioral testing for industrial control software and uses GraphWalker to generate abstract tests and mapping rules, then concretizes them into executable scripts [16]. In our approach, we combine these capabilities to implement the checks in Section 3. For this reachability analysis, the implementation builds on the GraphWalker and UPPAAL integration already discussed in [12]. The combined process uses requirements to guide both model analyses. Requirements are first interpreted to construct an MBT model, such as a finite-state model with states, transitions, guards, actions and variables. This test model is then transformed into an analysis model that can be checked against formalized requirements. This allows one to use the generated test cases to refine the original model before proceeding with model-based validation. Test criteria such as model coverage or random exploration then guide test case evaluation and repeated model executions produce the model mapping. Overall, our approach, implemented with a Graphwalker model, integrates requirements, behavioral modeling, formal model analysis, and model refinement, and can guide the evaluation of automated test generation within a common workflow.

Building on the combind model-checking and test-generation approach of Enoiu et al. [5], we propose checking externally generated test candidates against a behavioral model. Instead of asking the model checker to produce a new test, one would need to map an AI-generated candidate onto the behavioral model and check if its sequence of actions is feasible, its inputs satisfy guards and constraints and its expected outputs are supported. Model checking can additionally verify relevant properties and provide execution traces as evidence for the assessment. The resulting evidence can then be used to classify the candidate as model-backed, model-extending, or model-inconsistent, while model coverage indicates how much useful behavior the accepted AI-generated tests exercise.

Taken together, prior work on GraphWalker and UPPAAL model analysis [12], model checking-based test generation [5], and TIGER test suite analysis [16] provide the building blocks needed to apply this approach to different model-based methods.

6 Proposed Evaluation

We propose an evaluation that examines if the proposed model-based evaluation approach can distinguish effective and efficient AI-generated tests from tests that are unsupported by the behavioral model. A suitable experiment would use several representative systems with

available requirements, an MBT model and executable implementations. For each system, multiple LLMs or prompting configurations would generate candidate test cases from the same requirements. Each candidate would then be mapped to the MBT model and checked for path feasibility (for a behavioral model), guard satisfaction, input validity, expected outputs and model coverage. Initially, human reviewers should verify the classification of each test as model-backed, model-extending or model-inconsistent, against which the automated classification is compared. The experiment could also compare the complete framework with other baselines such as LLM-based reviews.

In addition, one should evaluate if the tests remain effective as a test suite. For each generated test suite, we propose to measure model coverage, requirement coverage, combinatorial input coverage, redundancy, execution cost and injected or naturally occurring fault detection effectiveness before and after evaluation. Fault detection could be assessed using independently seeded faults or mutation analysis, following the general experimental pattern used in earlier model-checking-based test-generation studies, where generated suites were executed on original and faulty program versions and a fault was considered detected when their observed outputs differed. It is particularly important to use repeated LLM generations, as in a real testing workflow and to account for variation under a fixed execution and review budget.

The evaluation should separate classification accuracy from test suite effectiveness. For mutation analysis, a fault should be counted as detected only when a test passes the validated implementation and its own oracle exposes the fault in the mutant. In addition, it is important to report the model construction and mapping costs separately from recurring evaluation costs.

7 Conclusion

This paper proposes using MBT as a check for AI-generated tests. We treat these as candidates that must be mapped to an explicit behavioral model. In a practical setting, this means checking whether the candidate tests follow the model execution and contribute to an adequacy criterion, and are consistent with MBT execution logs. The result is a classification of AI-generated tests as model-backed, model-extending, or model-inconsistent. This classification could help engineers decide whether to accept a test, evolve the model, or reject the candidate tests. The broader vision is that AI generates many potential tests, while MBT determines which are behaviorally meaningful and reusable for quality assurance.

Since the framework is conceptual, it has not yet been evaluated, and its usefulness depends on support from MBT and model-testing tools. In addition, human reasoning is still needed to classify tests as model-backed, model-extending, or model-inconsistent, so future work should evaluate this through concrete case studies and practitioner feedback.

