Migrating Katalon Studio Tests to Playwright with Model Driven Engineering

Nicolas Hlad[1,*], Benoit Verhaeghe[2]

[1]Berger-Levrault, Labège, France
[2]Berger-Levrault, Limonest, France

Abstract

This work explores a migration process to convert Katalon Studio tests into TypeScript Playwright. We designed this migration as a semi-automated Model Driven Engineering (MDE) process. We used Pharo Moose to design a metamodel of Katalon Studio and export this model into TypeScript Playwright code. In addition, we coordinated teams of Katalon testers and TypeScript engineers to manually check post-migration results and verify the tests. This paper reports our methodology and early results on a closed-source project in our sponsor company, Berger-Levrault. Using a migration strategy based on MDE and minimal manual validation, we managed to reduce the estimated manual migration time by more than 75%.

Keywords

Model Driven Engineering, Test migration, Moose Pharo

1. Introduction

Similarly to source code, tests evolve and need to be migrated to newer platforms or frameworks. In this work, we consider migrating existing functional test cases defined in Katalon Studio to Playwright. Katalon [1] is an automated testing platform designed for web, mobile, API, and desktop applications. It offers an accessible graphical editing interface for beginners, along with advanced options for experienced users. Katalon supports various types of testing, including functional, regression, performance, and behavioral testing, among others. The platform is built on Selenium and Appium, allowing users to define test objects and invoke them using keywords. Figure 2 gives an overview of the Katalon Studio interface and how test cases are declared. It shows that a typical Katalon *test case* is composed of *test steps*. Each step is build with Katalon's *keywords*, that are applied to *objects*, mostly Document Object Model (DOM) objects, of the application under test. Playwright is a framework with similar features to Katalon, but relies on written tests without a dedicated user interface. Playwright supports multiple programming languages, including JavaScript, TypeScript, Python, C#, and Java, and provides a comprehensive set of APIs to automate browser actions, capture screenshots, and generate test reports.

However, Katalon and Playwright tests differ in at least two axes: their *programming languages* and their *frameworks*. We illustrate an example of these differences in fig. 1. Considering the programming language: On one hand, in Katalon, the test case and its resources (objects and custom keywords) are defined using the graphical interface of Katalon Studio and are stored in a Groovy and XML-like combination file (see fig. 1-a). On the other hand, Playwright tests are written manually using languages such as TypeScript (see fig. 1-b). Considering the framework, Katalon and Playwright use different sets of Keyword-Driven API to interact with these platforms that would run the tested software (such as a web browser or a mobile app). With these differences in mind, an expert in both Katalon and Playwright could handle a migration manually, by considering each test case at a time and rewriting each Katalon object and keyword in Playwright. However, lacking experts and time, we explore the possibility of a semi-automated migration.

In this work, we study *how to migrate Katalon tests toward Playwright in a semi-automatic manner*. In related work, Model-Driven Engineering (MDE) has achieved success when used as an intermediary

IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland
*Corresponding author.
nicolas.hlad@berger-levrault.com (N. Hlad); benoit.verhaeghe@berger-levrault.com (B. Verhaeghe)
0000-0003-4989-2508 (N. Hlad); 0000-0002-4588-2698 (B. Verhaeghe)

```groovy
import com.kms.katalon.core.webui.keyword.WebUiBuiltInKeywords as WebUI

// Step 1: Open Browser
WebUI.openBrowser('http://katalon.com')

// Step 2: Click on Login button
WebUI.click(findTestObject('btn_Login'))

// Step 3: Enter Username
WebUI.setText(findTestObject('txt_Username'), 'user@katalon.com')

// Step 4: Enter Password
WebUI.setText(findTestObject('txt_Password'), 'katalon2016')

// Step 5: Click Submit button
WebUI.click(findTestObject('btn_Submit'))

// Step 6: Delay for 0 seconds (can be skipped as it's 0)
WebUI.delay(0)

// Step 7: Click on Community button
WebUI.click(findTestObject('btn_Community'))

// Step 8: Verify Logged-in Icon is present
WebUI.verifyElementPresent(findTestObject('btn_LoggedIcon'), 0)

// Step 9: Close Browser
WebUI.closeBrowser()
```

