Comparative Analysis of Model-based Testing Tools for Mobile Applications

Batuhan Sari*, Onur Kilincceker†, Serge Demeyer†
*Universiteit Antwerpen †Universiteit Antwerpen and Flanders Make

***Abstract*—Mobile application developers and testers struggle with creating efficient and comprehensive test suites that can adequately cover complex application behaviors across various scenarios and user interactions. The increasing complexity of mobile applications makes manual testing impractical while traditional automated testing often fails to systematically explore all critical paths, resulting in undiscovered defects and unreliable applications. GraphWalker and UPPAAL represent two distinct approaches to model-based testing for mobile applications, with GraphWalker offering practical path generation through directed graphs while UPPAAL provides more rigorous system analysis capabilities. Comparative analysis reveals GraphWalker's advantages for typical mobile testing scenarios due to its accessibility, straightforward model representation, and direct integration with mobile testing frameworks, making it particularly suitable for teams seeking to implement model-based testing in mobile application development workflows.**

***Index Terms*—Model-based testing, Mobile testing, GraphWalker, UPPAAL, Test automation, Path coverage**

I. Introduction

The term mobile testing refers to different types of testing, such as native mobile app testing, mobile device testing, and mobile Web app testing. We use mobile app testing to refer to "testing activities for native and Web applications on mobile devices using well-defined software test methods and tools" [1]. Manual testing of mobile applications may be a very costly activity both in terms of time and resources. The challenges intensify with the need to repeat the same tests across a large number of devices, operating systems, and execution environments. This repetitive process not only consumes significant time but also introduces inconsistencies and human errors, making manual testing unsustainable for modern software development cycles. Automation is crucial as it addresses these challenges by providing scalable, repeatable, and efficient testing solutions, thereby ensuring better reliability and performance of applications [2]. Model-based testing (MBT) has emerged as a systematic approach to address the challenges of comprehensive testing by enabling the automatic generation of test cases from abstract models that represent system behavior[3]. This approach allows testers to focus on modeling the system rather than manually designing individual test cases, potentially improving test coverage and reducing maintenance costs.

The mobile application ecosystem presents unique testing challenges due to device fragmentation, platform-specific behaviors, intermittent connectivity, and complex user interaction patterns. Traditional testing approaches often struggle to systematically cover all possible user scenarios and edge cases, leading to undiscovered defects and unpredictable application behavior. Furthermore, as mobile applications integrate with more complex backend services and implement sophisticated user interfaces, ensuring thorough testing across all possible execution paths becomes exponentially more difficult.

Despite the clear benefits of model-based testing for mobile applications, implementing an effective MBT approach requires selecting appropriate tools that balance ease of use, expressiveness, and integration capabilities. Testing teams face significant challenges when choosing between different MBT tools, each with their own modeling paradigms, strengths, and limitations. GraphWalker[4] and UPPAAL[5] represent two distinct approaches to model-based testing, with fundamental differences in their modeling capabilities, verification methods, and practical application to mobile testing scenarios.

GraphWalker employs directed graphs as its primary modeling paradigm, allowing testers to visually design models where vertices represent system states and edges represent transitions between states. While this approach is intuitive and accessible, it may lack formal verification capabilities for complex properties. Conversely, UPPAAL provides more rigorous analysis through its timed automata modeling approach but introduces a steeper learning curve and potential challenges when directly applying to typical mobile testing workflows. The existing literature lacks comprehensive comparative studies examining how these tools perform across applications of varying complexity specifically in the mobile testing domain.

The challenge extends beyond tool selection to practical implementation concerns. Mobile testing teams need clear guidance on which tool is more suitable for specific testing scenarios, how to effectively model mobile application behaviors with each tool, and what performance trade-offs to expect when generating and executing test cases. Additionally, there is limited empirical evidence regarding the effectiveness of these tools in discovering defects across applications with different levels of complexity, from simple single-screen applications to multi-functional apps with complex navigation patterns and state management.

In this paper, we evaluate the effectiveness, practicality, and performance of GraphWalker and UPPAAL for model-based testing of mobile applications across three distinct complexity levels. We systematically analyze how these tools handle increasingly complex application behaviors, providing insights into their strengths and limitations for mobile testing scenarios. As such, we make the following contributions:

1) We provide a comprehensive comparative framework for evaluating model-based testing tools specifically for mobile application testing, considering factors such as model expressiveness, test path generation capabilities, integration with mobile testing frameworks, and ease of use.
2) We present empirical results from applying both GraphWalker and UPPAAL to test three mobile applications of increasing complexity, documenting coverage metrics, and testing efficiency for each tool.
3) We offer practical guidance on modeling mobile application behaviors using both GraphWalker and UPPAAL, including strategies for addressing common mobile-specific testing challenges.
4) We analyze the performance characteristics of both tools' path generation algorithms, identifying scenarios where each tool excels or faces limitations, and proposing combined approaches that leverage the strengths of both tools for comprehensive mobile application testing.

Section II gives an overview of the state of the art in model-based testing tools, particularly GraphWalker and UPPAAL. Section III describes the case study set-up, which naturally leads to Section IV reporting the results. Section V enumerates the threats to validity to conclude the paper in Section VI.

II. Related Work

Model-based testing (MBT) has emerged as a significant approach in software testing, using models to represent the expected behavior of systems under test. This section reviews relevant literature on model-based testing tools, particularly GraphWalker and UPPAAL, along with research on their optimization, integration, and industrial applications.

A. Classification of Model-Based Testing Approaches

Within model-based testing, several distinct approaches exist with varying characteristics and capabilities.

- **Finite State Machine-based**: Tools like GraphWalker use directed graphs where nodes represent states and edges represent transitions. These are typically intuitive to use and visualize but may lack support for complex temporal properties.
- **Timed Automata-based**: Tools like UPPAAL extend finite state machines with clock variables, enabling the specification of timing constraints. This makes them suitable for testing time-critical behaviors but increases model complexity.
- **Extended Finite State Machine-based**: Tools like Spec Explorer[6] incorporate data variables into state models, enabling more expressive state representation but increasing state space complexity.
- **UML-based**: Tools like JUMBL[7] use UML state diagrams or activity diagrams for test generation, benefiting from standardized notation but sometimes lacking formal verification capabilities.
- **Markov Chain-based**: Tools like MaTeLo[8] use probabilistic models to generate tests based on usage patterns, useful for reliability testing but less suitable for exhaustive testing.

B. Positioning of GraphWalker and UPPAAL in the MBT Landscape

Our selection of GraphWalker and UPPAAL for this comparative study is strategically motivated by their representative positions in the MBT landscape and is supported by several literature-based considerations:

- **Representative of Different Modeling Paradigms**: GraphWalker represents the FSM-based approach that is widely utilized in industry practice, while UPPAAL represents the more formal timed-automata approach commonly used in academic research [9]. Together, they span the practical-formal spectrum of model-based testing approaches.
- **Maturity and Active Development**: Both tools are mature and actively maintained, with GraphWalker being used in industrial contexts and UPPAAL being a well-established tool in the formal methods community with numerous case studies [10].
- **Complementary Capabilities**: Recent comparative studies on model-based testing tools [11], [12] have identified GraphWalker as excelling in usability and integration, while UPPAAL offers stronger formal verification capabilities. This makes them excellent candidates for exploring the trade-offs between ease of use and analytical power.
- **Application to Mobile Testing**: Both tools have been applied to mobile application in the past, demonstrating their potential suitability for this domain while leaving open questions about their comparative effectiveness specifically for mobile applications.

C. Model-Based Testing Tools and Optimization

GraphWalker has established itself as a widespread open-source MBT tool that generates executable test cases from graph models of systems under test. Koroglu et al. conducted extensive experiments to evaluate GraphWalker's performance across three realistic systems [13]. Their findings revealed critical limitations: GraphWalker test cases exhibit high redundancy, limited edge pair coverage, and require significantly longer test sequences to improve coverage. Their work establishes an important baseline against which future optimization-based algorithms can be compared, highlighting that GraphWalker currently implements only two random test generation algorithms with no optimization-based approaches.

