Breaking the Model-Based Testing Barrier: How AI & BP Can Transform Software Testing

Michael Bar-Sinai, Gera Weiss

michael@provengo.tech, geraw@cs.bgu.ac.il

Abstract

Model-Based Testing (MBT) promises rigorous, automated quality assurance but remains underutilized due to the complexity of creating and maintaining the formal models on which it is based. We argue that combining Behavioral Programming (BP) with Generative AI, specifically Large Language Models (LLMs), breaks this barrier. BP enables modular, accessible, scenario-based formal modeling, while LLMs help translate informal descriptions into formal behavior fragments. Through case studies, we show how this combination empowers QA teams to adopt MBT practices without the traditional – and prohibitive - learning curve.

Biography

Michael Bar-Sinai is a software engineer and researcher. Co-Founder and CTO of Provengo Technologies, a startup building development tools based on formal methods. Michael earned a PhD in Software Engineering, studying Behavioral Programming, model-based software, requirement-based QA, and automation. Before launching Provengo, he was a postdoctoral researcher and fellow at Harvard's Institute for Quantitative Social Sciences, worked as a senior software architect, and led consultancy projects across industry, academia, and NGOs.

Gera Weiss is a Professor of Computer Science at Ben-Gurion University. He has led research and development efforts at the intersection of software engineering, formal methods, and artificial intelligence. His recent work focuses on using scenario-based modeling and Behavioral Programming to simplify test automation and Model-Based Testing. Gera collaborates with industry and academia to bring advanced testing techniques into practical software development workflows.

1 Introduction

Software testing and quality assurance (QA) stands at a pivotal crossroads. Traditional test methods—dominated by manual testing and brittle script-based automation—are increasingly insufficient as software systems grow more complex and agile release cycles become shorter. Automated tests frequently become tightly coupled with system implementations, resulting in fragile test suites that break easily with UI or API changes. This makes QA predominantly reactive, labor-intensive, and costly to maintain.

Model-Based Testing (MBT) promises a principled alternative. By abstracting system behaviors into formal models, MBT allows systematic test generation, quantifiable coverage metrics, and robust validation of requirements. However, despite MBT's theoretical strengths, it has seen limited adoption due to practical barriers. Most MBT frameworks—such as Microsoft's Spec Explorer and GraphWalker[1]—rely heavily on manually defined finite-state machines, or on complex statecharts, which testers often find unintuitive and burdensome to scale or maintain (Jacky et al. 2007).

Common limitations of traditional MBT tools include:

- **Complex Model Creation**: Tools like Spec Explorer require significant expertise to create accurate and comprehensive state-machine models, limiting adoption and usability among QA teams (Jacky et al. 2007).
- **State Explosion and Scalability Issues**: Tools such as GraphWalker or finite-state machine-based frameworks often struggle as system complexity increases, encountering severe performance degradation or outright failure when faced with large state spaces (Utting and Legeard 2006).
- **Coverage Blind Spots**: Automatically generated test cases from traditional MBT tools frequently miss critical edge cases or system states that the model creators haven't explicitly modeled. This leaves significant gaps in testing coverage (Utting and Legeard 2006).
- **High Maintenance Overhead**: Evolving requirements and iterative design processes demand continuous updates to test models. Traditional MBT tools typically fail to provide easy maintenance capabilities, resulting in high overhead and reduced agility.

This paper argues that Behavioral Programming (BP), combined with Generative AI, specifically Large Language Models (LLMs), effectively addresses these limitations. BP introduces a modular approach to modeling, where system behaviors, requirements, and constraints are represented as independent modules known as "b-threads." Each b-thread defines specific behaviors and synchronizes with others through a straightforward event-based protocol: threads request events, wait for events, or block events from occurring. This modular structure inherently simplifies model creation, maintenance, and scalability – alleviating the above-listed limitations. Furthermore, it maintains the descriptive power of state machines, while using modeling idioms that are more concise and intuitive for humans.