(a) Katalon source

```javascript
const { test, expect } = require('@playwright/test');

test('Login and verify icon in Playwright', async ({ page }) => {
  // Step 1: Open Browser and go to URL
  await page.goto('http://katalon.com');
  // Step 2: Click on Login button
  await page.click( await Login.BtnLogin(WebUI));
  // Step 3: Enter Username
  await page.fill('#txt_Username', 'user@katalon.com');
  // Step 4: Enter Password
  await page.fill('#txt_Password', 'katalon2016');
  // Step 5: Click Submit button
  await page.click(await Login.BtnSubmit(WebUI));
  // Step 6: Delay 0 seconds (optional, skipped)
  // Step 7: Click on Community button
  await page.click(await Login.BtnCommunity(WebUI));
  // Step 8: Verify Logged-in Icon is present
  await expect(page.locator('#btn_LoggedIcon')).toBeVisible();
  // Step 9: Close Browser — handled automatically by Playwright
});
```

(b) Playwright target

Figure 1: Illustrating the language difference from Katalon Groovy (our source) to Playwright typescript (our target)

transitioning model [2]. In our case, the challenge lies in the lack of an existing metamodel for Katalon, and on to deduce the migration rules from one framework to the other.

2. From Katalon to playwright

We introduce a tool name Katalon2Playwright (or *Ktl2Plw*), implemented in Pharo 12 with the Moose and Famix packages. Pharo is a live and interactive environment to develop in Smalltalk[3]. Moose is a platform to develop and navigate models in Pharo, allowing querying and visualizing those moose with graphical interface [4]. Finally, Famix is used as a set of pre-builds model's traits use to build new Moose metamodel [5].

2.1. Designing the MDE migration process

Relying on previous migration experience applying MDE, we adopt the *horseshoe* [6] migration strategy and perform it as 3 main steps. As illustrated in the C4 model fig. 3, a Pharo user setup a playground script were one can: 1) run a cleaning script on Katalon source project to ease the migration applying the quality first principle [7]; 2) automatically import the Katalon project as a model instance of our Katalon metamodel; and 3) export the model targeting the Playwright environment.

We build our metamodel by gathering concepts and their relations from the Katalon Studio documen-