Several researchers have addressed optimization challenges in model-based testing. Silistre et al. proposed a model reduction technique applicable to GraphWalker that employs community detection algorithms to divide large models into manageable components, thereby addressing scalability concerns [14]. While this approach effectively reduces model complexity, it does not optimize the generated test cases themselves. Complementing this research, Belli and Gökçe

presented a test optimization technique based on coverage metrics that utilizes a heuristic derived from the minimal spanning set concept to reduce test suite size [15]. This approach could potentially enhance GraphWalker's test generation capabilities, though it has yet to be fully integrated into the tool.

D. Integration of Testing and Formal Verification

The combination of model-based testing with formal verification techniques has gained attention for its potential to improve test quality. Tiwari et al. proposed an innovative hybrid approach that integrates GraphWalker with UPPAAL by transforming GraphWalker models into UPPAAL timed automata [16]. This integration enables automatic verification of critical properties such as reachability and deadlock freedom, allowing testers to improve test models before generating and executing tests on actual systems. Their work demonstrates how combining dynamic testing with static analysis can enhance the testing process, though applicability and scalability challenges remain.

Nielsen explored broader methods for combining model-based analysis and testing to improve system quality, particularly for embedded systems [17]. While acknowledging the difficulty of defining universally applicable methods for industrial settings, Nielsen's work suggests promising directions for integrating these complementary approaches. Similarly, Marinescu et al. described the verification of architectural models using the UPPAAL model checker [18], introducing a framework called ViTAL that captures models as timed automata and verifies them using UPPAAL. Their approach supports generating queries to verify reachability and liveness properties, demonstrating UPPAAL's capabilities for formal verification of complex systems.

E. Industrial Applications and Empirical Studies

Moving from theoretical approaches to practical applications, Zafar et al. reported on the application of GraphWalker in an industrial setting, specifically testing a Train Control Management System at Bombardier Transportation in Sweden [19]. Their case study provides valuable insights into the real-world efficacy of model-based testing using GraphWalker. They found that models created using both requirements and test specifications provided better understanding from a tester's perspective compared to models based solely on requirements. Furthermore, the model-based test cases generated were longer in terms of test steps but achieved better edge coverage and could cover requirements in different execution orders while maintaining equivalent requirements coverage compared to manually created test cases.

Despite these advances in research, significant gaps remain in comparative studies of different MBT tools, particularly in the context of mobile application testing with varying complexity levels. The unique challenges of mobile testing—including device fragmentation, diverse interaction patterns, and platform-specific behaviors—warrant dedicated investigation into how tools like GraphWalker and UPPAAL perform across different application complexities and testing scenarios. This gap forms the foundation for our current research, which seeks to systematically compare these tools across multiple dimensions when applied to mobile applications of varying complexity.

III. Case Study Set Up

In this study, we conduct a comprehensive comparison of GraphWalker and UPPAAL for model-based testing of mobile applications. To ensure a thorough evaluation, we design a multi-faceted case study that explores the tools' capabilities across different dimensions, including ease of use, test generation capabilities, and performance characteristics.

A. Research Questions

RQ1: *What are the key challenges in automating the testing of mobile applications, and how can they be addressed?*
Motivation. Mobile application testing presents unique challenges compared to traditional software testing due to device fragmentation, diverse user interaction patterns, platform-specific behaviors, resource constraints, and frequent updates. Understanding these challenges is crucial for developing effective testing strategies that ensure application quality across various devices and usage scenarios. Additionally, identifying common obstacles faced by testing teams can guide the development of more suitable testing tools and methods specifically designed for mobile environments.

RQ2: *How are these challenges addressed with existing solutions such as GraphWalker and UPPAAL?*
Motivation. Model-based testing tools offer promising approaches to address mobile testing challenges through automated test case generation and systematic state exploration. However, different tools employ distinct modeling paradigms and capabilities that may be more or less suitable for specific mobile testing scenarios. Understanding how GraphWalker and UPPAAL address common mobile testing challenges provides valuable insights for testing teams selecting appropriate tools and methods for their projects.

RQ3: *What are the comparative advantages and limitations of GraphWalker and UPPAAL?*
Motivation. Selecting an appropriate model-based testing tool requires understanding the trade-offs between ease of use, modeling expressiveness, performance, and integration capabilities. A detailed comparison of GraphWalker and UPPAAL across these dimensions provides testing teams with actionable insights for tool selection based on project requirements, team expertise, and application characteristics. This comparison also identifies opportunities for tool improvements and potential complementary usage of both tools.

B. Approach

Addressing **RQ1** regarding the key challenges in automating mobile application testing, our approach confronts the unique obstacles of mobile environments, including device fragmentation, complex UI interactions, and platform-specific behaviors. Our testing framework consists of six interconnected phases: Application Selection to identify varied test subjects; Model

Construction in both tools' environments; Test Generation using diverse algorithms; Test Execution via Appium; continuous Monitoring for validation; and comprehensive Result Collection and Analysis. This systematic workflow enables thorough evaluation of both GraphWalker and UPPAAL across mobile applications of varying complexity. The following sections detail our implementation approach.

1) Application Selection: To facilitate a comprehensive and realistic comparison between testing tools, we carefully selected three Android applications with varying complexity levels: Easy, Medium, and Complex. This stratified selection approach enables us to evaluate how GraphWalker and UPPAAL perform across a spectrum of application complexities.

The Easy application represents a simple utility with minimal screens and linear navigation paths, featuring straightforward UI elements and predictable state transitions. The Medium complexity application introduces a more elaborate navigation structure with multiple interconnected screens, moderate state complexity, and non-linear user flows. Finally, the Complex application embodies sophisticated behavior with numerous screens, complex state dependencies, concurrent processes, and extensive navigation possibilities.

2) Model Construction: The first phase of our approach involves creating formal models of the applications under test. For each application, we construct two separate models: one for GraphWalker and one for UPPAAL.

For GraphWalker, we leverage its web-based modeling environment, GraphWalker Studio, to construct directed graph models. Each model consists of vertices representing application states (such as screens or significant UI states) and edges representing transitions between states (such as button clicks, text inputs, or navigation actions). These models are then exported in JSON format for further processing.

For UPPAAL, we construct timed automata models using the UPPAAL modeling environment. These models correspond to the same conceptual states and transitions as their GraphWalker counterparts by construction—we manually created both models with intentional structural equivalence, though necessarily expressed using UPPAAL's distinct formalism and notation.

3) Test Generation: Once the models are constructed, the next phase involves generating test paths using both tools' path generation capabilities. This phase bridges model construction and test execution, producing concrete test paths that will be executed on the target applications.

For GraphWalker, we implement a command-line interface using the GraphWalker CLI tool. This allows us to programmatically generate test paths using different algorithms and coverage criteria.

For UPPAAL, we utilize its verification capabilities to generate test paths by formulating queries that ensure complete edge coverage.

While most UPPAAL verification queries produce deterministic results, certain configurations such as random/some and random/fastest search strategies introduce non-deterministic elements in path generation. To ensure statistical validity, each test generation combination is repeated 30 times, producing 30 potential paths for each tool, algorithm, and application combination. From these, we select the median path based on length and coverage characteristics for further execution and evaluation.

4) Test Execution with Appium: After generating the test paths, we utilize Appium [20], an industry-standard open-source test automation framework for mobile applications. Appium provides a robust client-server architecture that enables automated interaction with mobile applications without requiring code instrumentation or modification of the applications themselves.

Figure 1 presents an overview of our testing approach with Appium.

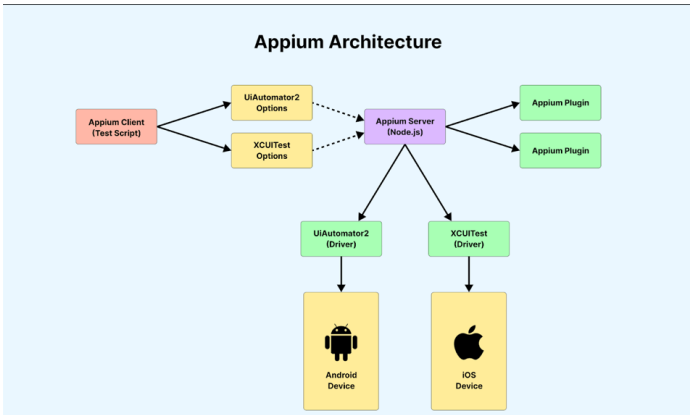

Fig. 1. An overview of the mobile testing implementation using Appium.

