Personas for Model-based Testing

Bogdan **Budihala**[1], Judith **Michael**[1,2,*] and Bernhard **Rumpe**[1]

[1]*Software Engineering, RWTH Aachen University, Aachen, Germany*

[2]*Programming and Software Engineering, University of Regensburg, Regensburg, Germany*

Abstract

The usage of personas, fictional representations of user archetypes, during the initial stages of requirements elicitation has yielded significant improvements in mitigating assumption bias, stemming from the development engineers' preconceptions about end users that frequently do not accurately reflect the actual needs of those users. This paper explores how personas can be used to support automated testing workflows, bridging the gap between higher-level conceptualizations and code. Although personas are commonly used in requirements engineering and during software development processes, they are rarely used for model-based testing. Our approach uses personas in automated testing to construct a software solution that is capable of validating system requirements captured in personas. We propose a domain-specific language capable of encompassing the high-level requirements of a target system through scenarios, flows, and execution plans, and we generate the test suite infrastructure, which features test scenario skeletons. Although these skeletons require handwritten code to perform the execution steps, this process may be further automated to produce executable source code with minimal human intervention.

Keywords

Automated Testing, Personas, Domain-Specific Languages, Model-Based Software Engineering, Test Models

1. Introduction

Personas are hypothetical archetypes and user-centered conceptual abstractions of real users to be used in software engineering processes [1]. They serve as a communication base between developers and stakeholders to achieve a common understanding of a product. In addition, personas help mitigate the risk that designers are unconsciously creating solutions tailored to their own experiences rather than those of the intended user base [2, 3]. Furthermore, relying solely on feedback from actual users has often proven to skew the design process, as actual users display unique behaviors that may not be ideal for a universal design strategy. Personas have seen successful wide-audience adoption and are a well-established practice [4, 5] in software development [6, 7], healthcare [8, 9], technology for impaired or elderly [10, 11], and education [12, 5, 13]. Historically, personas were constructed through interviews, surveys, and questionnaires; thus, research was initially concerned with proposing frameworks that facilitated data collection and processing [14], optimizing interview questions to maximize relevant information extraction for personas [15], or natural language processing for creating personas [16, 17].

Although research on personas has seen substantial advancements [18, 19, 20], the usage of *personas* for improving *automated testing* remains relatively unexplored. Such works can be found for game testing (also known as playtesting) [21, 22, 23] to construct personas for enacting different variations of user behavior when interacting with game content. Other approaches use personas to support user testing, e.g., for accessibility needs [24], explore using personas and Large Language Models (LLMs) for exploratory testing [25] or generating test cases with LLMs based on learning user behavior patterns [26]. These approaches have limited benefits for us because they primarily focus on game, user or exploratory testing, and lack support for structured integration testing of applications.

To improve the testing process based on information from the requirements elicitation phase, we are exploring *how personas can be efficiently translated into automated test cases to validate system requirements.* This enables us to bridge the gap between higher-level conceptualizations in the requirement

elicitation phase and the generated test code [27]. Our *main contribution* is the conceptualization of personas in models as containers for execution plans, with each plan corresponding to a test case in a suite. Personas allow for flexible mixing of execution plans to match specific persona characteristics and attributes. Technically, personas, execution plans, and scenarios are represented using a custom-built Domain-Specific Language (DSL), which allows for structured specification and data exchange. In addition, we show how execution plans are transformed into test case source code, producing executable artifacts for constantly evolving systems.

Structure: Sec. 2 provides background context, while Sec. 3 outlines key design considerations and the system architecture. Sec. 4 describes the system implementation and automated testing. We present a practice use case and analyze the results in Sec. 5, and compare our approach to related work in Sec. 6. The last section summarizes key findings, limitations, and future directions.

2. Preliminaries

Software testing is an umbrella term referring to a vast pool of approaches to assess software quality and reliability. Here, we set the context for automated testing methodologies, with emphasis on model-based testing, as well as the concept of personas and the engineering of DSLs.

I. Personas. Personas [1] are fictional characters created with the intent to represent various groups of software users and their behavior [28]. They are used with the forethought of avoiding assumption bias by outlining the needs of the end users, rather than the preconceptions held by system developers about those end users [18, 19, 20]. Karolita et al. [4] analyzed over 40 research papers summing up 100 personas. The authors uncovered several domains of application, among which software development, healthcare, and technology for the impaired or elderly have the biggest audience. Key dimensions for text-based personas are *narration* (narrative or summarized), *format* (unstructured, semi-structured, or structured), and *length* (briefly summarized to lengthy). We use *semi-structured personas* with narration and structured steps. The narration encapsulates the persona's goals and behavior during system testing, while the structured part details their concrete steps, organizing their actions systematically.

II. Model-based testing. Model-based testing leverages models to automate the test life-cycle (generation, execution, and evaluation), grounded in the principle that the behavior of a system can be accurately represented through a model from which test cases can be automatically derived [29, 30, 31]. These models can represent various aspects of system behavior, such as state transitions, control flow, or data interactions. They may be expressed using formal and semi-formal methods, like finite state machines [32], Petri nets [33, 34], or UML diagrams [35, 36]. By representing the system's behavior abstractly, model-based testing provides a more comprehensive coverage strategy for edge cases and other hard-to-reach scenarios. Once the model is created, model checking [37] explores different paths and conditions, and we can generate test cases executed on the target system across various levels of testing, ranging from unit and integration tests to system and acceptance testing. *We focus on integration testing*, with the primary goal to validate the interactions between different system components. Our approach is based on state transition and control flow models, allowing us to represent and test how system components interact and ensuring they function correctly when integrated.

III. Domain-specific languages. DSLs are specialized languages designed to aid development within specific domains using domain concepts [38, 39, 40, 41]. They can be categorized into internal DSLs embedded within General-Purpose Languages (GPLs) benefiting from the host language's infrastructure, and external DSLs with standalone syntax requiring custom parsers and compilers. We create DSLs with MontiCore [42], a language workbench supporting the compositional definition of DSLs. Language engineers define the abstract and concrete syntax of a DSL [43] as context-free grammars (CFGs). MontiCore generates infrastructure to parse textual models [44] into abstract syntax trees (ASTs), check their well-formedness, and process them via model transformation or template-based code generation. To create languages, one can reuse existing language components such as literals, types, or expressions [45], to create more complex languages, such as UML/P [46], a UML variant for programming, or BPMN [47] up to larger language families [48], such as for IoT systems [49] or assistive systems [50].

3. Main considerations

We examined the following structural and technical considerations for our approach.

I. Structural considerations outline the architecture of automated testing models, as well as the hierarchical structure they should follow to simulate more genuine user interactions. They focus on the importance of structuring the testing framework by granularity, together with the logical relationships that would ensure there is broader coverage of testing scenarios.