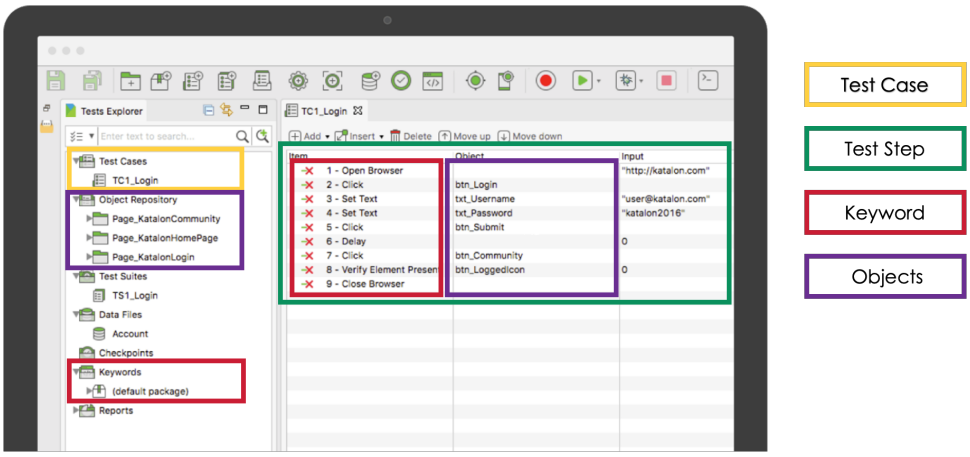

Figure 2: Katalon Studio and its test cases elements

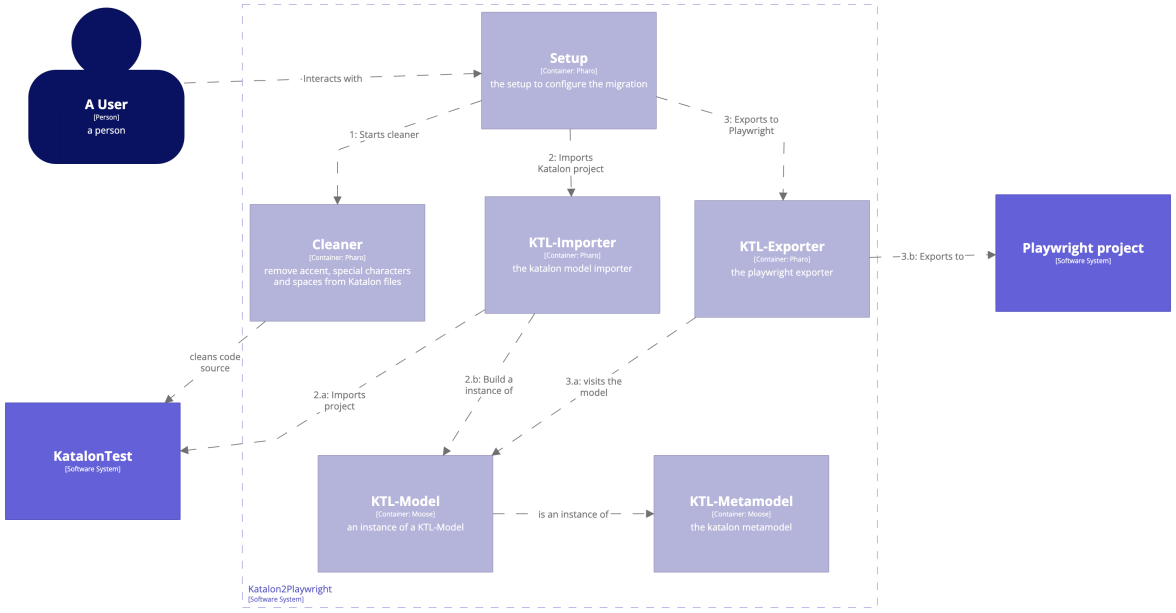

Figure 3: C4 model [8] of migrating Katalon tests using *Ktl2Plw*

tation[1] and papers [1, 9]. A simplified version of our metamodel is shown in fig. 4.

However, our metamodel contains some pitfalls. For instance, it combines concepts of Katalon tests with concepts from what could have been a proper Groovy metamodel. In our current implementation, the metamodel is composed of 33 concepts, with 13 concepts directly linked to Groovy entities.

Our cleaning step duplicates and refactors the Katalon source folder to ease some migration rules. For example, a Katalon test case script is named after its parent folder. The migration rule is to create a Playwright test case identifier named after this folder. However, folder names can include characters like spaces, which are not accepted in a Playwright identifier. We correct this by renaming the folder without spaces.

The import step is divided into two substeps: the parsing and the symbolic resolution. The parsing substep visits all the files of the Katalon project folder to ensure the creation of an instance in the model for each entity (e.g., test case, test steps, object, custom keyword, etc). This substep relies on the tree-sitter tool to parse the Groovy code of Katalon. We develop an open-source Foreign Function Interface (FFI) project in Pharo to interact with the tree-sitter parsing library [10]. For XML-like files,

[1] https://docs.katalon.com/

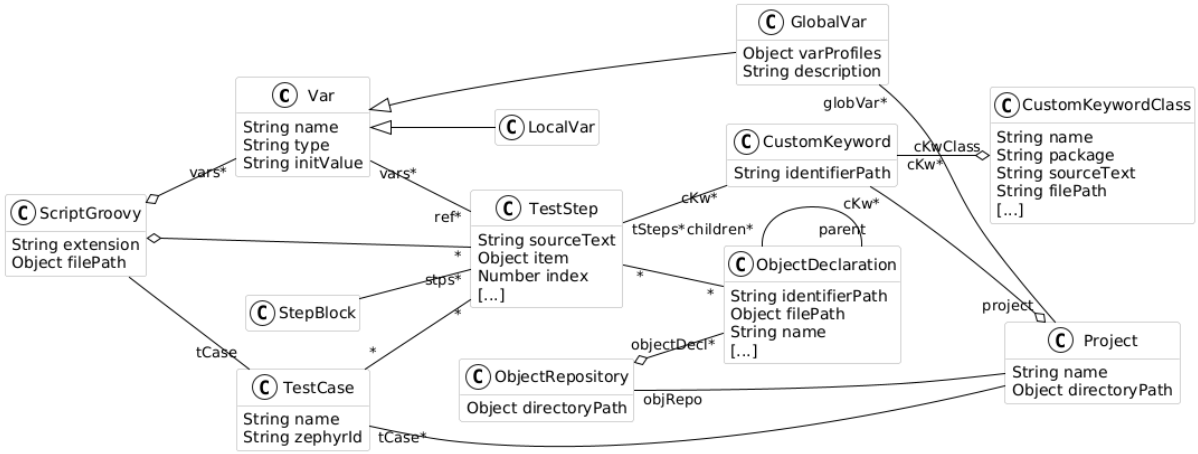

Figure 4: A simplified UML view of our Katalon Studio metamodel (excluding TestStep subconcepts)