Using Generative AI tools to generate and maintain the model further lowers MBT's entry barrier[2]. AI-driven generation of b-threads from informal user stories or high-level requirements greatly reduces manual modeling effort, enabling QA professionals to contribute directly to model creation even if they lack coding expertise. Additionally, AI-assisted MBT dynamically adapts, identifies coverage gaps, and generates richer, more comprehensive test suites.

[1] https://graphwalker.github.io/

[2] The authors have successfully used various GPT-4 and Claude Sonnet versions for generation and maintenance of BP models from natural language requirements and user stories.

Through practical case studies, including modeling and testing REST APIs and e-commerce workflows, this paper demonstrates how BP and AI-driven MBT can:

- Transition QA/test team's role in the SLDC from reactive, late-stage involvement to proactive engagement.
- Transform testing models into central artifacts for specification, communication, and design validation.
- Generate comprehensive, scalable, and easily maintainable test suites.

The remainder of the paper explores these benefits in detail, advocating for a significant shift in how QA is integrated into the software development lifecycle.

2 The Problem: Testing Today Is Still Manual and Brittle

Despite significant investment in test automation tools such as Selenium, Postman, and Cypress, the industry continues to suffer from testing practices that are largely manual or semi-automated. Common challenges include:

- **Fragile Scripts**: Automated test scripts often mirror implementation details too closely. For example, UI-level automation tests created using Selenium frequently break when minor UI changes occur, leading to extensive maintenance overhead (Leotta et al. 2013).
- **Delayed QA Involvement**: QA teams are frequently involved only in the late stages of development, limiting their influence on architectural or design decisions. This often results in fundamental quality issues being discovered late in the development cycle (Garousi and Felderer 2017).
- **Unclear Coverage**: Without explicit behavioral models, organizations lack systematic ways to quantify test coverage, assess readiness, or guarantee robustness. Consequently, testing remains incomplete and often fails to cover critical system behaviors (Utting and Legeard 2006).
- **Ad-hoc Requirements Traceability**: Test cases are typically written based on documents or tickets that rapidly become outdated or ambiguous, resulting in weak requirements traceability and uncertainty about test effectiveness (Garousi and Mäntylä 2016).

Traditional MBT approaches attempt to solve these issues but fall short because they require manually building and maintaining large state machines or transition systems. These data structures are cumbersome and unintuitive for many testers (Edvardsson 2016; Jacky et al. 2007).

This paper proposes that integrating Behavioral Programming (BP) with Generative AI significantly improves upon these limitations. BP's modular approach, combined with AI-assisted modeling, provides a practical, scalable, and intuitive solution for creating and maintaining large transition systems, and ensuring they are correct by construction. This empowers QA teams to create and manage comprehensive test models efficiently and effectively.

3 The Opportunity: What BP+AI Testing Offers

Behavioral Programming (BP) is an innovative scenario-based modeling approach where individual behaviors, requirements, or rules are encapsulated into independent modules called "b-threads" (Harel et al. 2012). These threads communicate and synchronize via a clear and simple protocol:

- **Request**: A b-thread signals it wants an event to occur.

- **Wait For**: A b-thread pauses its execution until a specified event occurs.
- **Block**: A b-thread explicitly forbids events from occurring.

The combined effect of active b-threads forms a comprehensive model reflecting a complete system behavior. This modular approach fundamentally transforms MBT by enabling:

- **Incremental Model Development**: Test models can be progressively enhanced, adding new scenarios without disrupting existing behaviors.
- **Independent Behavioral Specification**: User stories, business rules, and constraints are clearly and separately articulated, facilitating parallel and collaborative model building.
- **Concurrent and Interleaved Behavior Modeling**: B-threads naturally handle concurrent behaviors and interleaved scenarios without intricate manual synchronization.