SC1 *Representing system requirements* - The solution must incorporate dynamic representations, ensuring the test framework's design can be adjusted based on new information or changing requirements, providing a more comprehensive and better-performing test suite.

SC2 *Optimal level of granularity* - The solution must permit hierarchical structuring of requirements into different categories (encapsulation). This layering helps manage dependencies and ensures that changes at one level can be appropriately propagated throughout the model.

SC3 *Personas as part of a customer journey* - The solution must evaluate the interaction a customer has with the system over time. This allows testers to identify the most common paths customers take and prioritize testing those flows to ensure they are as smooth and error-free as possible.

II. Technical considerations cover some of the techniques and tools required to realize the structural blueprint, including domain-specific languages and automatic code generation. This draws attention to the technical underpinnings which make the proposed testing methodology technically practical and flexible enough to accommodate personas in automated testing workflows.

TC1 *Flexible representation* - The proposal must support the necessary specificity needed for detailed test scenarios, with the aim of reducing boilerplate code or additional layers of abstraction to simulate specific testing conditions.

TC2 *Automated code generation* - Translating test cases into code is a complex task and requires effort to adapt to changing requirements. Automating the translation of test scenarios ensures that the system can iteratively regenerate test code without manual intervention.

TC3 *Standardized structure of the generated code* - The generated code files must follow standard testing paradigms by incorporating a behavior test specification format that ensures the generated code files conform to a specific structure, namely *given - when - then*.

4. Approach

Our solution includes four main processes: collect system requirements, translate them into models, generate and populate the test suite, and run the suite while evaluating the system under test (see Fig. 1a). The System Requirements Specification (SRS) defines system expectations and is the basis of personas and feature graphs, which guide the test suite. The test suite helps generate the test plan, ensuring requirement coverage via a traceability report, which informs refinements to the SRS. To realize this big picture, we have created additional artefacts and tooling.
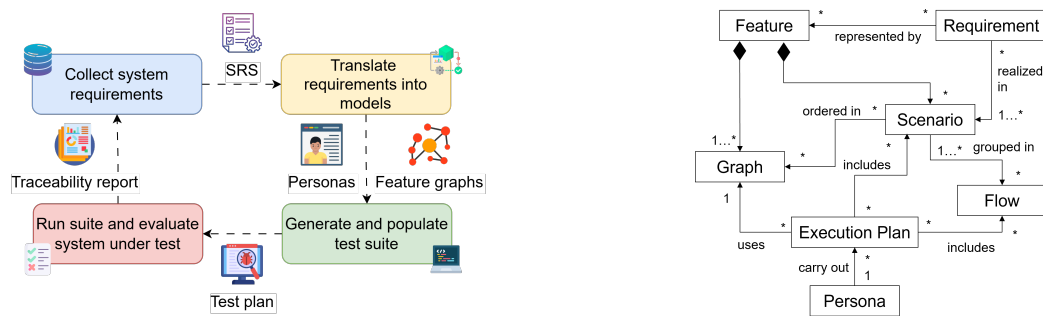
(a) Big picture of the developed solution

(b) Main concepts for describing system behaviour

Figure 1: Overview of our approach: (a) the overall pipeline; (b) the domain-concept metamodel.

I. Overview of the solution artefacts. `Features` (see Fig. 1b) represent high-level system capabilities, each composed of interconnected `scenarios` that work toward a shared objective realizing `requirements`. The scenarios' lifespan is linked to the features they have been defined in. The requirement's complexity determines whether it is best represented by a single scenario or multiple ones, and if it requires an entire feature or multiple features to be fully covered. After organizing the requirements in a structured way, the connections and logical order between different scenarios are represented in a `graph`. This graph shows the paths that can be taken from one scenario to another to achieve a particular goal. Similar scenarios that contribute to the same goal can be grouped into a `flow`, e.g., registering, confirming an email, and logging in can be grouped into an authentication flow. Scenarios can be reused across features in flow definitions to connect otherwise disconnected graphs. The graph provides an overview of all possible interactions that can occur within the system. The testing is based on `execution plans`, which define a subset of these interactions and concretize them into test cases. For an execution plan to be valid, each plan entry (scenario or flow) must be reachable from the previous entry in the graph. For a flow, the last flow node must be connected to the next entry within the plan to ensure continuity. `Personas` play an essential role in carrying out the execution plans. They represent an archetype of a user with specific goals and characteristics, underlying factors upon which a persona can initiate one or multiple execution plans to test the system's robustness.

Technical specification of the developed solution. We have developed our tooling using Gradle for build automation and Monticore's command-line tools and runtime environment for analyzing, processing, and generating AST representations from grammar files [51]. The project's backend has been developed using Java and integrates Monticore for language processing. The system's user interface was built using React for visually modeling the various aspects of the system under test.

II. DSL structure definition. To represent the concepts in Fig. 1b, we have created three domain-specific languages as CFGs: The Feature grammar, the Graph grammar, and the Persona grammar.

The `Feature` grammar[1] aids in constructing a feature representation in a format that is both human- and machine-readable. It defines the actions within the feature, a sequence of events following the `Given`, `When`, and `Then` behavior-driven development paradigm, where features are described in terms of preconditions (given), events that change these conditions (when) and the expected outcome (then).

The `Graph` grammar[2] enables the system user to provide structural graph information in an annotation-based format, similar to the annotation mechanism supported in JVM-based programming languages. As CFGs have limited expressivity, certain integrity constraints remain to be enforced as context conditions (CoCos). For instance, ensuring that the scenario identified in the file exists beyond its declaration (i.e., in a `Feature`) or ensuring that a scenario context file is present and readable.

```
1   grammar Persona extends de.monticore.MCBasics, de.monticore.literals.MCCommonLiterals, FeatureGraph {
2       start Persona;
3
4       symbol scope Persona = "persona" name:Name ":" PersonaAttribute*;
5       interface PersonaAttribute;
6
7       Description implements PersonaAttribute = "-" "description" ":" description:String;
8       ExecutionPlans implements PersonaAttribute = "-" "plans" ":" ExecutionPlan+;
9       ExecutionPlan = "-" name:Name ":" "[" ExecutionPlanEntry ("," ExecutionPlanEntry)* "]";
10
11      enum ExecutionPlanEntryType = FLOW: "flow" | SCENARIO: "scenario";
12
13      ExecutionPlanEntry = ExecutionPlanPrefix? Identifier;
14      ExecutionPlanPrefix = type:ExecutionPlanEntryType ":";
15  }
```

Listing 1: Persona Grammar

The `Persona` grammar (see Listing 1), allows the construction of user archetypes in a semi-structured manner, using syntactical components that are human-readable, yet sufficiently expressive to help create execution plans by assembling scenarios and flows. Apart from the entrypoint `Persona`, users can

---

[1]https://github.com/bbudihala/Personas-for-Model-based-Testing/blob/main/grammars/Feature.mc4
[2]https://github.com/bbudihala/Personas-for-Model-based-Testing/blob/main/grammars/Graph.mc4

define various attributes that extend the `PersonaAttribute` interface. The validity of the persona's execution plan chain is assessed as an application-level context condition.

