Extending Model-Based Testing for Agile Development: Managing Boundaries of Incomplete Systems

BOGDAN BUSUI, University of Twente, The Netherlands

Model-Based Testing (MBT) is a powerful approach to software testing that enables systematic test generation from formal models of system behaviour. While traditionally applied to fully implemented systems, its integration into Agile and iterative development workflows—where features may be only partially complete—remains a challenge. This paper investigates how MBT can be adapted for use in systems under active development, where certain components are not yet available. The focus is on modelling such systems using Labelled Transition Systems (LTS), identifying and formalising model boundaries at the interface between implemented and unimplemented features, and adapting the model creation to preserve the LTS structure and prevent invalid or unexecutable test cases. A small-scale prototype—a digital printer with an evolving 3D printing feature—is used to demonstrate the feasibility of this approach. The study proposes practical modelling strategies that incorporate bounds between modelled and unmodelled behaviour, and feature flags, allowing early testing and incremental test model refinement. The findings support the viability of MBT in dynamic, partially completed systems, offering a pathway to earlier fault detection and improved alignment with modern software engineering practices.

Additional Key Words and Phrases: Model-based testing, MBT, LTS, unimplemented features, incomplete systems, system under test, sut

1 INTRODUCTION

In software development, detecting defects early in the lifecycle is critical for minimising costs, ensuring system quality, and accelerating delivery timelines. As modern systems grow increasingly complex, traditional manual testing methods struggle to keep pace with the breadth of execution paths and feature interactions. This has led to a growing interest in Model-Based Testing (MBT), a technique that automates test case generation from formal models of system behaviour. By abstracting the system into a set of states and transitions, MBT enables more systematic and exhaustive testing compared to manually written test suites.

In principle, MBT offers significant advantages, including improved coverage, early detection of specification mismatches, and a clear link between requirements and test cases. However, its adoption in the industry has been limited due to several practical challenges. Firstly, MBT introduces a steep initial cost: it requires not only toolchain integration and modelling tools, but also a workforce trained in formal modelling. The quality of the entire testing process becomes highly dependent on the quality of the model—any design flaw or misrepresentation can lead to misleading tests or false confidence. Moreover, MBT's structured, model-driven process often clashes with the rapid, iterative cycles typical of Agile and DevOps methodologies.

TScIT 43, July 4, 2025, Enschede, The Netherlands

In an Agile framework, testing can consume up to 30% of total project workload [4], a figure that underscores both the importance and intensity of continuous testing. Agile teams require testing strategies that can evolve alongside the implementation, adapting to changes without incurring excessive rework. Unfortunately, traditional MBT methodologies typically assume a fully specified system under test (SUT), which poses problems in development environments where features are built incrementally and may remain partially implemented for extended periods.

To bridge this gap, this paper investigates the feasibility and methodology of applying MBT to unfinished systems—systems in which certain components are unimplemented, under construction, or delayed by design. The goal is to explore how such systems can be formally modelled, how boundaries between implemented and unimplemented features can be defined, and how test generation can remain effective in these conditions.

The modelling approach used in this study is based on Labelled Transition Systems (LTS), a well-established formalism in which system behaviour is described through a set of states, actions, and transitions [5]. The states model the system states; the labelled transitions model the actions that a system can perform. This paper focuses only on LTS for modelling, due to their formal simplicity, direct applicability to model-based testing frameworks and the availability of industry-ready tools that support automated test generation.

In the context of MBT, an LTS can be converted into executable test cases using specialised tools. While such tools already exist and are outside the scope of this paper, the modelling itself—particularly in cases of partial implementation—presents significant research opportunities. This project focuses on how to model such systems accurately, how to simulate or stub the behaviour of unimplemented components, and how to ensure that the resulting tests are both meaningful and maintainable as the system evolves.