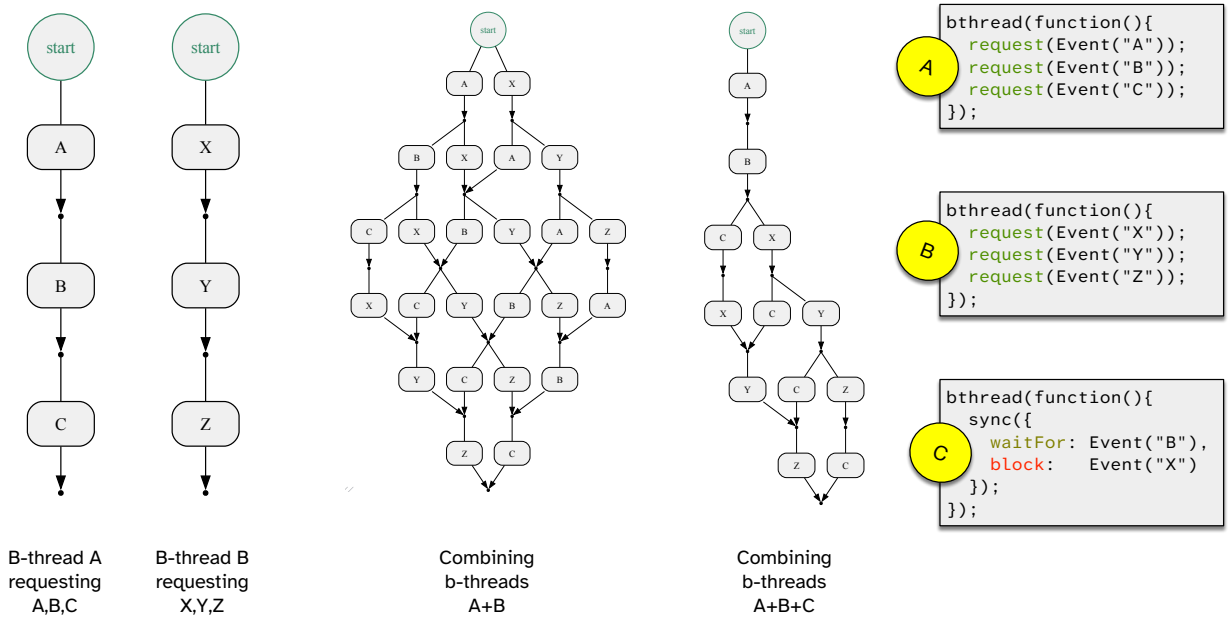

Fig 1. BP models composed of various b-threads. Note that adding more b-threads may reduce test scenario count, due to BP's `block` concept (rightmost model)

Leveraging Generative AI, particularly Large Language Models (LLMs), significantly enhances this process. Because each b-thread encapsulates a small, clearly defined behavior or requirement, LLMs can readily translate user stories or informal requirements into formal b-threads with minimal guidance. This significantly reduces entry barriers for testers and stakeholders who may not possess programming expertise (Brown et al. 2020), empowering non-technical team members to effectively draft, refine, and maintain complex behavioral models.

BP models define extensive system behavior spaces, allowing tools like Provengo to efficiently generate optimized test suites through automated scenario-space exploration. Behavioral Programming supports embedding custom coverage metrics directly within models, enabling precise and detailed coverage measurement and analysis of generated test suites. Visual representations of BP models offer intuitive, real-time documentation of system behavior, significantly improving communication among diverse stakeholders. The modularity of BP facilitates agile maintenance by allowing teams to manage

requirement changes easily through simple adjustments, additions, or removals of individual b-threads, leaving the broader model source code unaffected. Modular b-threads can be readily reused across different projects and systems, enhancing productivity and ensuring consistency. BP models can be integrated with widely used testing tools such as Selenium, Playwright, or HTTP clients, enabling full automation of web and API testing scenarios. Furthermore, the BP framework supports predictive analytics and readiness assessments by proactively identifying coverage gaps, systematically analyzing failure patterns, and providing robust evaluations of system readiness. Lastly, BP models serve effectively in test-driven design, acting as executable specifications and reliable test oracles guiding development from the initial stages (Beck 2003).