III. Application-level assembly. The processing pipeline is centered around a `PersonaTool` component using Monticore's code generation and injecting hand-written code into the framework. The pipeline applies a path scan to identify all relevant models of the three DSLs, parses these models, and creates their ASTs. In the next step, we create a symbol table utilizing a scope genitor to form associations between each of the individual AST components. The root symbol is added to the global scope, making it accessible throughout the processing flow. To ensure the integrity and add constraints, we apply several CoCos, e.g., single graph annotation per type, single graph node reference, and linkable execution plan components. We use the visitor mechanism to traverse the AST for generating code tailored to each specific type of component within the AST.

IV. Test components continuity. The workflow requires chaining multiple components. To ensure continuity between generated components, we used a key-value string store (i.e., `HashMap<String, String>` in Listing 2, l.4) passed between all components involved in the execution chain. We introduced various run$1 methods (e.g., `runScenario`, `runFlow`, `runPlan` in Listing 2, l.3-7) responsible for state control and propagation between components. Both scenarios and flows were designed with the intent to be idempotent, so that they may yield the same results regardless of the execution plan configuration they are currently being run from.

```
1   public class $5 {
2       @Test
3       public void run$6Plan() {
4           var state = new AtomicReference<Map<String, String>>(new HashMap<>());
5           new $7<String, String>().runScenario(state::get, state::set);
6           new $2<String, String>().runFlow(state::get, state::set);
7           new $8<String, String>().runScenario(state::get, state::set);
8       }
9   }
```

Listing 2: Persona State Management Skeleton

State management code blocks are constructed for a particular `ASTScenario` (see generated scenario template in Listing 3) and a `ASTWorkflow` respectively, using a custom-written wrapper for `JCodeModel`. When visiting an `ASTScenario`, a series of operations are conducted, e.g., the construction of an `init()` method (Listing 3, l.4) capable of receiving the state from the previous entry in the chain. The scenario is also expected to produce a state, using a `store` method (Listing 3, l.7) that loads the output of the scenario into an `outputConsumer` functional interface.

```
1   public class $1<KT,VT> {
2       private Map<KT, VT> input;
3       private final Map<KT, VT> output = new HashMap<>();
4       public void init(Supplier<Map<KT, VT>> ins) {
5           input = Objects.requireNonNullElse(ins.get(), Map.of());
6       }
7       public void store(Consumer<Map<KT, VT>> outc) {
8           outc.accept(output);
9       }
10      public void runScenario(Supplier<Map<KT, VT>> ins, Consumer<Map<KT, VT>> outc) {
11          init(ins);
12          store(outc);
13      }
14  }
```

Listing 3: Scenario State Management Skeleton

With the introduction of flows (linear groups of scenarios) as part of the graphs, a hybrid visitor approach was necessary. Each flow with its associated linear scenario subgraph leads to the generation of a `runFlow` method (Listing 4, l.2), capable of both receiving and passing state. Furthermore, all linear subgraph scenarios append a scenario invocation line in the newly constructed `runFlow` method's body for each of the subgraph vertices (see Listing 4 for a flow spanning across three scenarios).

The generated code forwards data between entries of the execution chain that allows for idempotent and thread-safe composition, with the use of an `AtomicReference` to the key-value store (see Listing 2

for an execution plan with two scenarios and a flow). However, the input data ingested into a scenario is not persisted as an output by default. To achieve this, the scenario skeleton must be adjusted to copy the data of the `input` attribute into the `output`. This may be relevant when the input data of a scenario needs to be used in a subsequent scenario within the execution chain. Since input data is not persisted as output by default, failing to copy it explicitly could potentially disrupt the flow.

```
1  public class $2<KT,VT> {
2      public void runFlow(Supplier<Map<KT, VT>> ins, Consumer<Map<KT, VT>> outc) {
3          new $1().runScenario(ins, outc);
4          new $3().runScenario(ins, outc);
5          new $4().runScenario(ins, outc);
6      }
7  }
```

Listing 4: Flow State Management Skeleton

V. Tool support for visual modeling. We have developed a graphical tool using an intuitive visual representation of the test suite generation pipeline components, capable of transforming the graphical model representation into textual models in the three developed DSLs. Depending on the application needs and use cases, a test suite developer may choose to use this tool or directly feed grammar-specific input into the pipeline.

(b) Flow allocation inside a scenario

(a) Scenario-creation form
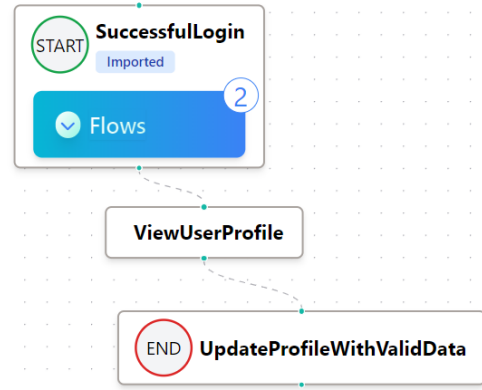
(c) Chaining with imported scenarios

Figure 2: Tooling examples: (a) creating a scenario; (b) allocating flows; (c) scenario importing and chaining.

We use an *authentication feature* and a *profile management feature* as running examples to describe our tool workflow from a user perspective. Fig. 2a and Fig. 3a show the creation of scenarios for successful user registration, including a description and the necessary steps for executing the scenarios.

Scenarios can be enriched interactively to meet specifications via static attributes that are injected via `.properties` file as key-value pairs.

Our tool provides functionality to link scenarios according to their possible execution order. E.g., the scenarios *successful* and *failed registration*, *successful* and *failed password resetting*, and *successful authentication.* have a logical order. Thus, developers should be able to define the scenario order in a test suite. In our running example, successful user registration is the graph's start node (see Fig. 3a). Once the system acknowledges this operation and produces an output in an actual test suite, three further actions may occur: registering with an existing email, resetting the password successfully, and

logging into the system successfully. The successful login scenario may originate from the two others or directly from successful registration. However, under no circumstances can an attempt to change the password, successful or not, occur before a successful login.