Our Appium test implementation translates the abstract test paths generated by GraphWalker and UPPAAL into concrete UI interactions that can be executed on the Android emulator. As illustrated in Figure 1, our testing infrastructure utilizes the Appium Client (Test Script) to communicate with the Appium Server (Node.js), which in turn interfaces with platform-specific drivers.

- Establishes a session with the Appium Server through the Appium Client, configuring UIAutomator2 Options for Android testing
- Locates the appropriate UI elements using Appium's element identification strategies (ID, XPath, accessibility ID, etc.)
- Determines the required action based on the current step in the test path
- Uses Appium's WebDriver commands to execute the action (tap, swipe, text input, etc.) on the Android Device through the UIAutomator2 Driver chain)
- Implements appropriate wait conditions to ensure the action completes successfully
- Verifies the transition to the expected next state before proceeding to the next step

This approach ensures reliable and reproducible test execution, with Appium handling the complexities of cross-platform UI interaction and synchronization. We use Java with

Appium's Java client library to communicate with the Appium server, which in turn interacts with the mobile application under test.

5) Monitoring: During test execution, we perform monitoring to ensure reliable results. This monitoring observes application behavior and test logs at UI-event granularity, where we manually analyze how each Appium UI event corresponds to states and transitions in our GraphWalker and UPPAAL models. This observation is crucial for detecting any discrepancies between the expected behavior (as modeled) and the actual application behavior.

6) Results Collection and Analysis: The final phase of our approach involves collecting and analyzing the test results to compare the effectiveness and efficiency of GraphWalker and UPPAAL. We generate detailed test reports containing test path metrics such as length and execution times.

These reports are stored in a structured format to facilitate systematic analysis between the different tools, algorithms, and applications. Our analysis focuses on key performance indicators that reveal the strengths and limitations of each approach across varying application complexity levels.

C. Experimental Setup

To assess the performance and reliability of our comparative approach between GraphWalker and UPPAAL, we conducted a series of controlled experiments. Our investigation was guided by the research questions outlined in Section III. The experiments were designed to provide comprehensive insights into how these model-based testing tools perform across different application complexities, focusing on both test generation capabilities and execution efficiency.

1) Testing Environment: The experimental environment for evaluating both model-based testing tools involved a consistent hardware and software setup to ensure fair comparison. All experiments were conducted using Windows 11, with hardware specifications including an AMD Ryzen 5 5600H CPU, NVIDIA RTX 3060 Laptop GPU, 16 GB of RAM, and 512 GB SSD. Tests were executed on a Pixel 6 API 31 (Android 12) emulator on Android Studio to ensure consistency across all test runs. Using a standardized emulator configuration allowed us to control for device-specific variables that might otherwise impact the testing process, providing a more reliable basis for comparing the performance of the two model-based testing approaches.

For test automation and execution, we set up Appium 2.17.0 as our interface between the test scripts and mobile applications. The Appium server was installed and configured on the same machine running the Android emulator, with Node.js as its runtime environment. We utilized the UiAutomator2 driver for Android automation, which offers enhanced stability and performance compared to earlier drivers. The development and execution environment used OpenJDK version 17.0.14. To ensure reliable test execution, we configured appropriate implicit and explicit wait strategies in our Appium settings, accommodating for variations in application loading times and UI element rendering. This Appium setup provided a robust platform-agnostic layer that allowed us to execute the test paths generated by both GraphWalker and UPPAAL without requiring modifications to the applications under test.

2) Test Applications: To comprehensively evaluate the tools across varying levels of complexity, we selected three mobile applications with distinct characteristics and user interaction patterns:

Dictionary (Low complexity): A simple application with 5 states and 5 transitions in its model, representing basic interaction patterns. This application focuses on straightforward text input and lookup functionality, providing a baseline for evaluating the tools' performance on simple interaction flows.

News (Medium complexity): A more sophisticated application with 17 states and 25 transitions, featuring moderate navigation complexity and data processing. This application introduces more complex UI interactions, including scrolling lists.

FriendZone (High complexity): A complex social media application with 41 states and 61 transitions, implementing advanced features such as user authentication, profile management, content posting, comments, likes, and multi-level navigation patterns. This application represents the high complexity typical in modern mobile applications, with rich state management and diverse user interaction possibilities. All applications were sourced from an open-source repository [7] to ensure reproducibility of our study. The selection of these three applications with increasing complexity levels enables us to analyze how each testing tool scales with application complexity and to identify potential strengths and limitations that may only become apparent at certain complexity thresholds.

TABLE I
DESCRIPTIVE MODEL STATISTICS FOR TEST APPLICATIONS

Application	States (Vertices)	Transitions (Edges)	Model Size
Dictionary	5	5	10
News	17	25	42
FriendZone	41	61	102

3) Model Construction and Test Generation: Models for each application were developed using their respective native environments, taking care to accurately represent all significant states and transitions.

GraphWalker Test Generation: We utilized six different test generation algorithms via CLI commands in Visual Studio Code, systematically varying both the algorithm type and coverage criteria.

TABLE II
GRAPHWALKER TEST GENERATION CONFIGURATIONS

Algorithm	Coverage Target
random	edge_coverage(50%)
random	edge_coverage(70%)
random	edge_coverage(100%)
quick_random	edge_coverage(50%)
quick_random	edge_coverage(70%)
quick_random	edge_coverage(100%)

These variations allowed us to analyze how different coverage targets and algorithm types affect the length, efficiency, and effectiveness of the generated test paths. The random algorithm employs a truly random approach to path generation, while quick_random uses a more directed strategy to achieve coverage objectives more efficiently.

UPPAAL Test Generation: We employed nine different test generation strategies through the UPPAAL UI, as CLI commands were not available for UPPAAL. These strategies combined different search methods with trace preferences:

TABLE III
UPPAAL TEST GENERATION CONFIGURATIONS

Search Strategy	Trace Option	Coverage Criterion
Breadth	Some	Edge Coverage 100%
Breadth	Fastest	Edge Coverage 100%
Breadth	Shortest	Edge Coverage 100%
Depth	Some	Edge Coverage 100%
Depth	Fastest	Edge Coverage 100%
Depth	Shortest	Edge Coverage 100%
Random	Some	Edge Coverage 100%
Random	Fastest	Edge Coverage 100%
Random	Shortest	Edge Coverage 100%

For all UPPAAL test generations, we utilized the following query to ensure complete edge coverage:

```
E<> forall(i : int[0, NUM_EDGES-1]) visited[i]
```

This query instructs the model checker to find an execution path (E<>) where all edges have been visited (forall i, visited[i] is true), effectively directing UPPAAL to continue exploration until all transitions have been traversed. This approach ensures that the generated test paths achieve 100% edge coverage, providing a fair comparison with the GraphWalker paths targeting 100% edge coverage.

The diversity of test generation approaches allows us to comprehensively evaluate how different path generation algorithms impact key metrics such as path length, execution time, and fault detection capability across applications of varying complexity.

4) Test Case Implementation: Once the test paths were generated from both tools, we implemented the concrete test cases using Appium's Java client library. For each application, we created a structured test framework following the Page Object Model design pattern to improve maintainability and readability of test code.

For each application state identified in our models, we created corresponding Page classes that encapsulated the UI elements and interactions specific to that state. These Page classes contained methods to locate UI elements using Appium's element finding strategies (ID, XPath, accessibility ID, etc.), perform interactions with these elements (tap, swipe, text input, etc.), and verify state-specific conditions through assertions.

Transitions between states were implemented as methods that triggered the appropriate UI events. For example, a transition representing a button click was implemented as a method that located the button element using Appium's locator strategies and then performed a click action using Appium's TouchAction API. This approach ensured clean separation between the abstract model transitions and their concrete implementation in the mobile application.