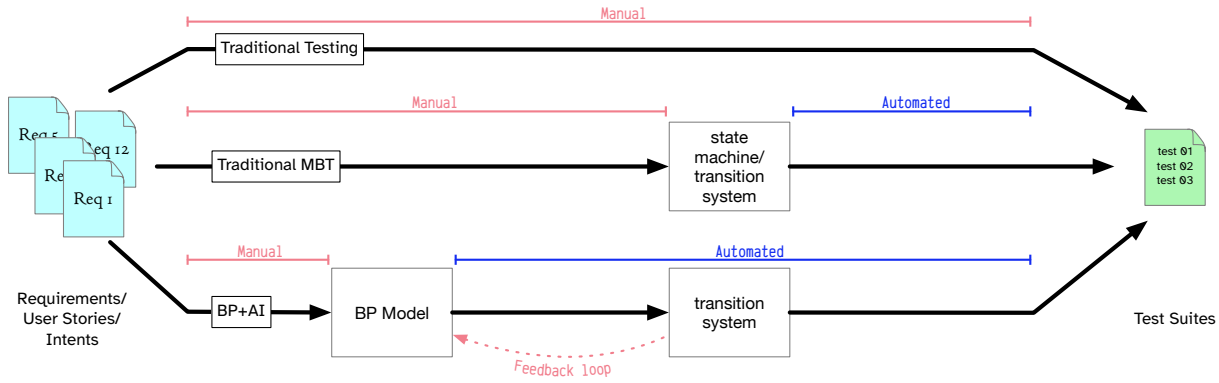

Fig. 2: Comparison of approaches for test suite creation. Traditional testing (top) creates and maintains individual test scenarios manually. Traditional Model-Based Testing approaches require manual creation and maintenance of state machines or similar formal models, but the test scenario generation itself is automated. The proposed BP+AI approach further extends automation, as test teams only need to create and maintain a BP model. The intermediate transition system model allows for an early feedback loop, which enables "human in the loop" model validation (e.g. by visualizations or sample scenario generation) as well as formal verification.

The integration of AI-enhanced Behavioral Programming (BP) models offers several advantages for software quality assurance, transforming how test suites are generated and maintained, and how system readiness is assessed. These benefits include:

- ✓ **Automated Test Suite Generation**: AI-enhanced BP models inherently define extensive behaviour spaces, allowing tools like Provengo to efficiently generate optimised test suites through automated exploration.

- ✓ **Traceability and Custom Coverage Metrics**: BP supports embedding custom coverage metrics directly within models, enabling precise and detailed coverage measurement and analysis. This mechanism can be used for implementing traceability – e.g. by embedding "this test scenario covers requirement 3.6.34" markers on relevant paths within the model.

- ✓ **Improved Communication and Documentation**: Visual representations of BP models offer intuitive, real-time documentation of intended system behavior, significantly improving communication among diverse stakeholders.

- ✓ **Agile Maintenance**: The modularity of BP facilitates agile maintenance by allowing teams to manage requirement changes easily through simple adjustments, additions, or removals of individual b-threads, without affecting the broader model.

- ✓ **Reusability**: **Modular b-threads can be readily reused across different projects and systems**, enhancing productivity and ensuring consistency.

- ✓ **Accessibility for Non-Technical Teams**: Generative AI, particularly Large Language Models (LLMs), empowers non-technical team members to effectively draft, refine, and maintain complex behavioural models.

- ✓ **Automation Tools Integration**: BP models can integrate with widely used testing tools such as Selenium, Playwright, or HTTP clients, enabling realistic web and API testing scenarios.

- ✓ **Predictive Analytics and Readiness Assessments**: The BP framework supports predictive analytics and readiness assessments by proactively identifying coverage gaps, systematically analysing failure patterns, and providing robust evaluations of system readiness.

- ✓ **Support for Test-Driven Design**: BP models serve effectively in E2E test-driven design, acting as executable specifications and reliable test oracles guiding development from the initial stages.

In the next three sections we discuss some case studies we ran with the proposed BP+AI approach. We then demonstrate how these case studies lead us to the list of advantages of the approach given above.

