HAL
open science

Challenges & Opportunities in Low-Code Testing

Faezeh Khorram, Jean-Marie Mottu, Gerson Sunyé

To cite this version:

Faezeh Khorram, Jean-Marie Mottu, Gerson Sunyé. Challenges & Opportunities in Low-Code Testing. ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), Oct 2020, Virtual, Canada. 10.1145/3417990.3420204 . hal-02946812

HAL Id: hal-02946812

https://hal.science/hal-02946812

Submitted on 23 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Challenges & Opportunities in Low-Code Testing

Faezeh Khorram
faezeh.khorram@imt-atlantique.fr
LS2N, IMT Atlantique
Nantes, France

Jean-Marie Mottu
jean-marie.mottu@ls2n.fr
LS2N, Université de Nantes, IMT Atlantique
Nantes, France

Gerson Sunyé
gerson.sunye@ls2n.fr
LS2N, Université de Nantes
Nantes, France

ABSTRACT

Low-code is a growing development approach supported by many platforms. It fills the gap between business and IT by supporting the active involvement of non-technical domain experts, named Citizen Developer, in the application development lifecycle.

Low-code introduces new concepts and characteristics. However, it is not investigated yet in academic research to point out the existing challenges and opportunities when testing low-code software. This shortage of resources motivates this research to provide an explicit definition to this area that we call it Low-Code Testing.

In this paper, we initially conduct an analysis of the testing components of five commercial Low-Code Development Platforms (LCDP) to present low-code testing advancements from a business point of view. Based on the low-code principles as well as the result of our analysis, we propose a feature list for low-code testing along with possible values for them. This feature list can be used as a baseline for comparing low-code testing components and as a guideline for building new ones. Accordingly, we specify the status of the testing components of investigated LCDPs based on the proposed features. Finally, the challenges of low-code testing are introduced considering three concerns: the role of citizen developer in testing, the need for high-level test automation, and cloud testing. We provide references to the state-of-the-art to specify the difficulties and opportunities from an academic perspective. The results of this research can be used as a starting point for future research in low-code testing area.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Low-code, Testing, Low-code Development Platform, Citizen Developer

ACM Reference Format:
Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. 2020. Challenges & Opportunities in Low-Code Testing. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3420204

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada
© 2020 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada*, https://doi.org/10.1145/3417990.3420204.

1 INTRODUCTION

A Low-Code Development Platform (LCDP) is a software on the cloud whose target clients are non-programmers aimed at building applications without having IT knowledge. It migrates the application development style from manual coding using traditional programming languages into interacting with graphical user interfaces, using prebuilt components, and setting configurations. The user interfaces, business logic, and data services are built through visual diagrams, high-level abstraction, declarative languages, and in specific cases by manual coding. Less traditional hand-coding causes more speed in application delivery and thus less cost [35, 37].

The main user of LCDP, called Citizen Developer, is a domain expert with no programming knowledge [35]. Ease of use and simplicity, from the citizen developer's point of view, is the determining success factor of an LCDP. The number of low-code development platforms is growing as they have been highly requested by citizen developers. These platforms fill the gap between business and IT through abstraction and automation, so they improve the quality of the final product and accelerate the release time.

Independently from the tool or platform used for system development, the system needs to be tested to ensure that all the requirements are realized. Each development approach, including low-code, has its own features and requirements that have to be considered in both system development and testing to provide the highest level of confidence to the final product.

The Citizen developer has a major role in both system development and testing in LCDPs, but his lack of technical IT knowledge leads to the emergence of new requirements and challenges in the context of low-code. There are many success stories of existing commercial LCDPs [26, 37], which demonstrate they overcome the challenges somehow, however, there is a vendor lock-in that confines the existing resources. In addition to commercial tools, to the best of our knowledge, there is no academic research that precisely describes the issues involved, especially for low-code testing; we use the term 'low-code testing' to denote the testing approaches considering the low-code context.