Building on this foundation, several prior studies have explored the use of LTS and related formalisms to support model-based testing in scenarios involving incomplete or evolving systems. For instance, Utting and Legeard [7] provide a comprehensive overview of MBT practices, including the role of abstract models in early testing phases. More recent work, such as by Artho et al. [2], investigates the use of stub models to enable testing in the presence of unavailable components, advocating for explicitly defined I/O behaviours at model boundaries. Similarly, research on interface automata and input-output labelled transition systems (IOLTS) has contributed techniques for managing test execution in systems with uncertain or evolving interfaces [1]. This paper builds upon these directions by addressing a gap in how formal models can be systematically extended with intelligent stubs or abstractions that allow for seamless integration into continuous development workflows, without compromising test validity or model fidelity.

2 PROBLEM STATEMENT AND RESEARCH OBJECTIVES

While Model-Based Testing (MBT) presents a promising alternative to traditional testing approaches, it is predominantly designed for systems that are fully specified and implemented. In practice, particularly within Agile and iterative development workflows, systems are often incomplete for extended periods. Certain features may be planned but not yet implemented, while others may be undergoing revision or awaiting integration. This poses several challenges when attempting to apply MBT in such environments.

The fundamental issue lies in the mismatch between MBT's assumptions and the realities of incremental development. Traditional MBT workflows presume the existence of a well-defined, complete model of the system under test (SUT), against which tests can be automatically generated and executed. However, in an evolving system, such a model cannot be fully realised. This raises a set of critical questions:

- **How can we accurately and meaningfully model only the parts of the system that have been implemented, while preserving their inner functionality?**
- **How can we formally define the boundaries of the model—the points at which implemented components connect to unimplemented or placeholder features?**
- **How do we ensure that the tests resulting from the test generation algorithms are valid and informative, even in the presence of missing features that do not represent real execution paths?**

These questions form the foundation of the research presented in this paper. The objective is to explore and develop methodologies that allow MBT to function effectively within partially implemented systems. By doing so, testing can begin earlier in the development cycle, providing feedback that is otherwise delayed until after the implementation is completed.

To address these challenges, the focus of this paper and contributions to the topic at hand are the following:

- **Investigate formal approaches for modelling partially implemented systems within the LTS framework.** The emphasis is on preserving testability and correctness while accounting for system evolution. In this context, *testability* refers to the conservation of the LTS structure, which can be imported to different software tools (for example TorXakis or GraphWalker). *Correctness* relates to the system's interactions being unchanged following the found implementations, and future modifications to the system, by expanding the functionality do not affect the existing feature-set.
- **Define strategies for handling model boundaries.** This includes the use of placeholder transitions to represent known inputs and outputs without requiring full internal behaviour implementation. This refers to the boundaries of the model, before the software implementation and expansion of the model happens.
- **Adapt test generation algorithms to produce executable test cases that avoid interactions with undefined states or transitions.** This includes incorporating guards or conditional logic to distinguish between implemented and placeholder behaviour. This, as it will be addressed further on, turned out not to be necessary as the entire adjustment is handled during the modelling process, leaving the test generation unchanged, adhering to the testability objective mentioned above.
- **Validate the proposed modelling strategies through a prototype example.** Although full automation is outside the project's current scope, a structured modelling and test design process is demonstrated using an illustrative system—a digital printer with evolving feature sets.

3 EXAMPLE PROTOTYPE IMPLEMENTATION

Throughout the project, given the theoretical nature of the findings, no actual implementation could be done. However, this does not mean the findings lack practical application. For this reason, a small-scale system example has been devised in order to combine all relevant parts of MBT. These include the modelling of the existing system, creating boundaries for the unimplemented features and generating comprehensive test cases.

Our working example represents a digital printer that has the following functionalities: it can load documents (via an external means, sending them directly to the printer, or by scanning them, which are outside of the scope of the model), and it can print them, after they have been loaded. There is a failsafe mechanism that prevents the system from wrong inputs (e.g. printing without having loaded a document beforehand). If the system enters the failsafe section, a red LED will light up, the "Reset" button must be pressed, and the printer goes back to idle (waiting for a document). The following diagram (Figure 1) presents the Labelled Transition System (LTS) representation of the printer.