Taking the assumption into account that test suites are initiated from a clean system state, user registration and login scenarios may appear frequently; we group them into a single linear flow - `AccessSetup` to perform both scenarios in a transactional manner, allowing the test developer to assess the system's functionality from a guaranteed authenticated context. Similarly, the `AccessRecovery` flow performs registration, password reset, and login transactionally. Once associated with a flow, a node's representation has an additional component representing all flows this node is contained in (see Fig. 2b). If the last node of a flow is deallocated from this flow, the whole flow is discarded.
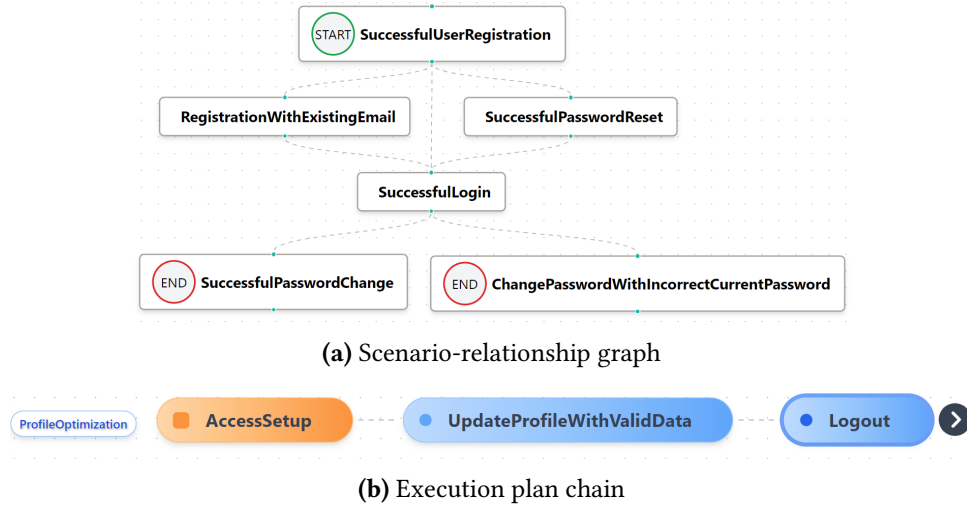

(a) Scenario-relationship graph

(b) Execution plan chain

Figure 3: Modeling overview: (a) scenario-relationship graph; (b) single execution plan chain.

Reusing scenarios in different features requires importing the scenarios with a few integrity constraints. Notably, any changes made to the imported node are reflected in the original scenario. Moreover, when translating the information into input files for test case generation, importing a scenario links the two graphs into a larger graph. This is illustrated in Fig. 2c. The imported scenario can be associated with a subsequent flow in the importing feature rather than being restricted only to the original flows.

Test case construction. After the graphs have been established, the next step is to proceed with the actual test case construction process. We define an execution plan that tests the behavior of the persona "Nina the networker" representing users that are primarily interested in keeping their profile up to date. In Fig. 3b, the `ProfileOptimization` execution plan illustrates the sequence of steps taken by the test suite to fulfill the goals of the persona. These steps include setting up access, updating the user profile, and logging out. The execution plan consists of three components, which are marked with distinct symbols: an orange node with a square icon representing a flow with multiple scenarios, while the two blue nodes with a circular icon represent two self-contained scenarios.

VI. Test skeleton population. To fully test the functionality, we have integrated LLMs to aid in populating the generated test cases within a predefined skeleton. We require a technical manifest when starting the process, an OpenAPI specification that outlines the request and response schemas of the system under test. This file guides the model in generating code that conforms to the actual specifications of the system. As we proceed with test case generation, the LLM receives prompts based on the manifest and the scenario skeleton, with the generated content being continuously refined through our feedback. The LLM's role in this process involves interpreting the OpenAPI specification and other relevant inputs, filling in placeholders for the test case steps, and ensuring that the generated test code is both syntactically and semantically accurate. We use few-shot learning and chain-of-thought to guide the model, helping it understand the context and structure of the test cases. The model is also given an illustrative scenario to follow, to steer it towards generating the correct Java test code for API interactions using the `java.net.http.HttpClient` class, while preserving the required method

signatures and incorporating authentication headers.

The following key segments are filled in using the `gpt-4-turbo` model (see prompt in appendix[3]). We first prompt the LLM to generate necessary *import statements* based on the specific scenario, adding additional imports if needed to support the scenario steps. *Context variables* represent immutable scenario-specific attributes injected from properties files. It can add new variables for use within the scenario, but we forbid it from modifying existing ones. For the *given step*, it generates code for initializing inputs (e.g., request bodies or parameters) while preserving the method signature, maintaining the test structure. In the *when step*, the LLM materializes the inputs into outputs by executing actions, e.g., API calls. The method signature must remain unchanged for consistency. In the *then step*, it asserts the output, verifying that the expected results were achieved while preserving the method structure, concluding by filling in the control flow, ensuring proper management of input and output states, with a standardized structure for consistent execution. We run a two-pass check (compile + simple assertions) before accepting a completion. In our case study, completions typically stabilized within 1 to 3 iterations per step, with manual edits focused on authentication headers and payload shape alignment.

5. Validation of the approach

We validate our approach using an industrial application from the financial domain, namely a sandbox environment of the Square Customers API[4]. It is a financial services platform that provides a RESTful interface supporting CRUD customer operations and integrates with Square's broader suite of services (payment processing, digital receipts, loyalty programs, etc.). It acts as a centralized source of truth and uses webhooks for real-time event notifications to enable the synchronization of customer data across multiple software components. We demonstrate how the API can be employed in a controlled testing environment to evaluate its integration within a larger software ecosystem.

We are describing system requirements of the Square Customers API through SRS tables (see the appendix[5]) used to test our approach. We explain the validation using the first SRS table, namely, customer profile management. Customer profile management is a mid-sized requirement covering basic operations that act upon a customer's profile: *creating a customer with valid and invalid data; retrieving a created customer; updating an existing or deleted customer; deleting a customer; listing all customers.*

Modeling. From persona traits to plans, we define attributes via three heuristics: (i) Goal intent: flows covering the happy path of that goal; (ii) Risk tolerance: negative/edge scenarios (e.g., invalid data, recovery) for QA-oriented personas; (iii) System touchpoints: features matching the persona's role (e.g., backend focuses on CRUD flows). Translating the SRS requirements into scenarios is a one-to-one mapping: one step of the SRS is one scenario. Modeling permits multiple representations of the same concept; thus, one of the modeling decisions applied consisted of splitting the SRS requirement into two features, yielding a more granular view of the scenarios: (1) All procedures that mutate the customer's profile in any shape or form are grouped into the Customer Profile Altering Feature, and (2) read-only procedures related to the customer's profile are grouped into the Customer Profile Querying Feature.