The quality of the system built using an LCDP has to be assured, so a dedicated testing component which satisfies low-code principles is required for LCDPs. As far as we know, there is no research at the moment which specifies the features of such testing component and the potential techniques which can be used for its development. In other words, according to Mary Shaw classification of the phases of research in software engineering [30], the research in low-code testing is in the first stage that is basic research since there is no formal structure to the ideas, concepts, and research questions of this area. Overall, this lack of resources for low-code testing motivates this research.

In this paper, we identify the challenges and discuss their difficulties and opportunities in low-code testing to prepare a roadmap for future academic research in the context. To this end, the testing components of five commercial LCDPs are discovered primarily and then are analyzed based on a detailed set of features for low-code testing which is proposed in this research. Afterward, the existing challenges of the domain, and consequently their associated difficulties and opportunities are described in a research-centric approach by providing related work in the state-of-the-art.

The rest of this paper is organized as follows: Section 2 describes the background of the low-code domain. In Section 3 an introduction to five commercial LCDPs and their testing services is provided. Section 4 introduces a feature list for low-code testing, and also the status of the testing components of selected LCDPs based on that list. The challenges in low-code testing area and the opportunities for further research in this context are presented in Section 5. The paper concludes with a discussion of the limitations and future work in Section 6.

2 BACKGROUND

Low-code is an alive and in-progress domain that requires new techniques and tools proposal for resolving the existing challenges or for realizing new requirements. This needs an understanding of the theory behind the low-code domain that is described in this section.

2.1 Domain-Specific Language

A Domain-Specific Language (DSL) is a computer language specialized to a particular application domain that enables domain experts to create a system using concepts they are familiar with. A DSL has to be designed in a way to be understandable for humans while executable by machine. For example, SQL is a well-known DSL specific for manipulation of databases [8].

Regarding the main objective of LCDPs, i. e., providing system development facilities for domain experts, DSLs are the underlying theory in the LCDP development. The target application domain of an LCDP, or more specifically, the aspects of a system that are modeled in that LCDP, defines which kind of DSLs are used on its basis. For instance, the Business Process Model and Notation (BPMN) is a well-known DSL for modeling business processes. It is used in Mendix LCDP to enable users to develop applications for automating the business processes of their organizations [31].

2.2 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software development methodology that uses models as the pivotal elements in the development. Model is an abstract representation of a system that conforms to a specific metamodel (i. e. the abstract syntax of a specific DSL), and is also independent of the technologies. To build a system following MDE principle, a domain expert first models the application domain manually, and then the models are automatically transformed to either intermediate models (i. e., model-to-model transformation) or source code (i. e., model-to-text transformation) [6]. In the first case, a model-driven execution platform is used for interpreting and running the intermediate models at runtime, while in the second, code generation engines produce executable code of the system (i. e., editable source code or bytecode) that can be executed at runtime environments [5]. In both cases, the transformation is implemented once by language engineers, and then is used several times to auto-generate many systems of the same type. Abstraction along with automation resulted in simplicity, reusability, higher accuracy, portability, interoperability, complexity management, lower cost, and faster release time [6], and these are the potential advantages offered by LCDPs.

LCDPs follow MDE principles. Indeed, they support system design through visual modeling, and automatic generation of the executable final system following two distinct architectural approaches. Some LCDPs, such as OutSystems, use code generation engines to produce executable code, and others, such as Mendix and Lightening, use a model-driven execution platform [5].

3 TESTING IN COMMERCIAL LCDPS

We previously mentioned that there is a lack of research in low-code testing and the objective of this paper is to define it precisely from an academic perspective. To this end, we initially conducted an analysis of the testing components of five LCDPs, to figure out what is offered by successful commercial platforms. They are selected based on the recent reports of Forrester [26] and Magic Quad Quadrant [37] in the low-code context. According to the reports, these LCDPs are known successful since they have a good level of market presence and are called leaders in the low-code community.

3.1 Mendix

Mendix LCDP is introduced for application development on the web, mobile, and IoT platforms. It includes two IDEs to support both no-code and low-code. The former is a drag & drop web-based studio providing pre-built reusable components, while the latter is an IDE for experienced developers to integrate models (e. g., data models, UI models, and microflow models) with manually written code [34].