we parse them with the Pharo built-in package for XMLDocument. Once done, the symbolic resolution substep builds the relation between the entities. For example, a test step can reference another test case using the `findTestCase` keyword (see fig. 6-a at line 4). Thus, the model's `TestStep` are re-visited to find their relations with other model entities such as `TestCase` or `ObjectDeclation`. Finally, the export step visits the model entities to output them inside a Playwright project folder. To ease the conversion of Katalon action keyword, we develop a TypeScript library, named *KeywordLib*, where each keyword is a function implementing the proper corresponding Playwright behavior. In fig. 5, we illustrate how we utilize the *KeywordLib* to translate the Katalon test step in fig. 5-*a-i* to our output in fig. 5-*a-iii*. In contrast to the expected equivalent Playwright action in fig. 5-*a-ii*, our library bridges the gap between the Katalon test step and what is expected in a Playwright test. This should allow the original Katalon tester to regain ownership of their migrated test. In fig. 5-*b*, we see that the actual *KeywordLib* implementation of `openBrowser` uses the `goto` step found in Playwright, just like in fig. 5-*a-ii*.

(a) Migrating a Katalon's keyword into Playwright code with *KeywordLib*

```
async openBrowser(url? : string,flowControl? : FailureHandling) {
    // […]
    if(url != undefined && url.trim() != '') {
        await this.page.goto(url);
    }
}
```

(b) Matching Katalon's keyword implemented in our *KeywordLib* library

Figure 5: Example of migrating the `openBrowser` Katalon keyword with our transition *KeywordLib* in for playwright.

2.2. Deploying the migration

The migration process involved three primary stakeholders: the original testers of the Katalon tests, the IDM engineers responsible for the migration tool in Pharo (*Ktl2Plw*), and the Playwright engineers, who were tasked with verifying the migrated code.

Each Katalon project was shared independently of the execution environment of its Applications Under Test (AUT). Consequently, it was not feasible for us to re-execute the original source code and verify the functionality of the tests before and after migration. Our focus was on delivering syntactically correct TypeScript Playwright code post-migration. We utilized the Playwright linter to identify syntax and environmental issues in the Playwright projects. Thus, we setup a new playwright environment in VSCode. We then open each file to report on the linter found issues (a.k.a. precompilation *problems*).

(a)

Scripts > Urban_Viewer > URB-T406_Visor_mapa_OpenTopoMap > Script1733412913159.groovy

```
3  // Test case references
4  WebUI.callTestCase(findTestCase('Administrador_y_gestion_de_permisos/URB-T333_Login_UV'),
5      [:], FailureHandling.STOP_ON_FAILURE)
6
7  // Katalon object reference
8  WebUI.waitForElementVisible(findTestObject('Object Repository/Edicion_visor/p_Visores'), 60)
9
10 //keyword + system call
11 WebUI.takeFullPageScreenshot(System.getProperty('user.dir') + '/Pantallazos/imagen_T4046.png')
```

(b)

tests > Scripts > Urban_Viewer > URB-T406_Visor_mapa_OpenTopoMap.spec.ts > ...

```
2  import { WebUILib } from 'Lib/WebUI';
3  // referencing test and objects
4  import { URBT333LoginUV } from 'Scripts/Administrador_y_gestion_de_permisos/URB-T333_Login_UV';
5  import { EdicionVisor } from 'Object Repository/EdicionVisor';
6
7  // Scripts/Urban_Viewer/URB-T406_Visor_mapa_OpenTopoMap/Script1733412913159.groovy
8  test('URB-T406_Visor_mapa_OpenTopoMap', async ({ page }) => {
9      const WebUI = new WebUILib(page)
10     // Test case references
11     URBT333LoginUV(WebUI);
12
13     // Katalon object reference
14     await WebUI.waitForElementVisible(await EdicionVisor.PVisores(WebUI), 60);
15
16     //keyword + system call
17     await WebUI.takeFullPageScreenshot(
18         System.getProperty('user.dir') + '/Pantallazos/imagen_T4046.png');
19 });
```

Figure 6: Migrating Katalon Test case with keywords and objects (a) to Playwright (b) with our tool

Once identified, the Playwright engineer had the option to manually correct the issue or, if the problem was recurrent, escalate it to the IDM engineers to find a solution during the migration process. This decision was primarily based on the recurrence of the issue and the feasibility of establishing a migration rule to resolve it definitively.