For state verification, we implemented comprehensive assertions that checked various aspects of the application state, including the presence of expected UI elements, correct text content in labels and fields, appropriate visibility of conditional UI elements, and valid application responses to user inputs. These assertions were crucial for confirming that the application behavior matched our model expectations.

Any discrepancies between expected and actual behavior were recorded as test failures, providing valuable information about potential model inaccuracies or application defects.

Each generated test path was translated into a sequence of these state transitions and verifications, creating executable test cases that followed the paths specified by GraphWalker and UPPAAL. This approach allowed us to maintain a clear separation between the abstract test models and the concrete test implementation, while ensuring thorough validation of application behavior.

5) Test Execution and Data Collection: Each test path generation was repeated 30 times per tool per application to account for variability in the generation algorithms. From these 30 generations, we selected the median path for each tool-application combination. This selected median path was then executed 10 times on the target application to collect reliable execution time data.The median execution time from these 10 runs was recorded as the final performance metric.

All tests were conducted as offline tests using predefined paths to ensure consistency in the execution environment and eliminate network-related variables.

IV. RESULTS AND DISCUSSION

This section presents the empirical results from our comparative study of GraphWalker and UPPAAL for mobile application testing. We discuss the performance of both tools across different applications and algorithm configurations, analyzing test generation efficiency, coverage characteristics, and execution metrics.

A. Ease of Use and Practical Considerations

This section addresses our research question **RQ2** regarding how GraphWalker and UPPAAL address the challenges of mobile application testing. GraphWalker offers a more intuitive and user-friendly interface that requires minimal learning curve. Its web-based modeling environment eliminates the need for installation, allowing testers to quickly create and modify models through a browser. A particularly valuable feature is GraphWalker's "Shared Node" capability, which enables complex models to be decomposed into manageable components, significantly simplifying the construction and maintenance of sophisticated test models.

The test generation process in GraphWalker is straightforward, requiring only a brief specification such as `random(vertex_coverage(100))` to define both the

path selection algorithm and coverage criteria. This simple syntax makes it accessible to testers with limited programming experience while still offering considerable flexibility through multiple generators and coverage options. Additionally, GraphWalker supports both online and offline testing modes with minimal configuration, enhancing its versatility across different testing environments.

In contrast, UPPAAL presents a more complex interface with a steeper learning curve. While powerful, its modeling environment requires more time to master and lacks the "Shared Node" functionality that facilitates management of complex models. This limitation can make large model creation and maintenance more challenging, particularly for testers new to model-based testing.

UPPAAL's test generation configuration requires writing specific queries, especially for coverage-based testing. While the tool includes some built-in options, generating custom test paths often involves crafting queries that some testers may find challenging to develop and debug. We also observed occasional performance degradation in the UPPAAL interface during test generation for complex models, which could potentially impact workflow efficiency in time-sensitive testing environments.

These usability differences complement our performance findings by highlighting that while UPPAAL may offer superior performance in terms of path efficiency and execution time, GraphWalker provides advantages in ease of use and model management that may be equally important considerations depending on team expertise and testing objectives.

B. Test Path Generation Analysis

For the simple model Dictionary, GraphWalker shows identical step counts between random and quick_random algorithms at equivalent coverage levels, ranging from 7 steps (50% coverage) to 11 steps (100% coverage). Execution times scale linearly with coverage, from 11 seconds (50%) to approximately 14 seconds (100%). UPPAAL consistently generates paths of exactly 11 steps across all nine configurations with execution times between 13.3-14.5 seconds, showing minimal variation. The consistency of results and negligible performance differences between configurations suggest that for very simple applications, algorithm selection has limited impact on test efficiency.

For the deeper analysis, Figure 2 presents the step count metrics for GraphWalker's random and quick_random algorithms across different coverage criteria.

As shown in Figure 2, GraphWalker's path generation for the News application demonstrates substantial variation based on both algorithm selection and coverage targets. The random edge coverage algorithm (R EC) generates significantly longer paths than quick_random (QR EC) for equivalent coverage goals, with R EC-100% producing paths more than twice as long as QR EC-100%. This indicates that quick_random's more directed approach to edge traversal produces more efficient paths while achieving equivalent coverage.

For the News application (Figure 3), UPPAAL demonstrates significant variations in test path length across different search and trace configurations. Most search strategies with the "Shortest" trace option and "Breadth/Some" consistently generate concise paths of approximately 52 steps. In contrast, Random/Some and Random/Fastest configurations produce substantially longer paths, with median values around 71-72 steps and outliers exceeding 100 steps.

Figure 4 reveals how application complexity affects GraphWalker's performance. For the more complex FriendZone application, the disparities between algorithms become even more pronounced. The random algorithm targeting 100% edge coverage generates extraordinarily long paths (median of approximately 900 steps) with extreme outliers exceeding 2500 steps. In contrast, quick_random maintains relatively consistent path lengths across applications, with QR EC-100% generating paths around 250 steps for the FriendZone application. This suggests that quick_random scales more effectively with application complexity, making it potentially more suitable for testing complex mobile applications.

When examining Figure 5, showing step counts for the FriendZone application, the Random/Some configuration shows dramatically increased path lengths, with a median exceeding 175 steps and outliers approaching 215 steps. This represents more than double the step count compared to the same configuration on the simpler News application, indicating that UPPAAL's Random search algorithm with "Some" trace option scales poorly with application complexity.

In contrast, the Breadth/Shortest, Random/Shortest and Depth/Shortest configurations maintain relatively stable path lengths even with increased application complexity, consistently generating paths of approximately 134 steps.

C. Execution Time Analysis

The execution time metrics presented here reflect the performance of the median test paths generated by each algorithm configuration. For each tool and algorithm combination, we selected the path with the median step count from our 30 test generations and executed it 10 times on the target application, recording the execution times.

For the News application (Figure 6), the execution times of GraphWalker's median paths show a strong correlation with their respective step counts. The median path from the random edge coverage algorithm with 100% coverage (R EC-100%) requires approximately 180 seconds of execution time, significantly higher than all other configurations. Quick_random consistently produces more efficient median paths, with QR EC-50% requiring only about 30 seconds and QR EC-100% taking approximately 80 seconds – less than half the time of R EC-100%. This demonstrates that quick_random's more efficient path generation directly translates into faster test execution, a critical consideration for continuous integration environments.

UPPAAL's execution time performance for the News application (Figure 7) reveals distinctive patterns based primarily on trace options rather than search algorithms. The median

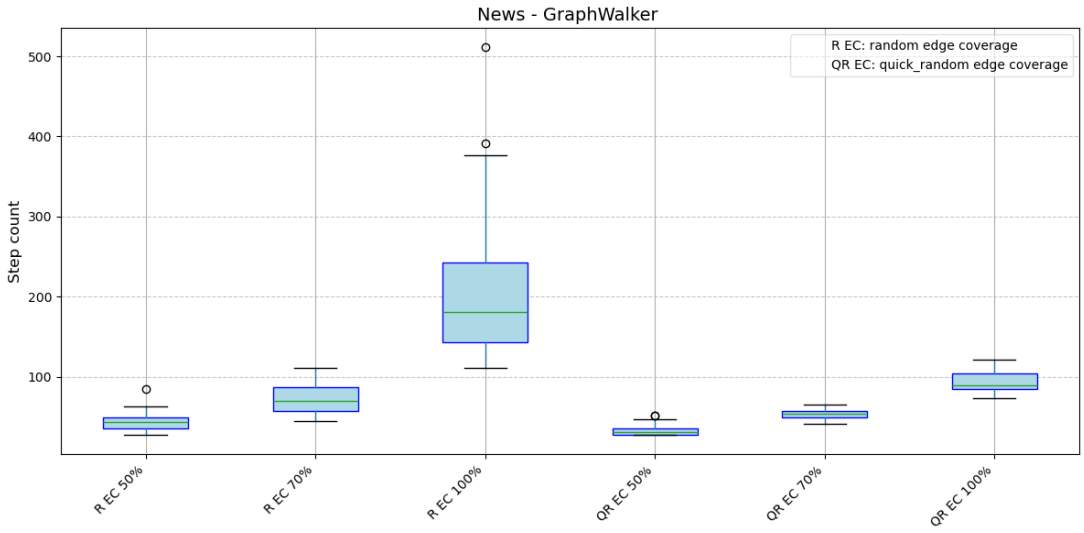

Fig. 2. Step count comparison for News application across different GraphWalker algorithms and coverage criteria