Fig. 4a shows the customer profile altering feature with six scenarios, five of which originate from this feature, and one being imported from the customer profile querying feature, identifiable by the `Imported` badge. The scenario `CreateCustomer` is marked as a starting node, as it is a prerequisite for all but one of the scenarios involved in this feature. The `CreateCustomerWithInvalidData` is represented as a floating node, signifying that no other node within this feature may be chained to lead to that scenario, though its connections are defined in another feature that imports it. This example showcases the expressive power of our DSL: All concepts are joined into a single reachability graph during processing. However, the tool currently lacks a preview of the joined reachability graph.

To cover the first SRS's range of operations, the customer profile querying feature (see Fig. 4b) depicts

---

(a) Customer profile altering feature   **(b)** Customer profile querying feature
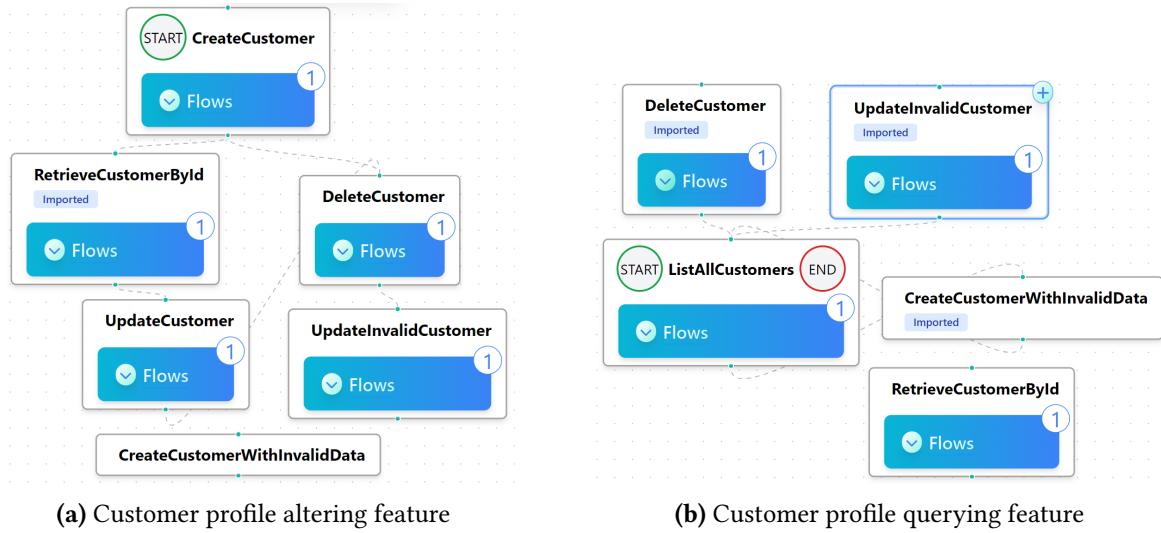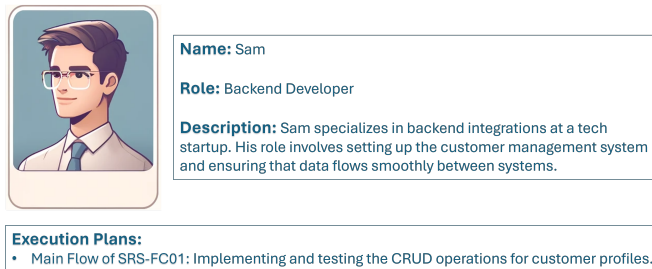
Figure 4: Customer profile features: (a) altering; (b) querying.

the connections between two scenarios originating from this feature and three imported scenarios from the customer profile altering feature. Here, the node `CreateCustomerWithInvalidData`, previously floating, is modeled as a two-way connection to `ListAllCustomers`, consequently allowing finite chain loops in a potential execution plan (e.g., list, create, and redo listing). Listing the customers is both a start and end node; thus, any execution chain involving the creation of customers with invalid data may only end with the retrieval of all customers.

Accounting for the technical nature of the system under test, the users utilizing the Square Customers API have a high software development literacy. As a result, we have constructed *five personas*, ranging from software development and quality engineering representatives to product managers (see all semi-structured personas in the appendix[6]).

Name: Sam

Role: Backend Developer

Description: Sam specializes in backend integrations at a tech startup. His role involves setting up the customer management system and ensuring that data flows smoothly between systems.

Execution Plans:
• Main Flow of SRS-FC01: Implementing and testing the CRUD operations for customer profiles.

(a) Sam, Backend Developer

```
1   persona Sam:
2   - description: "Sam specializes in
3   backend integrations at a tech
4   startup. His role involves setting
5   up the  customer management system
6   and ensuring that data flows
7   smoothly between systems"
8   - plans:
9       - SRSFC01MainFlow: [
10          flow:CustomerCRUD,
11          scenario:ListAllCustomers
12      ]
```

(b) Input model for Sam's persona

Figure 5: Sam's persona (a) and its corresponding input model (b).

For the Customer Profile Management SRS, we focus on Sam, the Backend Developer persona (Fig. 5a). The personas yield valuable insight into the flow allocation, with the core development team, depicted by Sam, more inclined towards happy paths and main flows, in contrast to the test and quality assurance engineers, who are more drawn towards the alternative, exception, and edge case flows, as was the case for the other personas. The follow-up step in the modeling phase involves constructing execution plans for individual personas (see appendix[7] for each persona's individual execution plan). The modeled execution plans offer full coverage of all of the steps within both SRSs, by utilizing a mix of scenarios and flows. Moreover, any additional requirements introduced during the requirements validation process may be represented with reduced effort, depending on the overlap factor with the current model.

After exporting the model within the UI, several artifacts are produced as input for the code generator

---

[6]https://github.com/bbudihala/Personas-for-Model-based-Testing/blob/main/pictures/Persona_Sam.png
[7]https://github.com/bbudihala/Personas-for-Model-based-Testing/blob/main/pictures/SRS_Execution_Plan.png

pipeline. The artifact presented in Fig. 5b is a sample pretty printed `.persona` model, whose contents are parsed, assembled, and processed according to the application-level assembly workflow. Though not depicted, other generated artifacts are a model fitting to the feature DSL containing feature and scenario definitions, and a model fitting to the graph DSL outlining the connections between the scenarios.

```java
public class Sam {
    @Test
    public void runSRSFC01MainFlowPlan() {
        var state = new AtomicReference<Map<String, String>>(new HashMap<>());
        new com.generated.ftg.CustomerCRUD<String, String>().runFlow(state::get, state::set);
        new com.generated.feature.customer.profile.querying.ListAllCustomers<String, String>()
            .runScenario(state::get, state::set);
    }
}
```

Listing 5: Generated Code for Sam's Persona

Listing 5 illustrates the generated source code corresponding to Sam's modeled persona. To successfully deploy the generated test suite, the concrete "given", "when", and "then" steps of the generated scenario skeletons needed to be implemented. An example of this population process is detailed in the appendix[8], which features the scenario `CreateCustomer`, subjected to a semi-automated filling process with the help of an LLM, queried multiple times until convergence was achieved.