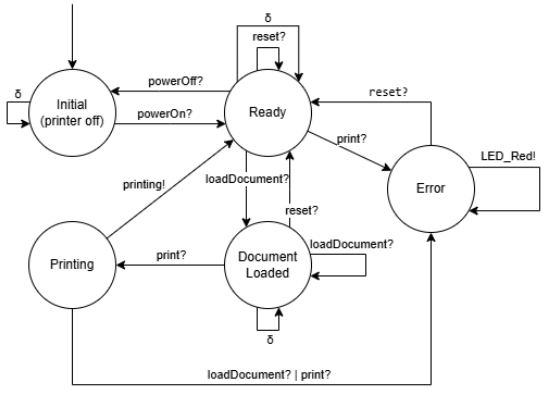

Fig. 1. A Labelled Transition System of a Printer

In the "Initial" state, the printer is turned off. By pressing the "Power On" button, it switches to the "Ready" state, from where, by loading a document, it transitions to "Document Loaded". Then, selecting the 'Print' button causes the system to produce its output: the printed document. If an input that violates the expected sequence of operations is received, the printer goes into the "Error" state, signalled by a red LED, which can be recovered from by pressing the "Reset" button. An error can happen in more than one way, all of which are modelled by arrows pointing from different states to

the “Error” state (for example, printing without a document loaded). All but the “Printing” and "Error" states have quiescence (δ), as they do not have any output, but that is the correct behaviour. Pressing the “Reset” button from any state that accepts input will end up in “Ready”. From the “Ready” state, by pressing the “Power Off” button, the system shuts off.

The next development cycle aims to extend the functionality of the system. For instance, the printer could also have a 3D printing feature. This takes a different type of document (3D printer files) and takes a completely different path inside the system. The normal “Document Loaded” and “Printing” states (for regular printing) would no longer adhere to the new specification. Our implementation is to have a separate UI element that only accepts 3D printing files and follows its instructions. This would be an upload field that accepts only files that can be 3D printed. Another distinct feature is that the user no longer has to start 3D printing; it is done as soon as the file has been checked for correctness. Figure 2 portrays a modified diagram that models the newly added changes, but does not integrate the new features. What we know so far (and where the implementation currently stops) is that there is an input loaded into the system, in the form of a 3D file, and there is an output provided, namely a 3D printed object. The exact details as to how the scanning or printing happens are unknown so far, which is why the placeholder "FutureState" represents an abstraction of these functionalities. The idea of stopping here and not have the feature implemented is to mimic an ‘unfinished’ behaviour and present how that can be adapted in the LTS.

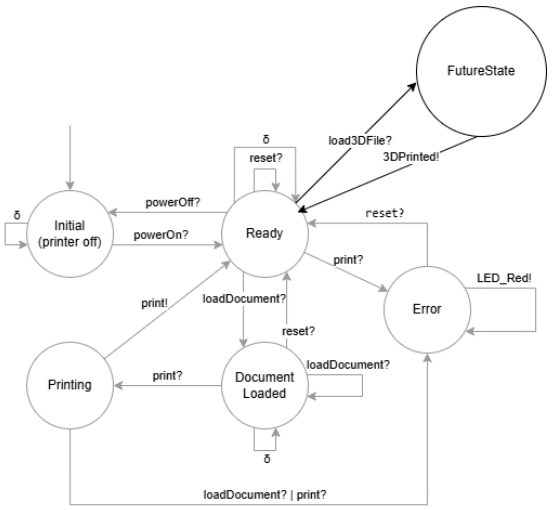

Fig. 2. Updated LTS with 3D printing partial modelling

This updated version has yet to implement the features; however, it knows what type of input and output is expected, making it a valid solution, respecting the LTS format. From the “Ready” state, providing a 3D file will load it and print it. Following that, the result is output. Despite the missing implementation states, the behaviour is known, making the system ready for testing. This refers to the fact that the unimplemented functionality does not affect the behaviour of the test generation itself, as the heavy modifications were handled during the modelling process (proper construction of the LTS).