The migrated project was then sent to the original testing team, who were responsible for connecting these new tests to their execution environments on their respective AUT. In the event of any issues, they could either correct the problem themselves or provide feedback to the Playwright engineer to find a solution. This collaborative approach ensured that the migrated code was not only syntactically correct but also functionally valid in the target test environment.

3. Early results

We gathered migration data from closed source tests from our company sponsor, Berger-Levrault. We've been given access to 13 Katalon projects. An example of one file is given fig. 6. We migrated these projects and analyzed each output inside a Playwright environment. With the latter, we collect compilation problems detected by the static analysis linter of TypeScript with Playwright in VSCode. The results are shown in table 1. On average, we observe a migration time of 16840 msec per project (excluding the cleaning steps). Compilation problems remain to be investigated, but we noticed that some come from missing keyword correspondences or standard library (as seen in line 18 in fig. 6-b).

This migration tool was built in close collaboration with the teams of each original Katalon tests. We manage to compute an approximation of the total hours of work saved by this semi-migration process. Considering an estimated 6 hours[2] of manual migration per test case from scratch, and around

[2]Estimated by the Playwright engineers

40 hours to design and code the *KeywordLib* library. We computed an estimated 8512[3] hours of manual migration for all test cases only.

With *Ktl2Plw*, we migrated all projects in 218920 msecs, to which we add an estimated 1.3 extra hours of manual checking, by test case[4]. Thus making 1875.7 hours[5] of approximated migration time. It represents an extrapolating saving of -77.96% from the estimated manual migration.

project	Object	TC[2]	Ckw[3]	Migr[4]	#Pb_BMC[5]	#Pb_AMC[6]
paReg	378	44	6	113276	1385	605
paPart	1239	116	0	9332	84	82
paUrb	410	39	0	4217	101	41
paAdm	136	13	0	5880	71	68
paEmp	5	1	0	4179	0	0
paRen	1649	118	0	24852	1056	365
sec	433	69	0	6763	219	24
csaKvP	785	408	0	13908	98	32
taBl	642	171	0	6897	297	240
subv	486	98	0	9630	101	41
fdw	829	160	10	10565	427	297
pruBas	512	109	1	3295	563	313
notiFica	441	66	0	6126	101	5
Average				16840	346.5	145.9

Table 1
Results of the migrated Katalon projects at Berger-Levrault. [2]Test Case; [3]Custom Keyword; [4]Migration Time in msec; [5]Problems Before Manual Checking; [6]remaining Problems After Manual Checking

4. Discussion

In this section we discuss our experiencing pitfalls of building the Katalon metamodel and our observed necessity of a semi-automated migration approach.

4.1. Pitfalls of a Hybrid Katalon/Groovy Model

Our experience in designing the Katalon meta-model highlights several challenges and insights. We adopted an incremental approach, migrating a project through multiple iterations and feedback loop, each adding the import and export of a new Katalon concept. We began with test cases, followed by objects repository, and concluded with custom keywords, thus growing our metamodel's concepts each time. However, we quickly encountered non-trivial issues with importing test cases, as they often contained more complex code structures than a simple sequence of keyword actions on objects. These structures included conditional statements (if/else), switch statements, try-catch blocks, and other constructs defined in the Groovy grammar, despite the primary definition of tests through the Katalon Studio interface.

To address this, we chose to incorporate these code structures into the Katalon model, resulting in a hybrid Katalon/Groovy meta-model. This decision increased the complexity of maintaining the meta-model. In retrospect, it would have been more effective to divide the meta-model into separate Katalon and Groovy components, which could then be connected as sub-meta-models within a common super-meta-model.

