FH Salzburg
MultiMediaTechnology

# *A Compiled Declarative Abstraction for Maestro-Based Android UI Testing: A Domain-Specific Language and Kotlin Compiler as a Test Abstraction Layer*

## Seminar Paper

Author: Denis Schüle
Advisor: FH-Ass.Prof. Andreas Bilke, MSc
Repository: https://gitlab.ct.fh-salzburg.ac.at/fhs48282/thesis

Salzburg, Austria, 30.01.2026

# A Compiled Declarative Abstraction for Maestro-Based Android UI Testing: A Domain-Specific Language and Kotlin Compiler as a Test Abstraction Layer

Denis Schüle
dschuele.mmtb-m2024@fh-salzburg.ac.at
Salzburg University of Applied Sciences

**ABSTRACT**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean venenatis nulla vestibulum dignissim molestie. Quisque tristique tortor vitae condimentum egestas. Donec vitae odio et quam porta iaculis ut non metus. Sed fermentum mauris non viverra pretium. Nullam id facilisis purus, et aliquet sapien. Pellentesque eros ex, faucibus non finibus a, pellentesque eu nibh. Aenean odio lacus, fermentum eu leo in, dapibus varius dolor. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin sit amet ornare velit. Donec sit amet odio eu leo viverra blandit. Ut feugiat justo eget sapien porttitor, sit amet venenatis lacus auctor. Curabitur interdum ligula nec metus sollicitudin vestibulum. Fusce placerat augue eu orci maximus, id interdum tortor efficitur.

## 1 INTRODUCTION

Mobile applications have become a core delivery channel for digital services, and their increasing feature richness and GUI complexity make it difficult to ensure correct functionality and behaviour under frequent updates and releases (Eke et al., 2025; Nie et al., 2023). In this context, GUI testing aims to verify that the application displays the correct information and that user interactions drive the system into the intended states. However, GUI testing is widely described as resource expensive and tedious, partly because GUI widgets are mutually dependent and interactions can trigger non-local effects across views and screens . This has fuelled continued research and practice interest in more effective and automatable GUI testing approaches for mobile apps (Nie et al., 2023).

In practical Android end-to-end testing, a recurring challenge is that executable UI tests are often written as step-by-step interaction scripts that encode detailed UI mechanics (e.g., widget-level actions and selectors). Such specifications are costly to keep consistent with evolving requirements and artefacts: manually ensuring consistency when requirements change is described as extremely time-consuming and error-prone (Silva & Fitzgerald, 2020). This problem becomes more pronounced for GUI-driven systems, where subtle UI or flow changes can invalidate many interaction steps even if the underlying behavioural requirement remains stable (Nie et al., 2023).

Tools that support higher-level specification formats can improve readability, but they do not automatically solve the core issue that many test suites lack an explicit, domain-level behavioural representation that remains stable under UI change. The resulting maintenance burden is not merely syntactic: inconsistencies can arise when requirements or stories lose synchronisation with other artefacts, producing mismatches between intended and checked behaviour (Silva & Fitzgerald, 2021). This thesis addresses this gap in the specific setting of Maestro-based Android UI testing by investigating whether a dedicated abstraction layer can separate domain intent from low-level execution steps.

Behaviour-Driven Development (BDD) structures user requirements as stories with acceptance criteria expressed as scenarios in a Given–When–Then format, where scenarios can be interpreted as state transitions (Silva & Fitzgerald, 2021). BDD is often positioned as enabling executable requirements and "living documentation" that communicates system status with respect to acceptance criteria (Silva & Fitzgerald, 2021). Importantly, BDD scenarios can be written at different abstraction levels: some steps use domain vocabulary (declarative) while others describe detailed user-system interaction (imperative), and this choice affects what can be reliably extracted or assessed from the text (Silva & Fitzgerald, 2021).

Prior work highlights that free-form story text makes automated interpretation and consistent downstream use difficult, motivating controlled natural language (CNL) and DSL-like constraints to reduce ambiguity and support reliable processing (Silva & Fitzgerald, 2021). While restricting vocabulary can reduce flexibility, it can also establish a shared terminology and mitigate miscommunication, ambiguity, and incompleteness in requirements and testing specifications (Silva & Fitzgerald, 2020, 2021). In parallel, mobile GUI testing research frequently uses model-based approaches; surveys report model-based methods as the most common family, but also note that model generation remains challenging and needs techniques that reduce modelling effort (Nie et al., 2023).

Taken together, these observations suggest a specific opportunity: combine (i) a constrained, domain-facing scenario language (to improve stability and processability) with (ii) a generator/translation pipeline (to keep executable tests in sync), rather than relying on runtime glue and ad-hoc manual updates.

This thesis proposes a *Compiled Declarative Abstraction* (CDA) for Maestro-based Android UI testing. The central idea is to specify interaction scenarios in a constrained, BDD-inspired external DSL—designed to encourage declarative, domain-level steps—then compile these specifications into

executable Maestro YAML flows. The approach is motivated by evidence that controlled vocabularies and structured story formats can support more consistent, comprehensive, and communicable specifications, reducing gaps between stakeholders and downstream artefacts (Silva & Fitzgerald, 2020, 2021).