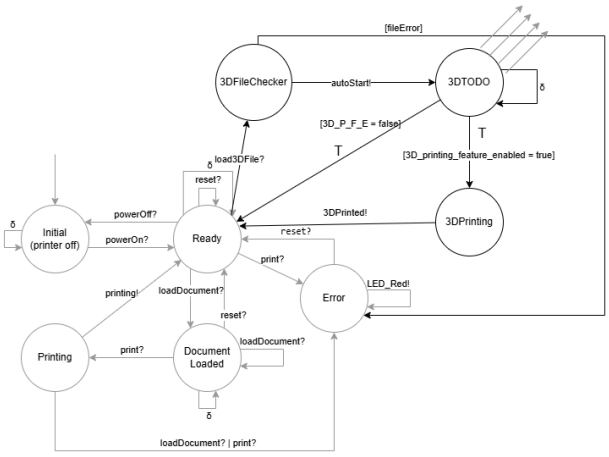

Fig. 3. Subsequently updated LTS with 3D printing possible functionality

Following the implementation of the 3D printing features in the system and a subsequent modification of the model (Figure 3), the system can now create 3D models. Once a file has been loaded, the 3D printer automatically starts printing. This design choice maximises the efficiency of the printer, since 3D printing file-checking can take an undetermined amount of time, and this removes the necessity of having a human in the loop once it is finished loading. This specific diagram comes at a time in the development process where the file-checker (3DFileChecker) has been implemented, but it accommodates both having and not having the actual functionality of the system (3DPrinting). This could happen if the feature has been accepted by the management team, but before it is implemented, to be able to present to the stakeholders the flow of the system and perform testing on 3D file checking. The basis of this update is having guards that allow the possibility to add new features, after they have been implemented. This ensures that the model is always in line with the development process, and it does not precede any functionality of the system. This safety measure protects against testing features that do not exist, which would lead to numerous errors. If the flag "3D_printing_feature_enabled" is set to true (i.e. the printer can 3D print), in the LTS diagram, this presents itself as going to the 3DPrinting state. From there, the output is the physical, printed object. Conversely, the boolean is set to false until the feature exists, meaning that nothing is output, and the system goes back into the Ready state. This ensures that nothing is unmodelled, and there are correct transitions between the appropriate states. The possible transitions point to the option of further expanding the system and adding more features, similar to Figure 2. It suggests and allows system growth, despite also being a stable and complete system.

4 MODELLING THE SYSTEM

Integrating Model-Based Testing (MBT) into an active development workflow requires a series of structured steps that support both the progressive construction of the model and the incremental implementation of system features. In the context of partially completed systems, modelling becomes more than a static representation—it acts as a border between the existing and future features, with only the current ones being modelled. Many times during the development cycle, it can happen that work is being done on multiple fronts, with several components being worked on concurrently, with some of them functional, while others are still in progress. The best way to differentiate between them is, in the case of MBT, by modelling only the working ones. This is advantageous in several respects. On one hand, the model conveys all the information about what is currently working in the system, making it a great showcase for stakeholders or higher management, who are not familiar with all components. On the other hand, having an accurate LTS that only models the live system makes it effortless to test, as it can simply be imported into specific tools that generate testing automatically [5]. The steps below describe the typical workflow in an MBT development cycle:

- **Feature Planning and Design:**
  - The management team and stakeholders define and analyse the new feature or requirement. After it has been approved and accepted, two tracks proceed in parallel: Development (Code) and Modelling (Test Model).
- **Track A: Software Implementation:**
  - Architecture and Design Decisions
  - Code Implementation
  - Unit Testing
- **Track B: MBT Model Development**
  - Creation of an LTS adhering to the specification or plan
  - Model Validation and Refinement
  - Test generation using dedicated tools, see section 5
- **Tracks A and B:**
  - Convergence of the Software Implementation and the Model Development
  - Correctness checking

  Track A is a conventional development process of a system that can follow any framework. Track B marks the modifications due to the MBT approach, with extra work needed concurrently to model the system, facilitating the testing goal. At the end, a convergence is made that ensures the implementation passes the created tests, signalling success on both branches. If inconsistencies appear, correctness checking has to be done by both sub-teams until the alignment is perfect. This cycle can then be repeated as many times as needed to complete the system, and it accommodates for both small or large features.