References

1 Issam Al-Azzoni, Saqib Iqbal, Taymour Al Ashkar, and Zobia Erum. Integrating model-driven engineering and large language models for test scenario generation for smart contracts. *Information*, 17(1):1, 2025.

2 Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: Model testing of complex software-intensive systems. In *Proceedings - 5th International Workshop on Green and Sustainable Software, GREENS 2016*, Proceedings - International Conference on Software Engineering, pages 789–792. IEEE Computer Society, May 2016. doi:10.1145/2889160.2889212.

193 **3** Jesús Carreño-Bolufer, Giovanni Giachetti, and Oscar Pastor. Towards model-driven testing
194 for assuring the quality of large language models. In *International Conference on Conceptual*
195 *Modeling*, pages 195–209. Springer, 2025.
196 **4** Peter Charbachi, Linus Eklund, and Eduard Enoiu. Can pairwise testing perform comparably to
197 manually handcrafted testing carried out by industrial engineers? In *2017 IEEE International*
198 *Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 92–99.
199 IEEE, 2017.
200 **5** Eduard P Enoiu, Daniel Sundmark, Adnan Čaušević, Robert Feldt, and Paul Pettersson.
201 Mutation-based test generation for plc embedded software using model checking. In *IFIP*
202 *International Conference on Testing Software and Systems*, pages 155–171. Springer, 2016.
203 **6** M. Felderer, Eduard Paul Enoiu, and Sahar Tahvili. Artificial intelligence techniques in system
204 testing. In *Optimising the software development process with artificial intelligence:*, volume
205 Part F1169 of *Natural computing series*, pages 221–240. Springer Science and Business Media
206 Deutschland GmbH, 2023. doi:10.1007/978-981-19-9948-2_8.
207 **7** Miraldi Fifo, Eduard Enoiu, and Wasif Afzal. On measuring combinatorial coverage of manually
208 created test cases for industrial software. In *International Conference on Software Testing,*
209 *Verification and Validation Workshops*, pages 264–267. IEEE, 2019.
210 **8** Daniel Flemström, Henrik Jonsson, Eduard Paul Enoiu, and Wasif Afzal. Industrial scale
211 passive testing with t-ears. In *2021 14th IEEE Conference on Software Testing, Verification*
212 *and Validation (ICST)*, pages 351–361. IEEE, 2021.
213 **9** Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa:
214 Escaping coverage plateaus in test generation with pre-trained large language models. In *2023*
215 *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931.
216 IEEE, 2023.
217 **10** Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D Ernst, and Mauro Pezze.
218 Do llms generate useful test oracles? an empirical study with an unbiased dataset. In *2025*
219 *40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages
220 278–290. IEEE, 2025.
221 **11** Rajdeep Mondal, Rathin Singha, Todd Millstein, George Varghese, Ryan Beckett, and Siva
222 Kesava Reddy Kakarla. Eywa: Automating model based testing using llms. *arXiv preprint*
223 *arXiv:2312.06875*, 2023.
224 **12** Saurabh Tiwari, Kumar Iyer, and Eduard Paul Enoiu. Combining model-based testing and
225 automated analysis of behavioural models using graphwalker and uppaal. In *Asia-Pacific*
226 *Software Engineering Conference (APSEC)*, pages 452–456. IEEE, 2022.
227 **13** Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing
228 approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.
229 **14** Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software
230 testing with large language models: Survey, landscape, and vision. *IEEE Transactions on*
231 *Software Engineering*, 50(4):911–936, 2024.
232 **15** Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling
233 Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on*
234 *Software Engineering*, 1(FSE):1703–1726, 2024.
235 **16** Muhammad Nouman Zafar, Wasif Afzal, Eduard Paul Enoiu, Zulqarnain Haider, and Inderjeet
236 Singh. A model-based test script generation framework and industrial insight. *SN Computer*
237 *Science*, 6(4):294, 2025.