4 Case Study: Using BP and LLMs for MBT

We explore a hybrid strategy that combines ChatGPT, a large language model (LLM), with Provengo, a scenario-based MBT tool grounded in BP. ChatGPT is used to rapidly draft modular behavioral models from informal descriptions, while Provengo systematically expands these models into extensive test suites that exercise both typical user behaviors and hard-to-detect edge cases.

This methodology addresses the current limitations of using ChatGPT alone for test generation. While LLMs excel at generating focused unit tests in response to specific prompts (e.g., writing a Python test for a function `add(a,b)`), they struggle to design coherent, exhaustive test suites that explore the full state space of an application. Critical corner cases, interaction interleaving, and system-wide constraints are often underrepresented or missed entirely. While prompt engineering and “deep-thinking” models may improve generated results, they cannot provide probable comprehensive coverage. In essence, they can generate tests suites, but these suites needs to be tested for scenario correctness and case coverage.

Provengo complements this by creating a computer-actionable formal and deterministic model of the overall system/feature behavior. This is done by combining independently authored *b-threads,* each of which represents a particular behavioral aspect of the system or feature being tested. By feeding AI-generated b-threads into Provengo, engineers can transform intuitive user stories into executable behavior models that yield broad test coverage.

We evaluated this approach on a mid-sized e-commerce web application supporting login, product search, cart operations, and checkout. Testers first provided informal descriptions of both "happy path" scenarios (e.g., successful login and checkout) and "rainy day" scenarios (e.g., invalid credentials, inventory conflicts, or checkout errors). ChatGPT was then prompted to translate these into Provengo b-threads. For example, the prompt:

> “Generate Provengo b-threads for login, add-to-cart (looped), and admin inventory actions, including error cases.”

resulted in well-structured behavior modules within 2–3 iterations per scenario.

Once loaded into Provengo, the composed behavior model was executed to automatically generate a rich set of interleaved test sequences. This process uncovered numerous edge cases, such as a user attempting checkout after a failed login or a cart being locked following payment rejection. Provengo

produced approximately 150 unique test scenarios, 31 of which revealed edge behaviors missed by standard scripted tests.

Key technical benefits emerged from this integration:

- **Rapid Modeling**: ChatGPT significantly reduced the time required to draft initial models, allowing testers and analysts to quickly iterate on scenarios without deep familiarity with BP syntax.
- **Systematic Exploration**: Provengo's execution engine explored complex interleavings, enabling automatic detection of undesirable state transitions and deadlocks.
- **Model-Driven Metrics**: Provengo yielded actionable artifacts such as state coverage maps, error frequency histograms, and behavioral traces—valuable for assessing readiness and guiding debugging.

Compared to related work, such as Bar-Sinai et al. (2023), which demonstrated Provengo's utility in manually modeling user stories, our study introduces automation into the modeling stage, further enhancing scalability. Similarly, our findings reinforce observations by Yaacov et al. (2024) on the modularity benefits of BP and extend them by demonstrating LLM-augmented model authoring.

Nevertheless, limitations remain. ChatGPT occasionally generated incorrect or inconsistent syntax (e.g., non-existent event names or missing synchronization logic), necessitating human review. These issues are consistent with prior findings on LLM hallucination and reinforce the importance of integrating model validation tools in the workflow.

Despite these caveats, the synergy between ChatGPT and Provengo represents a promising evolution in MBT practices. It enables development teams to move beyond reactive test script maintenance toward proactive modeling of intended behavior, enhancing not only testing efficacy but also communication and alignment across roles. As software systems grow in complexity, such AI-assisted, model-centric strategies will become increasingly essential.

5 Case Study: Testing a REST endpoint

In this case study, we used Provengo to test the REST API of a Library Management System. Our goal was to explore how behavioral programming can serve as a practical Model-Based Testing (MBT) approach that integrates smoothly with existing testing workflows, rather than requiring a complete change in mindset or tooling.

The process began with the creation of an interface module, `interface.js`, which defines a set of JavaScript functions that wrap REST calls using Provengo's `RESTSession`. Each function—such as `createBook`, `getBook`, `updateBook`, and `deleteBook` - encapsulates the relevant endpoint call, specifies the expected response status, and includes optional post-processing logic. For example, the `createBook` function not only sends a POST request to add a book but also extracts and stores the returned book ID for later use. This layer of abstraction makes the testing code clean and readable, while also making the test logic reusable and parameterized.