To begin the schematic creation, the model should be constructed using a formalism such as Labelled Transition Systems (LTS) (e.g. Figure 1), which offers clarity in representing system states, actions, and transitions. Modelling should focus first on the core, stable behaviour—features that are already implemented or have stable specifications. This forms a dependable foundation on which additional components can be incrementally layered.

When dealing with features that are not yet implemented, the model must still represent them in a meaningful and testable way. This involves defining boundary points—states and transitions that mark the interface between implemented and unimplemented functionality. In these cases, inputs and outputs can still be modelled based on design documentation, interface specifications, or expected contracts. These behaviours are represented using transitions or state representations that simulate the eventual behaviour without requiring actual code execution.

Importantly, LTS modelling allows for quiescence (δ), representing the absence of observable output. This concept is particularly useful for modelling unimplemented features: while the input may be accepted by the system, the expected output can be replaced with quiescence or redirected to a placeholder state. This ensures that the model remains logically complete and testable, even if the system itself cannot execute the intended behaviour yet.

Parallel modelling and integration support a more agile development process. As new features are implemented, model boundaries can be replaced with fully modelled behaviour, allowing the test suite to evolve without invalidating previous work. This iterative alignment of model and implementation minimises rework and maximises test reuse.

Despite its advantages, modelling under these conditions also presents difficulties. The boundaries between implemented and unimplemented features must be explicitly managed to avoid generating invalid or misleading test cases. Incorrect assumptions about future behaviour can also compromise the validity of the model. Therefore, maintaining clear documentation and formal specifications is crucial for long-term model integrity.

Ultimately, the goal is to construct a living model that grows in parallel with the system itself, supporting continuous testing, early fault detection, and a more predictable development cycle.

5 TEST CASE GENERATION

Once the system has been modelled using LTS or a similar formalism, the next step in the MBT process timeline is the automatic generation of test cases. Test generation relies on traversing the modelled transitions and producing test cases that verify the correct behaviour of the system under test (SUT).

In the case of partially completed systems, traditional MBT tools may struggle to differentiate between legitimate behaviour and modelling edges. Therefore, the adaptations made to the modelling cycle must still respect the LTS format. Furthermore, the model must not convey information about missing features, but include boundary states that emulate a functionality edge, like in Figure 2.

Some tools that support LTS-based test generation include:

- **TorXakis**: An experimental tool for on-line model-based testing. TorXakis implements the ioco-testing theory for labelled transition systems [6].
- **JTorX**: A Java-based MBT tool that performs on-the-fly test derivation and execution from labeled transition system (LTS) models based on ioco conformance testing. It generates and executes test cases dynamically, facilitating immediate feedback during model exploration [3].

- **GraphWalker**: An open-source MBT framework that leverages graph-based models (e.g. extended finite state machines) to automatically generate test paths. It supports various algorithms (random, edge coverage) and can be integrated into CI/CD pipelines [8].

The generation process can be tailored by setting coverage criteria such as transition coverage, state coverage, or specific scenario testing. For partially implemented systems, test selection strategies should avoid paths that rely on unimplemented features, or they should verify that the system responds with a stubbed output or enters a default state like “Ready” or “Error.”

In our first printer example (Figure 1), test cases would include:

- Verifying that a document cannot be printed without first loading it.
- Confirming that the “Reset” button restores the printer to the ready state from any error state.
- Ensuring that the intended execution flow—powering on the system, loading a document, and initiating the print operation—produces the expected output, namely the successful printing of the document.