Summary. We showed the application of our approach using an SRS table linked to two distinct features: customer profile querying and modification. With Sam's persona, we were able to fully capture the main flow of managing a customer profile corresponding to a single execution plan.

For a more comprehensive validation of our approach, we mapped two SRS tables to four distinct features, addressed by five different personas representing multiple archetypes, including frontend and backend developers, QA and automation engineers, and product managers. The latter spanned across 18 scenarios and five unique flows, evaluated in seven execution plans. Concluding the end-to-end demonstration, a Gradle test report summarizing the test performance is showcased in the appendix[9], displaying 100% success rate for the system requirements, including Sam's main flow.

6. Discussion & related work

To address both structural and technical aspects of our automated testing approach, we have implemented a solution that focuses on adaptability and seamless integration with system changes. We outline how our approach meets the structural and technical considerations defined in Sec. 3. *SC1–SC3* are satisfied through adaptable scenario/flow models, granular encapsulation, and persona based execution plans mirroring user journeys. *TC1–TC3* are met by a flexible DSL minimizing boilerplate, automated visitor code generation, and standardized "given-when-then" skeletons with explicit state hooks.

Assumptions. Our core assumption is that personas can accurately represent real-world user archetypes in the context of automated testing. Although our proposal integrates personas into the testing workflow, it presumes that these personas sufficiently capture diverse user behaviors.

Strengths. The strengths of our paper lie primarily in its integration of personas with automated testing, an area that has seen limited exploration in existing literature. Unlike existing tools such as GraphWalker[10], which often struggle with requirement traceability, our approach embeds requirements directly into test models, enhancing clarity and validation. In terms of generalizability, our approach has the potential to be applied to APIs of systems in which model-based testing is feasible. It is well-suited for different domains, particularly those relying on automated validation and continuous integration pipelines. We zoomed in on one such domain, customer management via the Square Customers API, to demonstrate the framework's ability to handle real business processes effectively, while also incorporating best practices from related research, such as reusing scenarios across multiple models and executing repeated test runs to analyze variations in results.

---

[8]https://github.com/bbudihala/Personas-for-Model-based-Testing/blob/main/snippets/CreateCustomer.java
[9]https://github.com/bbudihala/Personas-for-Model-based-Testing/blob/main/pictures/Gradle_Test_Report.png
[10]https://graphwalker.github.io/

Limitations. Currently, the system only supports linear execution plans, which may be insufficient for applications requiring dynamic branching logic. Using more complex workflow languages, e.g., BPMN [52], could help to overcome this shortcoming. Additionally, our approach does not incorporate built-in path coverage analysis or generation of random execution paths, a feature commonly found in other model-based testing tools. To overcome this, we may consider integrating a symbolic execution engine into our system, which allows us to track the different paths during execution and identify uncovered paths, as well as using fuzz testing techniques to produce random execution paths.

Related approaches. Despite extensive research on personas and automated testing separately, little work has explored their combined application. Most studies on personas in testing focus on procedural personas in playtesting [21, 22], where AI-driven personas evolve iteratively to assess gameplayability. While their focus is on game testing, their concept of evolving personas is relevant to software testing, suggesting a potential integration of adaptive personas into automated testing workflows. However, how well AI-generated personas can fully capture user archetypes remains an open question.

In model-based testing, Garousi et al. [53] evaluate model-based testing in an industrial context, noting that as test artifacts grow, they form a separate ecosystem requiring maintenance. They demonstrate test modeling using GraphWalker, where states represent GUI elements and edges denote user actions. Automated model generation using "GUI ripping" can result in overly detailed models, requiring substantial refinement. While this method is not always feasible for modern JavaScript-heavy applications, applying similar techniques to API-based systems could be promising. Additionally, the study highlights GraphWalker's lack of built-in requirement traceability, in contrast to our approach, where requirements are embedded within scenario descriptions.

Tiwari, Iyer and Enoiu [54] introduce model checking earlier in the model-based testing process by transforming GraphWalker models into timed automata. This enables verification of reachability and deadlocks before test execution. As seen across many GraphWalker-based studies, the level of state detail is often excessive, making requirement tracking difficult. Our approach instead employs broader scenarios with embedded natural language descriptions to maintain traceability.

Only a few approaches are using LLMs and model-based testing: Most studies focus on LLM-driven test case generation for GUI applications [55, 56]. Yu et al. [55] introduce an AI assistant that suggests test scenarios to manual testers, an approach that could enhance our approach by guiding test case creation. Liu et al. [56] explore converting natural language descriptions into executable test scripts—closely aligning with our methodology, where Javadoc descriptions are transformed into test steps via LLMs. Both studies highlight the limitation that LLMs struggle with long input prompts, sometimes causing memory overload and loss of contextual continuity. Our approach addresses this by considering truncation and segmented prompting strategies to optimize input processing.

7. Conclusion

This paper explored the use of personas for automated testing to validate system requirements through modeling and code generation. It proposes a framework using scenarios, flows, and execution plans to translate persona attributes into test cases. Even though the current implementation lacks the branching of execution plans and random path generation, our approach effectively validates requirements using model-based techniques. Future work may include evolving persona attributes over execution flows, generating models from API specifications, introducing random execution paths, and applying model checking for early validation. In addition, it would be interesting to test the approach for different kinds of systems, e.g., assistive system components [57], or accessible software [58].

Declaration on Generative AI

The author(s) have not employed any Generative AI tools for preparing this publication.

References

[1] A. Cooper, The Inmates are Running the Asylum, Vieweg+Teubner Verlag, Wiesbaden, 1999.

[2] A. Trujillo, H. Martínez, J. Flores, F. Sabogal, F. Gonzales, F. Paz, User centered design methods in software development: A case study in a peruvian office of financial aid and scholarships, in: 2024 IEEE ANDESCON, 2024. doi:10.1109/ANDESCON61840.2024.10755895.

[3] N. Bartels, S. André Scherr, B. Gültekin, S. Ludborzs, S. Storck, A. Zepp, Comic-based morphological box: Enhancing vision design - a research preview, in: IEEE 32nd Int. Requirements Engineering Conf. (RE), 2024. doi:10.1109/RE59067.2024.00039.

[4] D. Karolita., J. Grundy., T. Kanij., H. Obie., J. McIntosh., What's in a persona? a preliminary taxonomy from persona use in requirements engineering, in: 18th Int. Conf. on Evaluation of Novel Approaches to SE (ENASE), SciTePress, 2023. doi:10.5220/0011708500003464.