Testing in Mendix: Quality assurance in Mendix is performed using several tools and services, some of which are for testing while the others help to enhance the quality of the application.

Unit Testing Module is a Mendix-dedicated module for unit testing of the application's logic (i. e., microflow models). The unit tests can be created using microflows and JUnit operations without writing any code [33]. For supporting other kinds of testing, Mendix recommends the use of commercial tools such as SoapUI[1] for automated integration and API testing, Selenium IDE[2] for browser-based UI and acceptance testing, and TestNG [3] for scripting automated tests in Java language [33].

Moreover, there are three quality add-ons provided by Mendix which are not testing tools, but their usage improves the quality of the application: 1) *Application Test Suite (ATS)*: ATS is a set of tools built on top of Selenium [29] for embedding test into application lifecycle; 2) *Application Quality Monitor (AQM)*: By this service, the application models are analyzed statically and the technical quality of the application is calculated based on a subset of features of software maintainability derived from ISO 25010 [7]. The features

[1] https://smartbear.com/product/ready-api/soapui/overview/
[2] https://www.selenium.dev/selenium-ide/

are analyzability, modifiability, testability, modularity, and reusability; 3) *Application Performance Diagnostics (APD)*: This is a cloud service responsible for performance monitoring. It contains a set of tools including the Trap tool that records all levels of logging and stores them when an error occurs, the Statistics tool which identifies trends from application performance statistics, the Performance tool that analyzes individual functions and visualizes where improvement is possible, and the Measurements tool for CPU and Memory monitoring [32].

3.2 Power Apps

Microsoft introduced Power Apps LCDP as a service, for rapid application development across web and mobile, using a suite of apps, services, connectors, and data platforms. The design approach in Powerapps is twofold: 1) *Canvas-based*: By drag & drop elements into a canvas, the application can be designed. Excel-like expressions are used for logic definition, and business data can be integrated from different resources either Microsoft or third-parties; 2) *Model-driven*: This is a component framework usable for professional developers to create custom components, and use them for application development [22].

Testing in Power Apps: This platform offers different testing tools for its different design approaches. A test studio is introduced specifically to support automated end-to-end UI testing of an application designed based on canvas. The test cases can be manually written using Power Apps expressions or automatically generated using record and replay technique [18]. For the model-driven design approach, the automated UI testing framework, Easy Repro[3], that is built for testing Dynamics 365 implementations, can be used. In this tool, UI tests can be defined with no need to parse HTML elements, so the tests became resilient to changes of those. For the server-side testing, Power Apps support integration with existing testing tools such as Fake Xrm Easy[4] which is a .NET mocking framework and can be combined with other .NET testing frameworks if needed.

3.3 Lightning

Salesforce published its own LCDP named Lightning which is mainly focused on Customer Relationship Management (CRM). External systems such as Enterprise Resource Planning (ERP) or data from a connected device can be integrated into the processes defined in the platform. High-level support for blockchain and AI are the outstanding features of this LCDP. Another novel feature of Lightning is its Object Creator tool which enables any employee to turn spreadsheets such as Excel and .csv files into modern cloud-based applications [27].

Testing in Lightning: There are many testing tools built by Salesforce that can be used in Lightning as well, such as Salesforce CLI[5] and Apex test execution[6]. The main shortcoming of which is the need for technical knowledge and low-level of automation [28]. To overcome these drawbacks, integration with Selenium [29] and AccelQ testing platform [15] is provided to support UI, API, and End-to-End automated testing. In AccelQ, the tester uses English natural language to write test cases and Selenium is used for test execution.

3.4 Temenos Quantum

Temenos Quantum LCDP owned by Kony is designed to build mission-critical applications. It leverages novel technologies such as intelligent chatbots, conversational applications, augmented reality, and AI to build modern web and mobile applications [16].

Testing in Temenos Quantum: Automated testing of multi-channel applications is supported by Quantum Testing Framework (QTF) that is integrated with TestNG [3] and Jasmine[7]. TestNG is a Java testing framework inspired by JUnit and NUnit which uses Selenium server for test execution, while Jasmine is an open-source JavaScript testing framework that follows the Behavior-Driven Development (BDD) technique [23] and introduces new syntax for writing unit tests. QTF allows test case creation through recording the user activity on the application and also manual coding [36].