[3] $KeywordLibTime + (6 * totalTestCase)$
[4] We obtain this number by asking the Playwright and Katalon teams *"How many time do you spend on validating each migrated test case ?"*
[5] $KeywordLibTime + TotalMigrationWithKtl2Plw + (1.3 * totalTestCase)$ (with $totalTestCases = 1412$, see table 1)

	Code	Message	File	Source
⊗	ts(1005)	',' expected.	tests/Scripts/Administrador_y_gestio...	ts
⊗	ts(2554)	Expected 1 arguments, but got 3.	tests/Scripts/Administrador_y_gestio...	ts
⊗	ts(2339)	Property 'takeFullPageScreenshot' does ...	tests/Scripts/Administrador_y_gestio...	ts
⊗	ts(2339)	Property 'takeFullPageScreenshot' does ...	tests/Scripts/Administrador_y_gestio...	ts
⊗	ts(2304)	Cannot find name 'DriverFactory2'.	tests/Scripts/Administrador_y_gestio...	ts
⊗	ts(2304)	Cannot find name 'DriverFactory'.	tests/Scripts/Administrador_y_gestio...	ts
⊗	ts(2304)	Cannot find name 'System'.	tests/Scripts/Administrador_y_gestio...	ts

Figure 7: Playwirght's linter (in VScode) reporting some issues found in the exported tests of `subv`

4.2. Necessity and Benefits of Semi-Automatic Migration

Our semi-automatic approach proved essential, as numerous issues persisted in the code post-migration. Manually correcting these issues was often simpler than defining specific migration rules. We illustrate here the case of `subv` and its residual issues following the automatic migration, as well as their persistence after manual checks. Among the issues reported by the IDE linter, we identified three main classes: *syntax*, *Unknown Identifier*, and *Missing Property*. Syntax errors include issues such as missing commas or parentheses (e.g., `',' expected.`), unterminated strings or missing expected arguments in method calls (e.g., `Expected 1 arguments, but got 3.`). Unknown identifier errors occur when static analysis fails to locate the declaration of a variable (either global or local) based on its name (e.g., `Cannot find name System`). Lastly, missing property issues indicate that the accessed property does not exist on the given object type (e.g., `Property 'setEncryptedText' does not exist on type 'KeywordLib'`).

The table 2 categorizes the issues found in the `subv` project into the three classes defined above. Syntax errors were mainly caused during the export of model data to Playwright. Some exports were generated using string templates, which may introduce such errors. While we did not investigate all potential root causes, it appears they stem from errors in transforming Groovy test names (strings) into valid Playwright TypeScript identifiers. During our early results, we also observed that some syntax errors were caused by parsing issues. These were due to anomalies in the original Groovy code, such as unresolved Git merge conflicts (often found in obsolete tests) or the presence of emojis, which led to encoding errors in the Tree-Sitter parser. These problems have since been resolved, but we have not yet fully traced all the root causes of the remaining syntax issues.

Unknown identifier issues may arise from undeclared identifiers in the original test (e.g., deprecated ones), or from omissions during model import.

Finally, missing property issues are primarily due to unresolved equivalences, despite the library support. We prioritized re-implementing the most commonly used Katalon keywords across our projects, beginning with the simpler ones. Out of roughly one hundred keywords, *KeywordLib* currently supports 52. Consequently, some less frequently used keywords remained unimplemented during the initial development phase.

In most cases, as observed with `subv`, the remaining errors are mainly of the *Unknown Identifier* type. As illustrated in fig. 6-b, *Ktl2Plw* lacked migration rules to convert the Groovy `System` class (line 11) into its Playwright equivalent (line 18). Manually correcting these issues was often simpler than defining specific migration rules. We worked to implement some of the missing keywords and updated *KeywordLib*. However, certain complex keywords, such as those related to authentication or encryption, were considered too difficult to re-implement in Playwright and were thus excluded.

Thus, migrating Katalon/Groovy to Playwright/Typescript using static rules is challenging due to the nuances in syntax and framework API. Future researches could explore the potential of using Large Language Models (LLMs) to target specific parts of the code that are difficult to migrate with static rules. Code transformation with LLM is a research topic that has received many positive academic and industrial research coverage in the last 2 years [11, 12, 13]. By leveraging LLMs, we can further

subv	Syntax	Unknown Identifier	Missing property	Total
Pb_BMC	16	44	41	101
Pb_AMC	0	40	1	41

Table 2
Remaing issues types inside subv project, before and after manuel checking

automate the migration of complex code structures, reducing the manual effort required and improving the accuracy of the migration process.

5. Related Work

To the best of our knowledge, no prior work has addressed the migration of Katalon Studio tests to Playwright using model-driven engineering (MDE). Moreover, we found no industrial or academic software projects that target these challenges, with or without MDE support. Therefore, we broaden our related work to include both the migration of functional tests in general and the application of MDE techniques to test-case migration.