Using these interface functions, we built two different models. The first, found in `a_linear_test.js`, reflects a traditional linear test. It executes a single scenario in a fixed sequence: creating a book, retrieving it, updating its information, and then deleting it. This kind of test is very close to current industry practices, where test engineers write predefined test scripts that correspond to common user interactions. It is familiar and deterministic but limited in coverage and variability.

To move beyond this limitation, we developed a second model in `a_testing_model.js`. This version uses Provengo's behavioral programming framework to describe the test model as a set of independently defined behavioral threads, or b-threads. Some b-threads test valid sequences, such as a user adding

and viewing a book. Others describe negative test cases, such as attempting to create a book with a duplicate ISBN or trying to update a deleted book. The result is a much richer and more comprehensive test suite, automatically generated by composing the individual behavior threads.

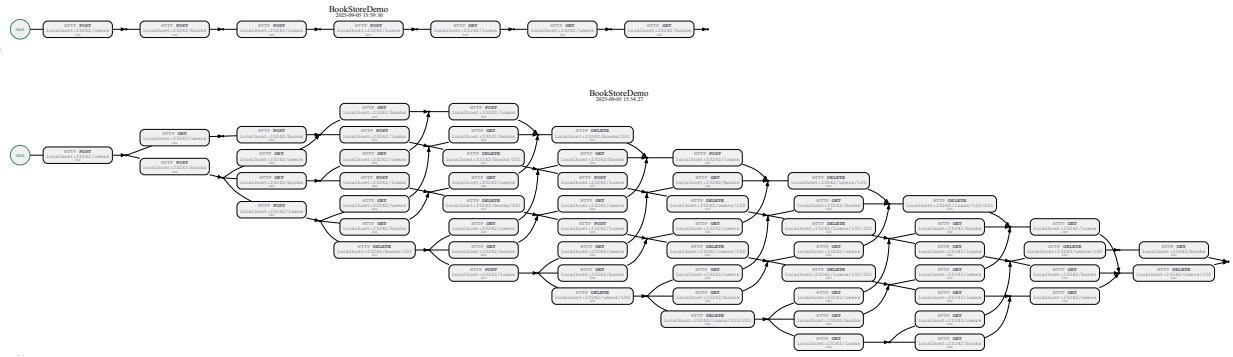

Fig. 3: State machines for the linear test scenario (top) and test model (bottom). The test model is composed of two b-threads, one describing a librarian scenario, and the other describing a book borrowing scenario. Both diagrams were automatically generated using Provengo, and represent the HTTP call sequences for the library's API.

This approach contrasts sharply with most existing REST testing tools, which typically focus on validating individual endpoints in isolation. These tools are well-suited for testing single-step interactions, such as verifying that a POST request returns a `201 Created` response, but they do not aid testers in composing multi-step, stateful workflows that involve sequences of operations with intermediate data dependencies. In practice, however, many important bugs and integration issues only emerge when such sequences are tested holistically. Provengo directly addresses this gap by supporting the modeling and execution of multi-step scenarios, enabling realistic and thorough exploration of system behavior over time.

The transition from the linear script to the behavior model illustrates one of Provengo's key advantages: it allows teams to evolve their testing practices incrementally. The same interface layer is used in both models, and the same REST testing infrastructure supports them. This continuity stands in contrast to many other MBT tools, which often require adopting new modeling languages, building explicit state machines, or learning specialized DSLs. For instance, tools like Microsoft Spec Explorer require C#-based model programs and explicit exploration strategies, while tools based on formal specification languages (e.g., Alloy, TLA+) demand a steep learning curve. In comparison, Provengo's approach is lightweight and accessible, letting developers and testers use plain JavaScript to describe system behaviors in terms they already understand.