3.5 OutSystems

OutSystems LCDP is well-known for building enterprise solutions as it provides many services for the context of case management, business process management, legacy modernization, and business operations [25]. In this platform, data models, business logic, workflow processes, and UIs had to be defined by models to build the final product.

Testing in OutSystems: Quality assurance in OutSystems LCDP is regarded as it offers several tools and techniques for it. The most important of which are: 1) *OutSystems Test Framework*: It is composed of three frameworks to support software testing at different levels (i. e., unit, integration, system, etc). One of which is a BDD framework that offers a no-code test environment to create unit and API test cases in BDD style [23]. The other one is the Unit Testing Framework which enables testers to define unit tests on the logic of the system without any dependency on the UI. Ghost Inspector[8] automated testing tool is also integrated with the framework to support automated UI testing. Besides the mentioned tools, if further features are required for testing a system in OutSystems LCDP, it is possible to integrate other testing tools within the framework [24]; 2) *Performance analyzer*: By capturing real-time data (e. g., fast, fair, or slow response time distribution), it provides detailed reports to help application performance monitoring during its usage growth; 3) *Automated UI testing with Selenium*: By Selenium IDE, UI test steps can be defined by recording user interaction with UI pages, using reusable scripts with some adjustments if needed, or manual scripting, while Selenium WebDriver enables writing complex test scenarios by common OO languages like Java, JS, and so on [29].

4 FEATURES OF LOW-CODE TESTING

Characterizing low-code testing is essential for performing a systematic comparison between the testing components of commercial LCDPs, and consequently for finding the gaps in the state-of-the-art to enable researchers to work on them.

In this section, we propose a set of 16 features customized for low-code testing. They are defined based on the low-code principles

[3] https://github.com/Microsoft/EasyRepro
[4] https://github.com/jordimontana82/fake-xrm-easy
[5] https://developer.salesforce.com/tools/sfdxcli
[6] https://developer.salesforce.com/docs/atlas.enus.apexcode.meta/apexcode/apex_testing.htm
[7] https://jasmine.github.io/
[8] https://ghostinspector.com/

as well as the capabilities and deficiencies of the testing components of commercial LCDPs. These features are indeed the decisions necessary to be made for building a low-code testing component. To this end, we also provide possible (may not complete) values for them to help with this decision making process. At the end of this section, the result of our analysis of the testing components of commercial LCDPs is also presented based on the proposed feature list.

4.1 Description of the Features

Table 1 demonstrates the features, classified in 5 categories, with the possible values for them. Some of the features are general, while the rest are related to different testing activities i. e., test design, test generation, test execution, and test evaluation.

General: This category specifies the high-level features of the test component of an LCDP. Generally, a *Testing Framework* has to be used for building a test component. If the LCDP is going to have such a component, it should be discussed whether a new Low-Code Testing Framework (LCTF) has to be implemented or an existing one, which is not necessarily for the low-code domain, is preferable. In both cases, the *Supported Testing Scale* and the *Verification Support* features have to be determined. The former defines in which levels (unit, integration, system, UI, API, and End-to-End) the behavior of the system can be tested, while the latter specifies the characteristics (functional and non-functional) of the system that can be verified, such as functionality, performance, security, and so on.

The last feature of this category is *Openness to third-party testing tools*. It is a good practice to enable the test component to integrate with other testing tools since it allows reusing the existing resources. Therefore, the technique and the scale of openness should be specified. The integration could be closed, partially open through import/export techniques to reuse testing artifacts of other sources, or completely open via web-technologies.

Test Design: The features of this category are defined by considering the tasks of the citizen developer role in the testing activities. Several roles can be supported in the test design phase and *The Role of Test Designer* feature aimed at defining them. The citizen developer is the expert of the system functionalities which are used for deriving tests. Therefore, in addition to IT developers and technical testers, she should be involved in the test design activity. However, special techniques and tools should be used for supporting *Collaboration on Test Design* to enable multiple people from different backgrounds to collaborate on the testing of the same application.