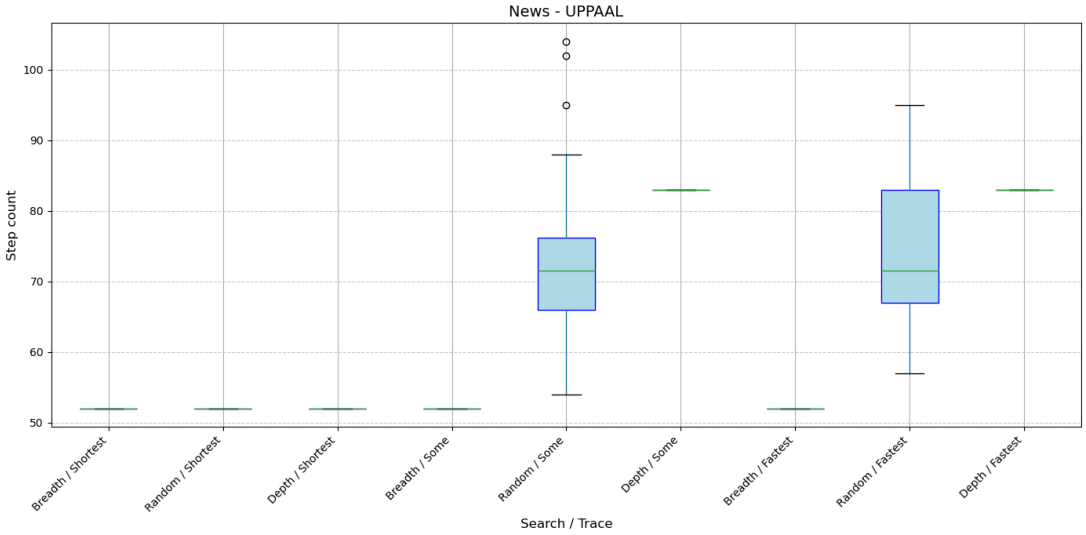

Fig. 3. Step count comparison for News application across different UPPAAL search and trace strategies

paths from all configurations using the "Shortest" trace option maintain relatively consistent execution times between 60-70 seconds, regardless of the search strategy used. In contrast, median paths from Random/Some and Random/Fastest configurations require substantially more time (110-115 seconds), with Depth/Some falling in between at approximately 90 seconds. This suggests that for the News application, the trace option selection has a greater impact on execution efficiency than the search algorithm itself.

When testing the more complex FriendZone application (Figure 8), the execution time differences between median paths become even more dramatic. The median path from R EC-100% requires approximately 900 seconds (15 minutes) to execute, representing a substantial testing overhead. In contrast, the median path from QR EC-100% completes in around 210 seconds, making it over four times more efficient while achieving the same coverage. The QR EC-50% configuration offers the fastest execution at approximately 90 seconds, suggesting that for complex applications, strategic trade-offs between coverage and execution efficiency may be necessary depending on testing constraints.

For the FriendZone application (Figure 9), execution times for all median paths increase, but with varying degrees. The median path from Random/Some requires approximately 180 seconds with outliers exceeding 200 seconds. Random/Fastest follows at around 155 seconds, while most other configurations cluster between 130-140 seconds. Notably, the median paths from "Shortest" trace options remain the most efficient across all search strategies, with execution times consistently below 140 seconds. This reinforces that for complex applications,

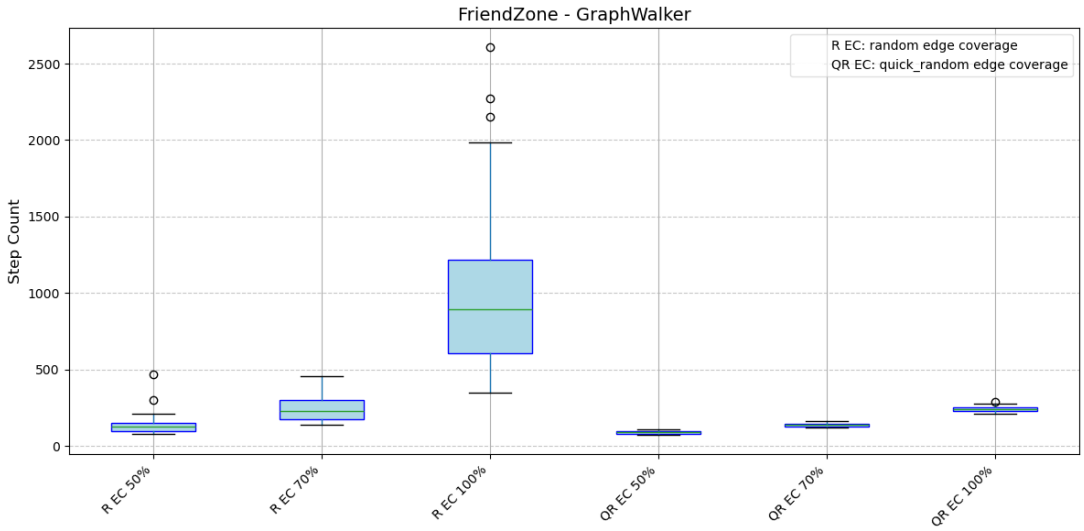

Fig. 4. Step count comparison for FriendZone application across different GraphWalker algorithms and coverage criteria

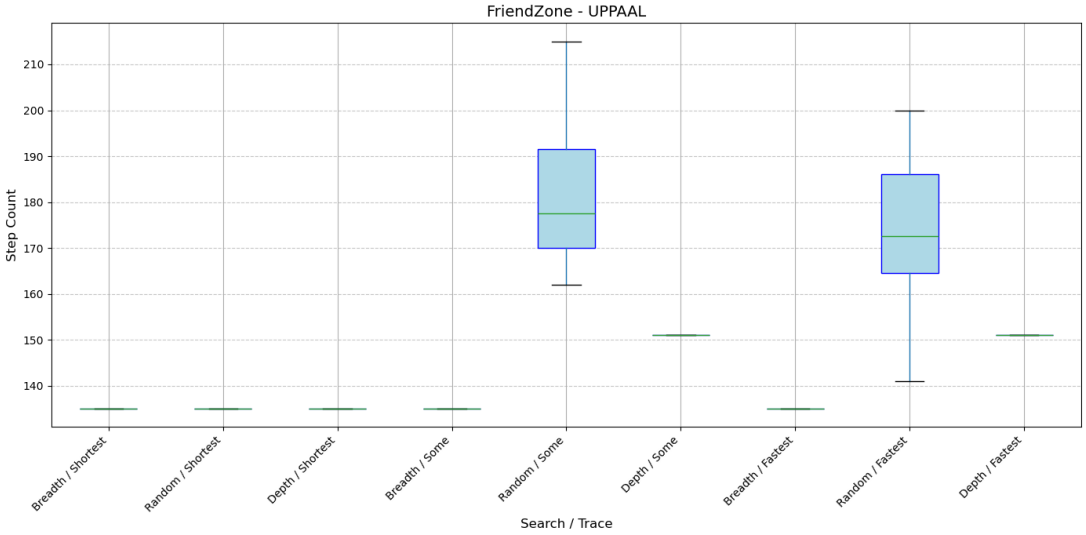

Fig. 5. Step count comparison for FriendZone application across different UPPAAL search and trace strategies

UPPAAL's trace option selection significantly influences execution efficiency, with "Shortest" consistently providing the best performance regardless of the associated search algorithm.

D. Comparative Step Count Analysis of GraphWalker and UPPAAL Algorithms

To provide a more focused comparison between similar algorithms, we analyzed GraphWalker's quick_random edge coverage (100%) against UPPAAL's Random algorithm with its three different trace options. This comparison is particularly valuable as these algorithms represent the most directly comparable approaches between the two tools.