By combining a familiar programming environment with powerful model-based capabilities, Provengo makes it easy to enhance test coverage and robustness without abandoning existing workflows. This case study illustrates how real-world teams begin with conventional tests and gradually evolve them into expressive, scenario-rich models, reaping the benefits of MBT without the usual barriers to adoption.

All the code referenced in this case study is available in the open-source Provengo REST Tutorial repository: https://github.com/Provengo/REST-Tutorial.

6 Case Study: Applying Generalized Coverage Criteria in a Behavioral Programming Context

To demonstrate the practical benefits of integrating generalized coverage criteria with behavioral programming (BP), we conducted case studies on two representative systems: the Alternating-Bit

Protocol (ABP) and the Moodle Learning Management System (LMS). These studies illustrate how system-specific behaviors can be modeled using BP and tested using automata-based coverage criteria, as proposed in the generalized sequence testing framework of (Elyasaf et al., 2023).

In our approach, the test model is represented as a set $P \subseteq \Sigma^*$, where each element in *P* is a sequence of events the system might experience (such as a login, entity creation, or an API call). We defined coverage criteria as an indexed set $C(i)$ for i in I, with each $C(i) \subseteq \Sigma^*$ specifying a class of event sequences that should be represented in the test suite. This formalism allowed us to explore different strategies for defining what it means for a test suite to be "adequate."

We explored two types of criteria. The first, based *t-way sequence coverage* (Kuhn et. al., 2012)[3], defines each $C(i)$ as a regular language accepting any sequence containing a given t-length subsequence (e.g., $\Sigma^*\sigma_1\Sigma^*\ \sigma_2$). The second, which we refer to as *generalized consecutive coverage*, is stricter: each $C(i)$ requires that a specific subsequence w appear in the test contiguously (i.e., $\Sigma^* w \Sigma^*$). This distinction is critical in cases where event order and proximity are semantically meaningful, such as in transaction protocols or interactive user sessions.

Our evaluation methodology combined BP-based model execution with evolutionary optimization. We constructed modular behavioral models using b-threads and used random exploration to generate a pool of 50,000 test cases. Coverage criteria were encoded as ranking functions, and we employed a genetic algorithm[4] (GA) to generate optimized test suites that maximized the chosen criterion.

We opted to use GA here, as it provides measurably good results, and is easily generalizable, since users can define their own coverage criteria by providing their own fitness function. Additionally, while the GA algorithm involves a lot of randomness, its final result is explainable, a desired property in systems that involve AI. Note that the problem of creating a "best" test suite for general criterion, being a general combinatorial optimization problem, is computationally hard (basically an instance of the Knapsack problem).

In the ABP case study, we introduced faults into the implementation and measured how different coverage criteria affected bug discovery. For example, the bug triggered by the sequence `'sNak, sNak, rAck'`—in which the receiver fails to send an acknowledgment after two negative acknowledgments—was detected in 6.7% of random test suites and 9.1% of suites generated to maximize t-way coverage, but was found in 21.6% of those optimized for the consecutive coverage criterion ($\Sigma^* w \Sigma^*$). Similarly, the bug triggered by `'send, send, sAck'` was detected in 82.1% of random suites, 80.6% of t-way suites, and 97.8% of suites generated using the generalized consecutive coverage criterion. These results support the claim that the ability to precisely define and target meaningful sequences in the behavior space substantially improves test effectiveness.

In the Moodle LMS case study, we applied our method to model user roles and actions, such as a teacher adding quiz questions and a student submitting answers. Our behavioral model included three b-threads representing administrator setup, teacher quiz management, and student quiz interaction. Using this model, we uncovered a bug that allowed a teacher to submit a new question to a quiz while a student was mid-submission, a behavior that contradicts Moodle's intended access control. This error was reliably detected using the Kuhn and Higdon 3-way sequence coverage criterion, achieving a detection rate of

[3] T-way sequence coverage is a method for composing test suites for interactive systems that may receive multiple events, where event order matters (such as communication devices). Given a set containing T events, this method will generate a test suite testing all ordering of these events. For T=2, this method is similar to pairwise testing.