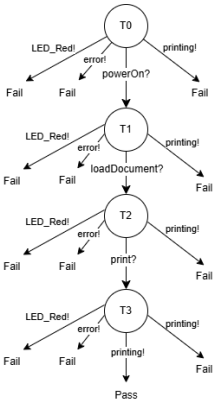

Fig. 4. Hand-generated Case for Printing Functionality in the Initial Model

Figure 4 presents a simple test case for the initial system (Figure 1). It starts from T0, which correlates to the "Initial" state in the model. From there, given the inputs/ outputs from the transitions, the test case grows in a tree-like manner. We observe that only after correctly powering on the printer, followed by loading a document, pressing the print button and receiving the printing, the test is a success, all other possibilities lead to failure. This simple example showcases the structure of a test case, as well as the conditions to pass one, having exactly one pass, but may have multiple failure points.

Figure 5 presents a test case on the boundaries of the updated printer (Figure 2), where the functionality of the printer is missing, but a stub implementation would allow this test to pass, as the tested feature is not the print itself, but rather the connection between uploading a 3D file and receiving an output from the system. At every step where an input is given, the system may produce an output or just continue its execution. The figure denotes a fail for every single output that is given at the wrong time, as it correlates to

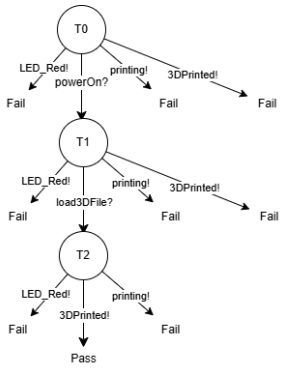

Fig. 5. Manual Test Case for 3D Printing in the Updated Model

a wrong implementation. The correct sequence is the following: the printer is powered on, a file is uploaded (unimplemented currently, only a placeholder exists), and a 3D printed file is created (also a stub in this case). This test case may seem redundant because no functionality is added, both file checking and printing are just stubs, but it does hold the information of the possibility of expansion in the future, once the features are integrated. Figure 2 represents how the system boundary performs when a feature is not yet implemented, but a placeholder is in its place, and Figure 5 is the test case for the connected links, making abstraction of absent elements.

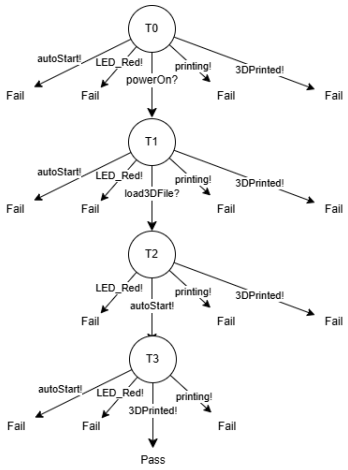

Fig. 6. Hand-Crafted Test Case the Implemented 3D Printing Functionality

Figure 6 displays a test case for the complete, implemented 3D printing feature. The increased number of possibilities is directly tied to the multitude of possible outputs that the system can produce, with all of them having to be accounted for. The main difference between this test case and the previous one is the availability of the system’s functionality. Now, the 3D printing and file checking are no longer stubs in the code, but are working and should lead to the expected results. Similar to the previous test cases, for the test to pass, after it is turned on, a 3D file is input. After the file is successfully checked, the auto-start function starts the printing

process. At the end, the correct output is the created object. For systems where output behaviour depends on feature toggles (e.g., 3D_printing_feature_enabled), test cases should be conditionally generated based on those flags, allowing flexible execution across multiple development stages. This increases reusability and reduces test maintenance overhead. By reusability we mean the system becomes more modular due to the addition of new parts or functionalities. The core structure remains the same, it is just expanded on and it can grow as much or as little as the system requires. It accommodates both complex and simple systems, the difference being the overall size of the project.

6 EVALUATION AND DISCUSSION

Our results so far have demonstrated that Model-Based Testing can be adapted to systems that are under development, through the combined use of formal modelling, boundary definitions, and carefully constructed stubs. Modelling with LTS provides a robust framework to represent both current functionality and anticipated features, while clearly separating testable paths from speculative ones. This refers to modelling the parts of the system that exist or that have stubs, as they can be directly tested in different conditions (existence, performance, etc.), and it eliminates the need to model completely missing features, as they cannot be tested yet anyway.