[5] A. Farooq, A. Alabed, P. S. Msefula, R. A. Tamime, J. Salminen, S. gyo Jung, B. J. Jansen, Representing groups of students as personas: A systematic review of persona creation, application, and trends in the educational domain, Comp. and Ed. Open 8 (2025). doi:10.1016/j.caeo.2025.100242.

[6] J. Cleland-Huang, A. Czauderna, E. Keenan, A persona-based approach for exploring architecturally significant requirements in agile projects, in: J. Doerr, A. L. Opdahl (Eds.), Requirements Engineering: Foundation for Software Quality, Springer Berlin Heidelberg, 2013.

[7] F. Anvari, D. Richards, M. Hitchens, M. A. Babar, Effectiveness of persona with personality traits on conceptual design, in: IEEE/ACM 37th IEEE Int. Conf. on Software Engineering, volume 2, 2015, pp. 263–272. doi:10.1109/ICSE.2015.155.

[8] A. Queirós, A. G. Silva, P. Simões, C. Santos, C. Martins, N. P. d. Rocha, M. Rodrigues, Smartwalk: personas and scenarios definition and functional requirements, in: 2nd Int. Conf. on Techn. and Innovation in Sports, Health and Wellbeing (TISHW), 2018. doi:10.1109/TISHW.2018.8559574.

[9] C. Arenas, L. Garcés, M. J. C. Carmona, C. M. Simões, Requirements specification of a software-intensive system in the health domain: An experience report, in: XIX Brazilian Symposium on Software Quality, SBQS '20, ACM, 2021. doi:10.1145/3439961.3439996.

[10] S.-H. Ho, C. J. Lin, The requirement analysis for developing the assisted living technology for the elderly, in: Cognitive Cities, Springer, 2020.

[11] K. Schäfer, P. Rasche, C. Bröhl, S. Theis, L. Barton, C. Brandl, M. Wille, V. Nitsch, A. Mertens, Survey-based personas for a target-group-specific consideration of elderly end users of information and communication systems in the german health-care sector, Int. Journal of Medical Informatics 132 (2019). doi:10.1016/j.ijmedinf.2019.07.003.

[12] N. Askarbekuly, A. Solovyov, E. Lukyanchikova, D. Pimenov, M. Mazzara, Building an educational product: Constructive alignment and requirements engineering, in: Advances in Artificial Intelligence, Software and Systems Engineering, Springer, 2021.

[13] B. Spieler, V. Krnjic, W. Slany, K. Horneck, U. Neudorfer, Design, code, stitch, wear, and show it! mobile visual pattern design in school contexts, in: IEEE Frontiers in Education Conference (FIE'20), 2020. doi:10.1109/FIE44824.2020.9274120.

[14] M. Oriol, M. Stade, F. Fotrousi, S. Nadal, J. Varga, N. Seyff, A. Abello, X. Franch, J. Marco, O. Schmidt, FAME: Supporting Continuous Requirements Elicitation by Combining User Feedback and Monitoring, in: IEEE 26th Int. Requirements Engineering Conf. (RE), 2018. doi:10.1109/RE.2018.00030.

[15] J. Salminen, K. Chhirang, S.-g. Jung, B. J. Jansen, Helping professionals select persona interview questions using natural language processing, in: 18th IFIP TC 13 Int. Conf. on Human-Computer Interaction (INTERACT'21) Proc. Part 3, Springer, 2021. doi:10.1007/978-3-030-85613-7_20.

[16] A. DeLucia, M. Zhao, Y. Maeda, M. Yoda, K. Yamada, H. Wakaki, Using natural language inference to improve persona extraction from dialogue in a new domain, 2024. arXiv:2401.06742.

[17] S.-M. Yang, J. Lee, W. Cho, Chamain: Harmonizing character persona integrity with domain-adaptive knowledge in dialogue generation, 2024. doi:10.18653/v1/2024.nlp4convai-1.7.

[18] B. Ayoola, M. Kuutila, R. R. Wehbe, P. Ralph, User personas improve social sustainability by encouraging software developers to deprioritize antisocial features, 2024. arXiv:2412.10672.

[19] D. Karolita, J. C. Grundy, T. Kanij, J. McIntosh, H. O. Obie, Lessons Learned from Persona Usage

in Requirements Engineering Practice, in: IEEE 32nd Int. Requirements Engineering Conf. (RE), 2024. doi:`10.1109/RE59067.2024.00021`.

[20] R. Sera, H. Washizaki, J. Chen, Y. Fukazawa, M. Taga, K. Nakagawa, Y. Sakai, K. Honda, Development of data-driven persona including user behavior and pain point through clustering with user log of b2b software, in: 2024 IEEE/ACM 17th Int. Conf. on Cooperative and Human Aspects of Software Engineering, CHASE '24, ACM, 2024. doi:`10.1145/3641822.3641870`.

[21] C. Holmgård, M. C. Green, A. Liapis, J. Togelius, Automated playtesting with procedural personas through mcts with evolved heuristics, 2018. `arXiv:1802.06881`.

[22] C. Holmgård, A. Liapis, J. Togelius, G. N. Yannakakis, Evolving models of player decision making: Personas versus clones, Entertainment Computing 16 (2016). doi:`doi.org/10.1016/j.entcom.2015.09.002`.

[23] S. Ariyurek, E. Surer, A. Betin-Can, Playtesting: What is beyond personas, IEEE Transactions on Games 15 (2023). doi:`10.1109/TG.2022.3165882`.

[24] A. Henka, G. Zimmermann, Persona based accessibility testing, in: HCI Int. 2014 - Posters' Extended Abstracts, Springer, 2014.

[25] A. de Almeida, E. Collins, A. C. Oran, AI in Service of Software Quality: How ChatGPT and Personas Are Transforming Exploratory Testing, in: 23rd Brazilian Symp. on Software Quality, SBQS '24, ACM, 2024. doi:`10.1145/3701625.3701657`.

[26] V. Dantas, Large Language Model Powered Test Case Generation for Software Applications, Technical Disclosure Commons (2023). URL: https://www.tdcommons.org/dpubs_series/6279.

[27] R. France, B. Rumpe, Model-driven Development of Complex Software: A Research Roadmap, Future of Software Engineering (FOSE '07) (2007) 37–54.

[28] J. Michael, B. Rumpe, S. Varga, Human behavior, goals and model-driven software engineering for assistive systems, in: Enterprise Modeling and Information Systems Architectures (EMSIA 2020), volume 2628, CEUR Workshop Proceedings, 2020, pp. 11–18.

[29] H. G. Gurbuz, B. Tekinerdogan, Model-based testing for software safety: a systematic mapping study, Software Quality Journal 26 (2018). doi:`10.1007/s11219-017-9386-2`.