Figure 10 presents the step count comparison for the News application. GraphWalker's QR EC 100% generates significantly longer test paths with a mean of 93.3 steps and considerable variability (ranging from approximately 73 to 121 steps). In contrast, UPPAAL's Random/Shortest produces remarkably consistent paths at exactly 52 steps across all test generations, representing a 44% reduction in path length compared to GraphWalker. UPPAAL's Random/Some and Random/Fastest algorithms show intermediate results with means of 72.1 and 73.9 steps respectively, both exhibiting moderate variability but still producing more efficient paths than GraphWalker's quick_random approach.

The differences become even more pronounced for the more complex FriendZone application (Figure 11). GraphWalker's QR EC 100% generates substantially longer paths with a mean of 244.5 steps and a wide range (210 to 290 steps). UPPAAL's Random/Shortest again demonstrates superior efficiency with a mean of just 135 steps and minimal variation, representing

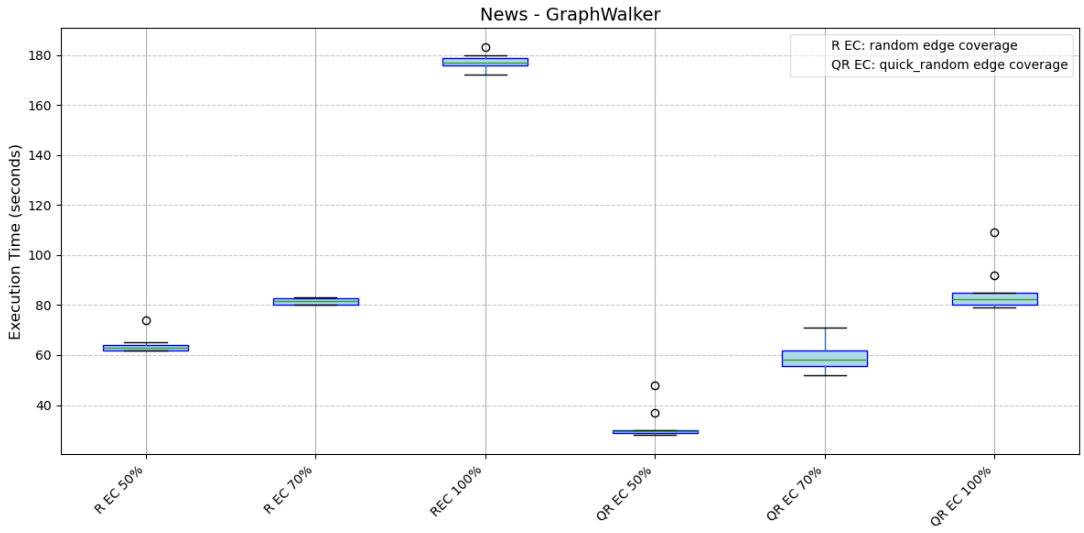

Fig. 6. Execution time comparison for News application using the median test paths from different GraphWalker algorithms and coverage criteria

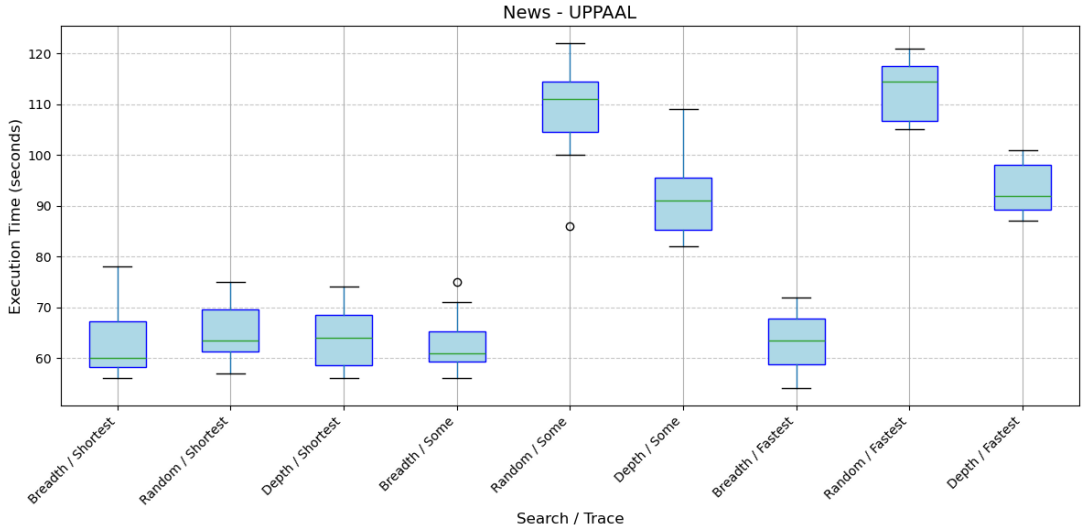

Fig. 7. Execution time comparison for News application using the median test paths from different UPPAAL search and trace strategies

a 45% reduction in path length. Random/Some and Random/Fastest show intermediate performance with means of 180.2 and 175.0 steps respectively.

These comparative results reveal that UPPAAL's Random algorithm, particularly with the Shortest trace option, consistently produces more efficient test paths than GraphWalker's quick_random edge coverage approach while achieving equivalent coverage.

This efficiency gap widens as application complexity increases, suggesting that UPPAAL's path generation algorithms may scale better with application complexity. The significantly lower step counts translate directly into faster test execution times, making UPPAAL potentially more suitable for testing complex applications or in environments where test execution time is a critical constraint.

The consistent performance of UPPAAL's Random/Shortest algorithm across multiple test generations also suggests greater determinism in its path generation approach, which could be beneficial for test reproducibility. Meanwhile, GraphWalker's higher variability might occasionally discover edge cases through its more diverse path generation, but at the cost of consistently higher execution times.

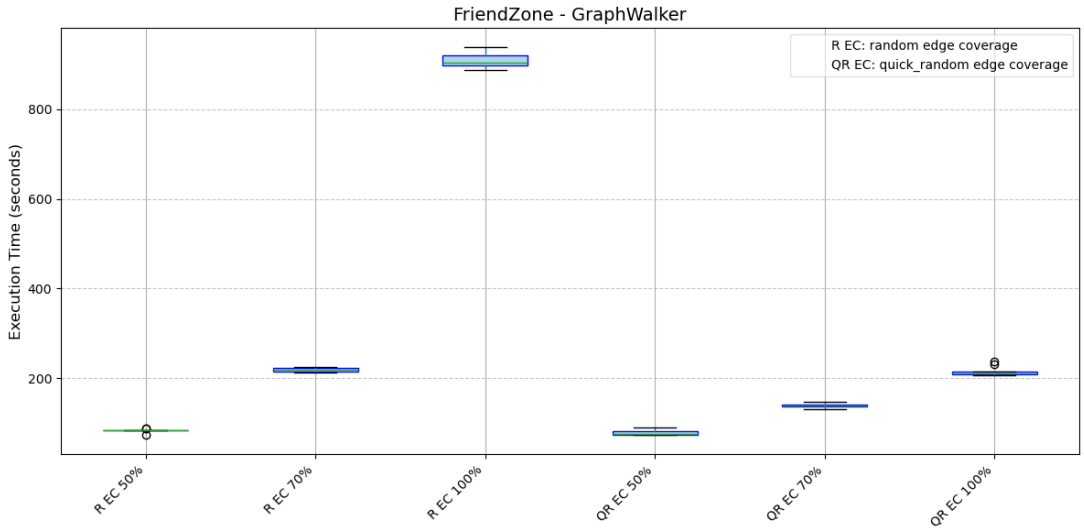

Fig. 8. Execution time comparison for FriendZone application using the median test paths from different GraphWalker algorithms and coverage criteria