With Maestro[1] , tests are expressed as UI interaction sequences, defined via YAML "flow" files that specify actions and assertions, and executed on a running mobile application. Because this style of testing exercises only externally observable behaviour, Maestro is best positioned as a primarily black-box approach for system-level validation (Eke et al., 2025).

Conceptually, the CDA treats the DSL specification (together with an explicit mapping from domain element names to GUI interaction elements) as a lightweight test model; compilation then becomes a systematic model-to-test derivation step. This aligns with the broader motivation to promote automated verification in an ever-changing environment, while keeping the specification readable and tightly connected to acceptance-criteria-style scenarios (Silva & Fitzgerald, 2020, 2021).

### 1.1 Research Questions

**RQ1** To what extent can a BDD-inspired DSL specify the interaction scenarios of a Jetpack Compose Android app in a way that is consistently translatable into executable Maestro tests?

**RQ2** How does the CDA approach (DSL + compiler) compare to manually written Maestro tests in maintenance effort and error-proneness when adapting to UI and behaviour changes?

### 1.2 Hypotheses

**H1** The DSL can specify all or almost all core scenarios without inline Maestro code, and compiled tests exhibit the intended behaviour.

**H2** Under realistic change scenarios, CDA requires less authored change (e.g., time and Lines of Test Specification) than the manual Maestro baseline.

**H3**

### 1.3 Contributions

This thesis contributes a *Compiled Declarative Abstraction* (CDA) for Maestro-based Android UI testing. Concretely, it provides: (i) a BDD-inspired external DSL for specifying mobile interaction scenarios at a domain level, (ii) a locator configuration scheme and mapping/ambiguity policy that separates semantic element names from concrete selectors, (iii) a Kotlin-based compilation pipeline (ANTLR parsing, AST, code generation) that translates DSL scenarios into executable Maestro YAML flows with early error reporting for unsupported or ambiguous specifications, and (iv) an empirical evaluation comparing the CDA approach against a best-practice manual Maestro baseline with respect to expressiveness/translation reliability and maintenance effort and error-proneness under change.

### 1.4 Thesis Outline

Section 2 reviews related work and positions the thesis in mobile GUI testing, BDD-style specifications, model-based testing ideas, and test abstraction patterns. Section 3 introduces the CDA concept and the underlying assumptions, including how scenarios, screens, and locators are represented and how correctness is operationalised. Section 4 details the design and implementation of the DSL, locator mapping, and the Kotlin compiler that generates Maestro YAML. Section 5 describes the case application and the two test suites (CDA-generated and manual Maestro) used in the study. Section 6 presents the evaluation methodology, metrics, and results for the research questions. Section 7 discusses the findings, limitations, threats to validity, and implications for test abstraction design. Section 8 concludes the thesis and outlines directions for future work.

## 2 Related Work

Introduce why this specific related work is important for your own work. Which areas do you cover and why? What do you take as inspiration and what do you do differently/improve upon?

## 3 Methods

This chapter explains and justifies the data and methods used in the thesis, detailing the chosen methodology's suitability for the research question and situating it in relation to existing approaches discussed in the Related Work section.

## 4 (Optional) Implementation

Provide implementation details such as the used software and our software architecture, highlight your own solutions to encountered difficulties. Describe relevant iterations of your implementation.

See code **??** for high efficiency code.

## 5 Results

This section details the technical and methodological implementation of the methods, including their adaptation to the underlying database and the developed solution to the problem.

## 6 Discussion

Discuss your results to answer your research question. Does your data support you hypotheses? Put your results into perspective by situating it in the research field/related work. Outline limitations.

## 7 Conclusion and Outlook

Summarize your work and future work.

[1] https://maestro.dev/

## REFERENCES


Eke, N., Salihu, I., Usman, A., Ibrahim, R., & Mshelia, Y. (2025). A survey of automated testing techniques for android-based mobile applications. *Nig. J. Tech.*, *44*(2), 311–337. https://doi.org/10.4314/njt.v44i2.15

Nie, L., Said, K. S., Ma, L., Zheng, Y., & Zhao, Y. (2023). A systematic mapping study for graphical user interface testing on mobile apps. *IET Software*, *17*(3), 249–267. https://doi.org/10.1049/sfw2.12123

Silva, T. R., & Fitzgerald, B. (2020). Interactive Workshop on the Industrial Application of Verification and Testing ETAPS 2020 Workshop (InterAVT 2020).

Silva, T. R., & Fitzgerald, B. (2021). Empirical Findings on BDD Story Parsing to Support Consistency Assurance between Requirements and Artifacts. *Evaluation and Assessment in Software Engineering*, 266–271. https://doi.org/10.1145/3463274.3463807


This work has the following word count (counted by texcount):