The *Test Design Technique* affects the collaboration since it defines the method of test case definition; If the technique is too technical, the citizen developer cannot collaborate in test design. According to our investigation on commercial LCDPs, the following techniques are some potential options for low-code testing, each of which able to resolve specific needs: Model-Based Testing (MBT) for supporting abstraction and automation in different levels of testing, Visual/Graphical modeling for designing test cases as graphical test models, Record and Replay for automated UI testing, Artificial Intelligence (AI) for recommending potential test cases, Keyword-driven for writing tests in natural languages such as English, Data-Driven Testing (DDT) for separating test data from test cases and consequently offering reusability, and Behavior-Driven Development (BDD) or Test-Driven Development (TDD) for providing traceability from system requirements to test cases, from the initial steps of the application development lifecycle. This should be noted that the approach used for designing the tests has a direct impact on the quality of the test suites and their adequacy. Additionally, in some approaches such as MBT, there are techniques to evaluate these features automatically.

Usually, various artifacts can be used or will be produced during test design. The next feature, named *Used/Produced Artifacts in Test Design*, is prescribed to define them. For instance, system requirements and/or system models can be used to derive tests directly from them or to be linked to the test cases (e. g., in BDD/TDD method). Thereupon, when a test case fails, it is easy to identify which system requirement or specification is not realized. Besides, in some test design methods, the definition of test-specific artifacts is required, such as test specifications, test models (e. g., in MBT method), and test data (e. g., in DDT method).

LCDPs claim to have faster release time by offering various features, one of which is reusability. This principle should also be regarded in the testing phase to maintain the pace, so we considered *Reusability* as a feature for low-code testing. This feature can be offered by low-code testing component in different ways. For example by providing the possibility of reusing test data/test cases of other sources, offering reusable test cases from a pre-defined repository, supporting the definition of reusable test cases for testing of an application built in an LCDP, which could be reused in the testing of other applications built in the same LCDP, and also supporting the possibility of defining reusable test cases compatible with various LCDPs, which means they can be used in the testing of several applications developed in various LCDPs.

Test Generation: The LCDPs are supposed to provide as much automation as possible in all activities, especially those technical, including test generation. *Automation of Test Generation* feature specifies the level of provided automation in generating tests which could be High, meaning most of the steps are automated and only simple tasks have to be done manually, Medium which means some tasks are automated but some others have to be performed manually (e. g., definition of test data), and Low that refers to no support for automation.

In different levels of automation, especially medium and low, manual scripting is required, e. g., to implement the test cases that are not auto-generated. *Test Script Language* feature is considered since it should be defined which language is supported by low-code testing component for scripting tests. Various languages can be used, such as test-specific DSLs defined by the LCDPs, test-specific languages such as Testing and Test Control Notation version 3 (TTCN-3)[9], and programming languages (e. g., Java).

Test Execution: Automation, distribution, and cloud are the main concerns of the features of this category. LCDPs are cloud-based, they support the development of cloud-based and distributed applications, and they tend to be more scalable. Therefore, the low-code test component needs to support distributed test execution over the cloud, and also to perform this activity in an automated manner.