One key insight is the importance of interface-level contracts when modelling unimplemented features. Even without execution logic, well-defined I/O specifications enable the model to simulate interactions and define expected behaviours. These placeholders are instrumental in maintaining model continuity and allowing early-stage test generation.

Furthermore, test generation tools such as TorXakis and JTorX support extensions for quiescence, unobservable transitions, and conditional behaviours. This enables testers to generate test suites that are both comprehensive and realistic, avoiding unexecutable paths.

However, the approach is not without limitations. Stub behaviour (in the code implementation) may diverge from future implementations if design assumptions change, which can invalidate previously generated tests. Additionally, the boundary between stubs and actual logic must be managed with precision to prevent false positives or negatives during test execution.

The integration of MBT into agile development cycles also requires cultural and procedural shifts. Teams must align on modelling practices and maintain synchronised documentation to prevent drift between implementation and test models.

7 CONCLUSION AND FUTURE WORK

This paper has explored the application of Model-Based Testing to systems that are still in development, with an emphasis on formal modelling using LTS, handling unimplemented features through boundary-aware stubbing that prevent the need for adapting generating test cases algorithms, thus adhering to the existing tools.

A small-scale prototype model—a printer with partially implemented 3D printing functionality—was used to demonstrate the viability of these techniques. By defining expected I/O behaviour and modelling future states as model boundaries, we maintained model integrity and test generation capability throughout the development lifecycle.

Future work could involve:

- Automating the identification of model boundaries and stub implementations, and allowing for a preview of the complete system from the early implementation.
- Extending the prototype to a possible real-world scenario, using MBT in a future project, and utilising the techniques and methods found during the research period.
- Integrating MBT into CI/CD pipelines with dynamic model updates as features are implemented.

The findings of this research support the position that MBT, when adapted with formal strategies for partial systems, can play a crucial role in improving software quality even before the system is fully implemented.

REFERENCES

[1] Bernhard K. Aichernig and Tanja T. Gschwind. 2004. Formal Test-Case Generation from Timed I/O-Automata. *Electronic Notes in Theoretical Computer Science* 111 (2004), 3–20. https://doi.org/10.1016/j.entcs.2004.01.001

[2] Cyrille Artho, Klaus Havelund, and Armin Biere. 2012. Using Stubs to Improve Model-Based Testing of Concurrent Programs. In *Proceedings of the 7th Workshop on Model-Based Testing (MBT)*. ACM, 1–12. https://doi.org/10.1145/2103222.2103223

[3] Axel Belinfante. 2010. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 6015)*. Springer, 266–270. https://doi.org/10.1007/978-3-642-12002-2_21

[4] Liang Cao. 2022. Estimating Efforts for Various Activities in Agile Software Development: An Empirical Study. *IEEE Access* 10 (Aug. 2022), –. https://doi.org/10.1109/ACCESS.2022.3196923

[5] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. *Formal Methods and Testing* 4949 (2008), 1–38. https://doi.org/10.1007/978-3-540-78917-8_1

[6] Jan Tretmans. 2017. On the Existence of Practical Testers. In *ModelEd, TestEd, TrustEd*, Joost-Pieter Katoen, Rom Langerak, and Arend Rensink (Eds.). Lecture Notes in Computer Science, Vol. 10500. Springer, Cham, 87–106. https://doi.org/10.1007/978-3-319-68270-9_5 Introduces TorXakis as an industrially practical tester based on LTS and ioco theory.

[7] Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-372501-1.X5000-5

[8] Muhammad Nouman Zafar, Wasif Afzal, Eduard Paul Enoiu, Athanasios Stratis, Aitor Arrieta, and Goiuria Sagardui. 2021. Model-Based Testing in Practice: An Industrial Case Study using GraphWalker. *Innovations in Software Engineering Conference (ISEC)* (2021), 1–11. https://doi.org/10.1145/3452383.3452388

AI disclosure. Text generative AI was used during the research project. The instances where it was used were only for clarifying and explaining key concepts in the initial parts of the research, and for finding synonyms or similar expressions during the documentation and writing sections of the paper. Generative or other types of AI were not used to generate sentences or paragraphs for the written section of the project.