[30] A. Rodrigues, J. Vilela, C. Silva, A systematic mapping study on techniques for generating test cases from requirements, 2024. doi:`10.5220/0012551900003705`.

[31] C. D. Q. Lima, E. L. G. Alves, W. L. Andrade, A systematic literature review on mbt test cases maintenance, in: IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), 2024. doi:`10.1109/COMPSAC61105.2024.00179`.

[32] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, G. Sagardui, Model-based testing in practice: An industrial case study using graphwalker, in: 14th Innovations in SE Conf. (fka India SE Conf.), ISEC '21, ACM, 2021. doi:`10.1145/3452383.3452388`.

[33] T. Pospíšil, Control flow models using petri nets for model based testing, in: 2017 9th IEEE Int. Conf. on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), volume 1, 2017. doi:`10.1109/IDAACS.2017.8095142`.

[34] L. Chen, Y. Gao, C. Liu, W. Wang, Test case generation approach relying on colored petri nets and its application in the r handover process of the train control system, in: 2024 3rd Int. Conf. on Cloud Computing, Big Data Application and Software Engineering (CBASE), 2024. doi:`10.1109/CBASE64041.2024.10824374`.

[35] Z. Xu, F. Sun, W. Zhang, Research on activity diagram testing method based on uml testing profile, in: 6th Int. Conf. on El. Eng. and Inf. (EEI), 2024. doi:`10.1109/EEI63073.2024.10696704`.

[36] P. Jain, D. Soni, A survey on generation of test cases using uml diagrams, 2020. doi:`10.1109/ic-ETITE47903.2020.395`.

[37] T. Aoki, A. Hata, K. Kanamori, S. Tanaka, Y. Kawamoto, Y. Tanase, M. Imai, F. Shigemitsu, M. Gondo, T. Kishi, Model-checking in the loop model-based testing for automotive operating systems, 2023.

[38] A. Khakpour, R. Colomo-Palacios, A. Martini, M. Sánchez-Gordón, The use of domain-specific languages for visual analytics: A systematic literature review, Technologies 11 (2023). doi:`10.3390/technologies11020037`.

[39] N. Baumann, J. S. Diaz, J. Michael, L. Netz, H. Nqiri, J. Reimer, B. Rumpe, Combining retrieval-

augmented generation and few-shot learning for model synthesis of uncommon dsls, Modellierung 2024 Satellite Events, 2024. doi:10.18420/modellierung2024-ws-007.

[40] J. Zhang, Towards the transformation of heterogeneous language components, in: SE 2024 - Companion, Gesellschaft für Informatik e.V., 2024. doi:10.18420/sw2024-ws_18.

[41] E. Chavarriaga, F. Jurado, F. D. Rodríguez, An approach to build json-based domain specific languages solutions for web applications, Journal of Computer Languages 75 (2023). doi:10.1016/j.cola.2023.101203.

[42] K. Hölldobler, O. Kautz, B. Rumpe, MontiCore Language Workbench and Library Handbook: Edition 2021, Aachener Informatik-Berichte, Software Engineering, Band 48, Shaker Verlag, 2021.

[43] H. Krahn, B. Rumpe, S. Völkel, MontiCore: a Framework for Compositional Development of Domain Specific Languages, Journal on Software Tools for Technology Transfer (STTT) 12 (2010).

[44] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, Textbased Modeling, in: 4th Int. WS on Software Language Engineering, Informatik-Bericht 4/07, 2007.

[45] A. Butting, R. Eikermann, K. Hölldobler, N. Jansen, B. Rumpe, A. Wortmann, A Library of Literals, Expressions, Types, and Statements for Compositional Language Design, Journal of Object Technology (JOT) 19 (2020) 3:1–16.

[46] B. Rumpe, Modeling with UML: Language, Concepts, Methods, Springer International, 2016.

[47] I. Drave, J. Michael, E. Müller, B. Rumpe, S. Varga, Model-driven engineering of process-aware information systems, SN Computer Science 3 (2022). doi:10.1007/s42979-022-01334-3.

[48] M. Heithoff, N. Jansen, J. C. Kirchhof, J. Michael, F. Rademacher, B. Rumpe, Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report, in: 16th ACM SIGPLAN Int. Conf. on Software Language Engineerin (SLE), ACM, 2023, pp. 194–207. doi:10.1145/3623476.3623527.

[49] J. C. Kirchhof, B. Rumpe, D. Schmalzing, A. Wortmann, MontiThings: Model-driven Development and Deployment of Reliable IoT Applications, Journal of Systems and Software (JSS) 183 (2022) 1–21. doi:10.1016/j.jss.2021.111087.

[50] J. Michael, B. Rumpe, Software Languages for Assistive Systems, SSRN (2024). doi:10.2139/ssrn.4423849.

[51] B. Rumpe, K. Hölldobler, O. Kautz, MontiCore Language Workbench and Library Handbook: Edition 2021, volume 48 of *Aachener Informatik-Berichte, Software Engineering*, Shaker, Düren, 2021. doi:10.2370/9783844080100.

[52] OMG, Business Process Model and Notation (BPMN), Version 2.0.2, Technical Report, Object Management Group, 2013.

[53] V. Garousi, A. B. Keleş, Y. Balaman, Z. Özdemir Güler, A. Arcuri, Model-based testing in practice: An experience report from the web applications domain, Journal of Systems and Software 180 (2021). doi:10.1016/j.jss.2021.111032.

[54] S. Tiwari, K. Iyer, E. P. Enoiu, Combining Model-Based Testing and Automated Analysis of Behavioural Models using GraphWalker and UPPAAL, 29th Asia-Pacific Software Engineering Conference (APSEC) (2022). doi:10.1109/APSEC57359.2022.00061.

[55] S. Yu, C. Fang, Y. Ling, C. Wu, Z. Chen, Llm for test script generation and migration: Challenges, capabilities, and opportunities, 2023. arXiv:2309.13574.

[56] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, Q. Wang, Fill in the blank: Context-aware automated text input generation for mobile gui testing, in: 45th Int. Conf. on Software Engineering, ICSE '23, IEEE Press, 2023. doi:10.1109/ICSE48619.2023.00119.

[57] J. Michael, V. Shekhovtsov, A Model-Based Reference Architecture for Complex Assistive Systems and its Application, Journal Software and Systems Modeling (SoSyM) 23 (2024) 1247–1274. doi:10.1007/s10270-024-01157-1.

[58] D. Bork, S. Klikovits, J. Michael, L. Netz, B. Rumpe, Inclusive Model-Driven Engineering for Accessible Software, in: 28th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: MODELS-NIER, IEEE, 2025.