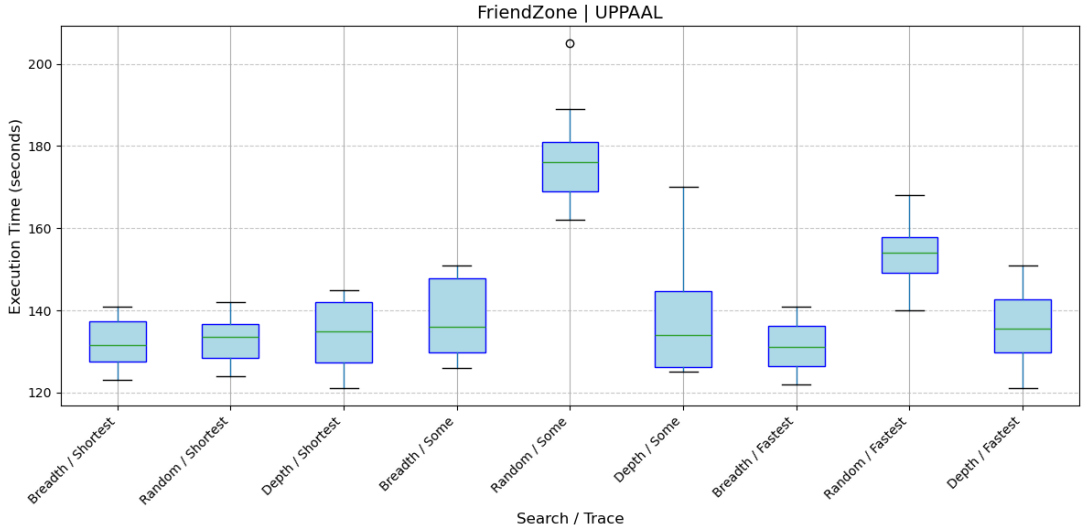

Fig. 9. Execution time comparison for FriendZone application using the median test paths from different UPPAAL search and trace strategies

E. Comparative Execution Time Analysis Between GraphWalker and UPPAAL

To directly compare execution performance between the most comparable algorithms from both tools, we analyzed execution times for GraphWalker's QR EC 100% against UPPAAL's three Random trace variants across both test applications.

Figure 12 presents execution time results for the News application. UPPAAL's Random/Shortest configuration demonstrates superior performance with a mean execution time of just 65.1 seconds and remarkably consistent results (limited variance). GraphWalker's QR EC 100% shows moderate performance with a mean of 85.4 seconds, representing a 31% increase over Random/Shortest. UPPAAL's other trace options perform significantly worse, with Random/Some and Random/Fastest averaging 108.8 and 112.9 seconds respectively - approximately 73% longer than Random/Shortest.

The tight interquartile range of Random/Shortest indicates high consistency in execution times, an important factor for predictable testing workflows. GraphWalker also shows reasonable consistency but with notable outliers extending up to 109 seconds.

For the more complex FriendZone application (Figure 13), the performance differences become more pronounced. UPPAAL's Random/Shortest remains the most efficient with a mean execution time of 133.0 seconds. GraphWalker's QR EC 100% exhibits the least efficient performance at 215.2 seconds - representing a 62% increase over Random/Shortest. UPPAAL's Random/Fastest (153.5 seconds) and Random/Some (177.5 seconds) occupy intermediate positions.

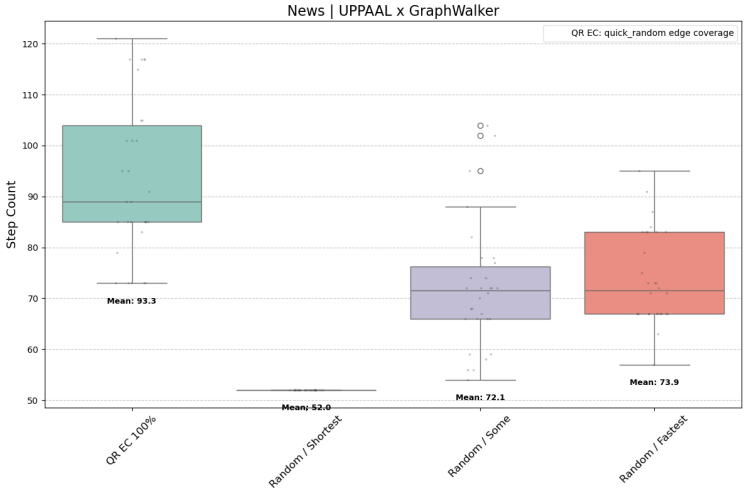

Fig. 10. Step count comparison between GraphWalker QR EC 100% and UPPAAL's Random algorithm variants for News application

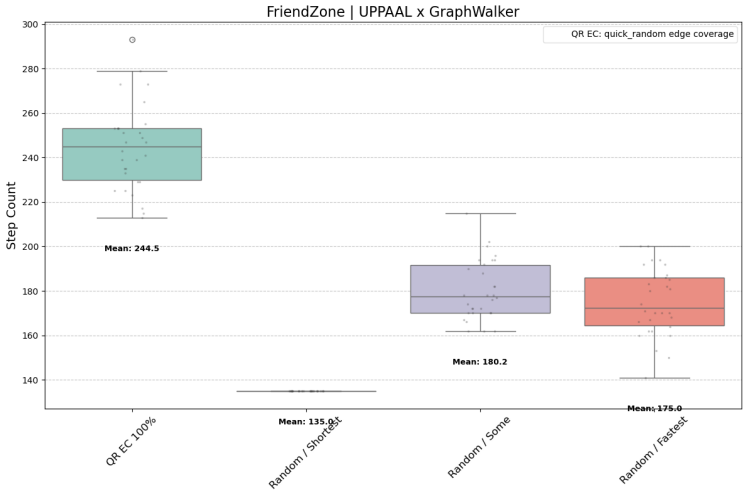

Fig. 11. Step count comparison between GraphWalker QR EC 100% and UPPAAL's Random algorithm variants for FriendZone application

Notably, GraphWalker's execution time increases by 152% when moving from News to FriendZone, while UPPAAL's Random/Shortest increases by only 104%. This differential scaling indicates that UPPAAL's Random/Shortest configuration demonstrates better scalability with increasing application complexity.

These results demonstrate that UPPAAL's Random/Shortest configuration consistently outperforms GraphWalker's QR EC 100% across both applications, with the performance gap widening as application complexity increases. The superior execution efficiency of UPPAAL's Random/Shortest, combined with its consistent performance and better scalability, makes it particularly suitable for testing complex mobile applications where execution time is a critical factor.

F. Comparative Analysis Between Tools

Directly addressing **RQ3** ("What are the comparative advantages and limitations of GraphWalker and UPPAAL?"), this subsection synthesizes our empirical findings to provide a comprehensive comparison between the two tools. Our analysis across applications of varying complexity reveals several important distinctions and complementary strengths:

Path Efficiency: UPPAAL's Random/Shortest configuration consistently generates the most efficient paths across all tested configurations, with approximately 44-45% fewer steps than GraphWalker's quick_random algorithm at equivalent coverage levels. Among GraphWalker's options, quick_random generates significantly more efficient paths than random, particularly at higher coverage levels.

Execution Performance: UPPAAL's Random/Shortest demonstrates superior execution efficiency, completing test paths 31-62% faster than GraphWalker's quick_random depending on application complexity. This efficiency advantage becomes more pronounced with more complex applications, making Random/Shortest particularly valuable for testing sophisticated mobile applications.