[4] The "organism" used in this algorithm is a test suite, and the "genes" are test scenarios. The fitness function reflects the coverage criterion, and is calculated over the test suites, based on the content and properties of the test scenarios they contain. We used 3000 generations with crossover probability of 0.7 and mutation probability of 0.05. Population size in each generation is 50.

98.6%. Generalized criteria also detected it, albeit with slightly lower rates, depending on how the sequence definition aligned with the fault condition.

These studies highlight the advantages of using generalized coverage criteria in conjunction with BP. Testers can express domain-specific concerns, such as ordering constraints, safety conditions, or concurrency patterns, as regular languages, and tools can then optimize test suites to cover them. Moreover, BP's modularity allows these behaviors to be added incrementally, without requiring global restructuring of the model. When test resources are limited, coverage criteria can be relaxed, grouping similar behaviors to reduce effort while still ensuring meaningful exploration. Finally, the Bayesian estimation method proposed in the generalized framework can quantify residual risk, guiding further test generation based on remaining uncertainty.

Together, these capabilities create a powerful testing strategy that is precise, scalable, and adaptable. The integration of generalized sequence coverage criteria, behavioral modeling, and evolutionary search supports rigorous testing in complex domains without the overhead of traditional Model-Based Testing frameworks. This approach enables testing teams to maintain process and automated test agility without sacrificing software quality.

7 Conclusion

Combining behavioral programming and LLMs makes model-based testing accessible, scalable, and deeply integrated into real-world testing practices. This approach shifts QA team role from reactive test case maintenance to proactive behavioral modeling—unlocking the full promise of MBT.

References

Bar-Sinai, Michael, Achiya Elyasaf, Gera Weiss, and Yeshayahu Weiss. 2023. Provengo: A Tool Suite for Scenario-Driven Model-Based Testing. CoRR abs/2308.15938.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. "Language Models are Few-Shot Learners." arXiv abs/2005.14165. DOI: 10.48550/arXiv.2005.14165.

Beck, Kent. 2003. "Test-Driven Development: By Example". Addison-Wesley Professional. ISBN 0-321-14653-0. 220 pages.

Elyasaf, Achiya, Eitan Farchi, Oded Margalit, Gera Weiss, and Yeshayahu Weiss. 2023. "Generalized Coverage Criteria for Combinatorial Sequence Testing." IEEE Transactions on Software Engineering.

Garousi, Vahid, and Michael Felderer. 2017. "Developing, Verifying, and Maintaining High-Quality Automated Test Scripts." IEEE Software 34, no. 6: 68-74.

Garousi, Vahid, and Mika V. Mäntylä. 2016. "When and What to Automate in Software Testing? A Multi-Vocal Literature Review." Information and Software Technology 76: 92-117.

Harel, David, Assaf Marron, and Gera Weiss. 2012. "Behavioral Programming." Communications of the ACM 55, no. 7: 90–100.

Jacky, Jonathan, Margus Veanes, Colin Campbell, and Wolfram Schulte. 2007. Model-Based Software Testing and Analysis with C#. Cambridge: Cambridge University Press.

Leotta, Maurizio, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. "Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution." In IEEE International Conference on Software Maintenance, 272-281.

Utting, Mark, and Bruno Legeard. 2006. Practical Model-Based Testing: A Tools Approach. Burlington, MA: Morgan Kaufmann Publishers Inc.

Yaacov, Tom, Achiya Elyasaf, and Gera Weiss. 2024. "Boosting LLM-Based Software Generation by Aligning Code with Requirements." In Proceedings of the 32nd IEEE International Requirements Engineering Conference Workshops (REW 2024), 301–305.

Yaacov, Tom, Achiya Elyasaf, and Gera Weiss. 2024. "Keeping Behavioral Programs Alive: Specifying and Executing Liveness Requirements." In Proceedings of the 32nd IEEE International Requirements Engineering Conference (RE), 91–102.

D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker and Y. Lei, "Combinatorial Methods for Event Sequence Testing," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, Canada, 2012, pp. 601-609, doi: 10.1109/ICST.2012.147.