The authors of [14] propose a model-driven *horseshoe* reengineering method, inspired by [6], to co-migrate large test-case suites alongside the migration of the original tested software. Their implementation achieves an automatic migration of 92% of 4000 JUnit tests to C#'s MSUnit in an industrial setting. By contrast, our work focuses on semi-automated mapping of Katalon's keyword-object XML/Groovy artifacts into Playwright's code-centric APIs, facilitating the migration of functional (UI) tests with minimal expert intervention.

Similarly to our approach, the work of [15] advocates for a semi-automated approach where humans check the migrated tests, here through merge-request validation. However, our work differs since they rely on an AST transformation technique, from JUnit to Gtest. While we use a parsing tool, Tree-Sitter, and its AST only to import the parsed tests into our model instance.

6. Conclusion

Our migration shows that MDE is effective for test case migration. Using Pharo's FFI, we connected to tree-sitter's external analysis library, showcasing Pharo's interoperability. However, projects with specific test behaviors coded in Groovy without keywords leads to limitations. Future work will explore using Large Language Models (LLMs) to migrate remaining code snippets, combining MDE and LLM in a hybrid approach. Future work can improve our contribution by further comparing our migration tool with a manual approach and by improving our tool. For the later, a possible evolution is to add a Groovy-specific metamodel and connect it with the Katalon metamodel. This would allow the Groovy metamodel to be reused outside of Katalon migration, simplifying its maintainability. Furthermore, future works could study the impact using LLM to remove some of the remaining problem caused by translating the tests language from Groovy to Typescript.

7. Declaration on Generative AI

During the preparation of this work, the author(s) knowledge a limited used of TexGPT (from Overleaf) for grammar and spelling check. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Z. Ereiz, Automating web application testing using katalon studio, Zbornik radova Međunarodne naučne konferencije o digitalnoj ekonomiji DIEC 2 (2019) 87–97.

[2] B. Verhaeghe, N. Anquetil, A. Etien, S. Ducasse, A. Seriai, M. Derras, Gui visual aspect migration: a framework agnostic solution, Automated Software Engineering 28 (2021) 6.
[3] A. Bergel, D. Cassou, S. Ducasse, J. Laval, Deep into Pharo, Square Bracket Associates, 2013. URL: https://inria.hal.science/hal-00858725.
[4] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, M. Derras, Modular moose: A new generation software reverse engineering environment, arXiv preprint arXiv:2011.10975 (2020).
[5] S. Tichelaar, S. Ducasse, S. Demeyer, Famix and xmi, in: Proceedings Seventh Working Conference on Reverse Engineering, 2000, p. 296–298. doi:10.1109/WCRE.2000.891485.
[6] R. Kazman, S. G. Woods, S. J. Carrière, Requirements for integrating software architecture and reengineering models: Corum ii, in: Proceedings fifth working conference on reverse engineering (Cat. No. 98TB100261), IEEE, 1998, pp. 154–163.
[7] L. Włodarski, B. Pereira, I. Povazan, J. Fabry, V. Zaytsev, Qualify first! a large scale modernisation report, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 569–573.
[8] S. Brown, Software architecture for developers, Coding the Architecture (2013).
[9] R. P. Octavially, R. R. Riskiana, K. A. Laksitowening, D. Sulistyo, M. A. Kusumo, N. Selviandro, Test case analysis with keyword-driven testing approach using katalon studio tools (2022).
[10] B. Verhaeghe, A. Hosry, N. Hlad, Evref-bl/pharo-tree-sitter: v1.0.2, 2025. URL: https://doi.org/10.5281/zenodo.15089054. doi:10.5281/zenodo.15089054.
[11] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, D. Kroening, Towards translating real-world code with llms: A study of translating to rust, arXiv preprint arXiv:2405.11514 (2024).
[12] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, Z. Chen, A survey on large language models for software engineering, arXiv preprint arXiv:2312.15223 (2023).
[13] H.-W. Yeh, S.-P. Ma, Y. Chen, Test case migration from monolith to microservices using large language models, in: 2024 IEEE International Conference on e-Business Engineering (ICEBE), IEEE, 2024, pp. 29–35.
[14] I. Jovanovikj, G. Engels, A. Anjorin, S. Sauer, Model-driven test case migration: The test case reengineering horseshoe model, in: International Conference on Advanced Information Systems Engineering, Springer, 2018, pp. 133–147.
[15] Y. Gao, X. Hu, T. Xu, X. Xia, D. Lo, X. Yang, Mut: Human-in-the-loop unit test migration, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–12.