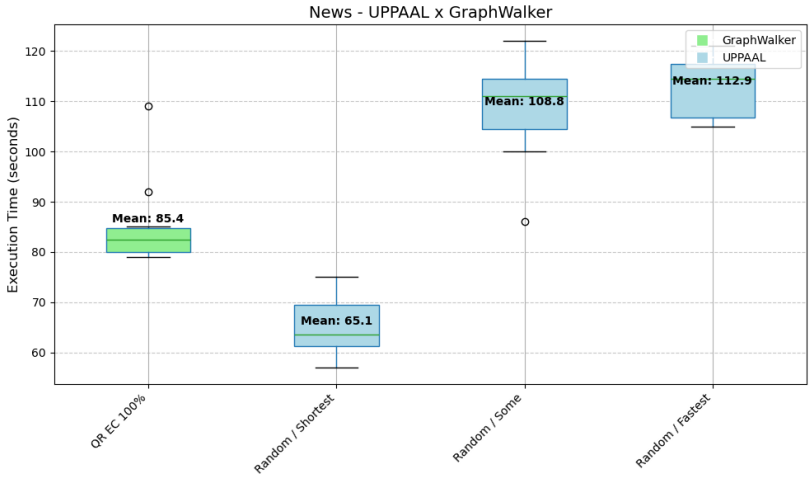

Fig. 12. Execution time comparison between GraphWalker QR EC 100% and UPPAAL's Random algorithm variants for News application

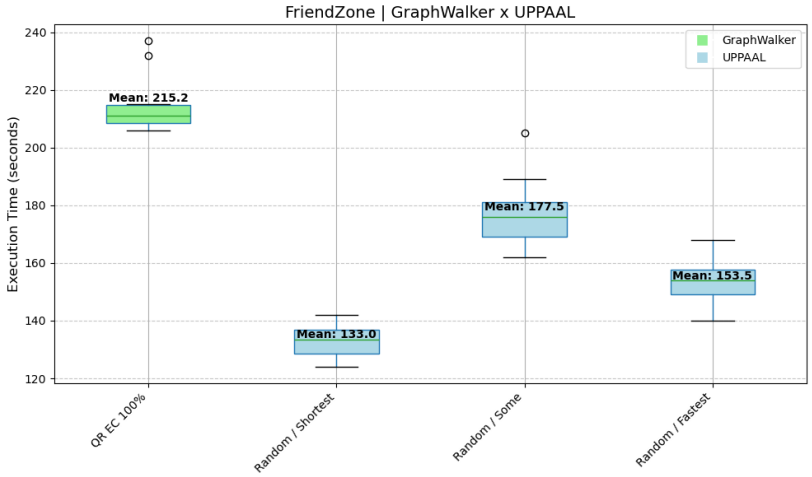

Fig. 13. Execution time comparison between GraphWalker QR EC 100% and UPPAAL's Random algorithm variants for FriendZone application

Scalability: Both tools show performance degradation with increased application complexity, but UPPAAL's Random/Shortest scales more effectively. When moving from News to FriendZone, UPPAAL's Random/Shortest execution time increases by 104% compared to GraphWalker's 152% increase. GraphWalker's random algorithm demonstrates particularly poor scalability, with exponential increases in path length for complex applications.

Consistency: UPPAAL's Random/Shortest configuration exhibits remarkable consistency across test generations, producing identical path lengths with minimal execution time variance. This predictability is beneficial for creating reliable test schedules in continuous integration environments. GraphWalker, while reasonably consistent with quick_random, shows higher variability in both path generation and execution times.

Simple Application Performance: For very simple applications like Dictionary, both tools perform comparably with minimal differences in path length and execution time, suggesting that tool selection for simple applications may depend more on integration requirements than performance considerations.

V. Threats to Validity

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. We organise them into four categories.

Construct validity: Our initial experimental design included the comparison of test generation times between GraphWalker and UPPAAL as an additional performance metric. However, we were unable to collect this data from UPPAAL as the application does not provide programmatic access to generation time metrics. This limitation restricted our comparison to step count and execution time metrics only. While these metrics provide significant insights into tool efficiency, the absence of

generation time data means we could not evaluate the full test generation lifecycle performance.

Internal validity: The randomness inherent in stochastic test generation algorithms presents a potential threat to internal validity. To mitigate this threat, we conducted 30 separate test generations for each algorithm configuration and measured execution times across 10 runs for each median path.

External validity: Our study mainly focuses on mobile applications where the main logic is handled on the front-end. The three applications we tested represent this kind of app, where most user interactions and state transitions happen within the app itself. However, we did not include other types of apps that come with different testing challenges. For example, thin-client apps often rely on backend systems to control navigation, while highly interactive apps such as mobile games have UI states that change quickly based on user behavior. These categories present distinct state exploration challenges that could affect GraphWalker and UPPAAL performance differently than observed in our test applications. Further studies focusing on these specific types of applications would complement our findings.

Reliability: To mitigate reliability concerns related to application-specific results, we deliberately included three applications with varying complexity levels: Dictionary (simple), News (moderate), and FriendZone (complex). This diversity in application complexity helps ensure that our findings are not overly influenced by characteristics specific to a single application.

VI. Conclusion

Our comparative analysis of GraphWalker and UPPAAL for mobile application testing reveals that both tools offer distinct advantages in different contexts. UPPAAL's Random/Shortest configuration consistently demonstrates superior performance metrics, generating test paths with 44-45% fewer steps and executing 31-62% faster than GraphWalker's quick_random algorithm, with this advantage becoming more pronounced for complex applications. However, GraphWalker provides significant usability benefits through its intuitive interface, web-based modeling environment, and shared node capability that simplifies the management of complex models. For simple applications, performance differences between the tools are negligible, while for complex applications, UPPAAL's better scalability and consistency become valuable assets. The optimal choice between these tools ultimately depends on the specific testing context, balancing performance considerations against ease of use and team expertise.

References

[1] Gao, Jerry, et al. "Mobile application testing: a tutorial." Computer 47.2 (2014): 46-55.

[2] Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., & Lo, D. (2015, April). Understanding the test automation culture of app developers. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST) (pp. 1-10). IEEE.

[3]] Sinha, A. and C. Smidts, Hottest: A model-based test design technique for enhanced testing of domainspecific applications, ACM Transactions on Software Engineering and Methodology (TOSEM) 15 (2006), pp. 242–278

[4] GraphWalker, an open-source model-based testing tool , Online available: https://graphwalker.github.io/, Last Accessed 30.05.2025.

[5] UPPAAL, a model-based testing tool, Online available: https://uppaal.org/ , Last Accessed 30.05.2025.

[6] Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., & Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with Spec Explorer. In Formal Methods and Testing, 39-76.

[7] Prowell, S. J. (2005). JUMBL: A tool for model-based statistical testing. In Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 318b.

[8] Dulz, W., & Fenhua, Z. (2013). MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. In Proceedings of the Fourth International Conference on Quality Software.

[9] David, A., Larsen, K. G., Legay, A., Mikučionis, M., & Poulsen, D. B. (2015). Uppaal SMC tutorial. International Journal on Software Tools for Technology Transfer.

[10] Behrmann, G., David, A., & Larsen, K. G. (2004). A tutorial on Uppaal. In Formal Methods for the Design of Real-Time Systems.

[11] Yaghoubi, A., & Fink, G. (2019). A comprehensive survey of model-based testing tools. International Journal of Software Engineering & Applications, 10(2), 1-17.

[12] Shafique, M., & Labiche, Y. (2015). A systematic review of state-based test tools. International Journal on Software Tools for Technology Transfer, 17, 59-76.

[13] Y. Koroglu, M. Beyazıt, O. Kilincceker, S. Demeyer, and F. Wotawa, Towards Improving Automated Testing with GraphWalker," in 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2025, pp. 54-58.

[14] A. Silistre, O. Kilincceker, F. Belli, M. Challenger, and G. Kardas, Community detection in model-based testing to address scalability: Study design," in 2020 15th Conference on Computer Science and Information Systems (FedCSIS), 2020, pp. 657–660.

[15] F. Belli and N. Gökçe, Test prioritization at different modeling levels," in Advances in Software Engineering: International Conference, ASEA 2010, 2010, pp. 130–140.

[16] S. Tiwari, K. Iyer, and E. P. Enoiu, Combining Model-Based Testing and Automated Analysis of Behavioural Models using GraphWalker and UPPAAL," in 2022 29th Asia-Pacific Software Engineering Conference (APSEC), 2022, pp. 452-456.

[17] B. Nielsen, Towards a method for combined model-based testing and analysis," in Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014, pp. 609–618.

[18] R. Marinescu, C. Seceleanu, and P. Pettersson, An integrated framework for component-based analysis of architectural system models," in ICTSS 2012 Ph.D. Workshop, 2012, pp. 1–6.

[19] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui, "Model-Based Testing in Practice: An Industrial Case Study using GraphWalker," in Proceedings of the 14th Innovations in Software Engineering Conference (ISEC '21), 2021, pp. 1–11.

[20] Appium, an open-source framework, Online available: https://appium.io/docs/en/latest/, Last Accessed 30.05.2025.