[9] http://www.ttcn-3.org/

Table 1: Features of low-code testing with some possible values for them

Category	Feature	Possible Values
General	(1) Testing Framework	No support, New Low-Code Testing Framework (LCTF) dedicated to LCDP, Leveraging third-party frameworks (e. g., Selenium, TestNG).
	(2) Supported Testing Scale	Unit, Integration, System, UI, API, End-to-End (E2E).
	(3) Verification Support	Functionality, Performance, Security, Usability, Compatibility, Reliability, etc.
	(4) Openness to other testing tools	Closed, Import/Export of test models, test script, or test data, Integrate via web-technologies (e. g., REST).
Test Design	(5) The Role of Test Designer	Citizen developer (i. e., non-technical tester), IT developer, Technical tester.
	(6) Collaboration on Test Design	No support, Collaborative test design, Continuous feedback mechanism.
	(7) Test Design Technique	Model-Based Testing (MBT): Modeling the System based on a DSL and auto-generating the executable test cases from it, Visual/Graphical modeling of the test cases, Record and Replay for automated UI testing, Artificial Intelligence (AI): Automatic recognition of test cases, Keyword-driven: Using natural languages such as English, Data-Driven Testing (DDT): Separating test data from test cases, Behavior-Driven Development (BDD)/Test-Driven Development (TDD).
	(8) Used/Produced Artifacts in Test Design	None, System requirements, System models (e. g., Data models, Logic models, UI pages), Test specification, Test models, Test data.
	(9) Reusability	Reusing test data/test cases of other sources, Reusable test cases provided by the testing component, The possibility to define new test cases that can be reused in a specific LCDP, The possibility to define new test cases that can be reused in various LCDPs.
Test Generation	(10) Automation of Test Generation	High (support no-code), Medium (support low-code), Low (manual coding).
	(11) Test Script Language	New executable DSLs defined by the platform (e. g., PowerApps expressions), Existing test-specific languages such as TTCN-3, Programming languages (e. g., Java).
Test Execution	(12) Automation of Test Configuration	High (support no-code), Medium (support low-code), Low (manual coding).
	(13) Distribution	Not supported, Distributed test execution.
	(14) Test Execution Tool/Service	New tools provided by LCDP, Third-party tools such as Selenium server.
	(15) Test Execution Platform	Provider cloud, Public cloud, On-premises, Standalone.
Test Evaluation	(16) Test Result Evaluation Technique	Monitoring, Comparison, Visual/textual reporting, Analyzing execution traces.

Automation of Test Configuration feature investigates the level of automation provided by the testing component for performing test configuration. We mentioned earlier that the more automation the LCDP provides, especially for technical tasks, the less time and cost spent on releasing the final application. Therefore, it is required to provide automation for test configuration as well.

Distribution refers to the capability of the low-code test component in distributed test execution. Distributed architectures are highly-used for developing systems, and they are supported in most LCDPs, so the LCDP's test component should be able to test the systems under such architectures. Moreover, the cloud-native of LCDPs provides the infrastructure for executing tests in a distributed manner by leveraging the cloud. Therefore, offering this feature by the low-code test component results in its more functionality.

Test Execution Tool/Service feature defines which tool or service is used in the test component for running executable test cases. LCDPs could propose new tools, however, our analysis on commercial LCDPs demonstrates that integrating third-party tools such as the Selenium server is preferable, even if the LCDP has its own LCTF.

The last feature that we identified in the test execution category is *Test Execution Platform*. It specifies on which platforms the tests can be run. According to our investigation of commercial LCDPs, provider cloud, public cloud, on-premises, and standalone are the possible options of system deployment that are provided by LCDPs. These options could also be supported as a platform for test execution.

Test Evaluation: The low-code test results should be generated in a way to be understandable for all the roles involved in the testing phase. Therefore, several techniques should be used to demonstrate abstract/non-technical results to citizen developers, while presents concrete/technical results to the IT developers and the technical

testers. *Test Result Evaluation Technique* feature is defined to specify which techniques are used for evaluating test results.

4.2 The Status of the Testing Component of Commercial LCDPs

Section 3 presented an overview of the testing facilities, provided by the selected LCDPs. Generally, most of the testing activities are supported by their integrated third-party testing tools such as Selenium, SoapUI, AccelQ, TestNG, GhostInspector, Jasmine, and EasyRepro. In this subsection, we organized our analysis result based on the proposed feature list, to specify the status of commercial LCDPs in providing testing facilities.

(1) Four LCPDs, except Quantum, propose a new Low-Code Testing Framework (LCTF) but with limited capabilities. Consequently, they all provide integration with third-party testing tools to have reasonable coverage of all testing activities.
(2) In general, all testing scales are supported in LCDPs, but mainly by their integrated third-party testing tools. Particularly, the LCTF of Mendix is designed for unit testing, the LCTF of PowerApps for UI and E2E testing, the LCTF of Lightning for unit testing, and the LCTF of OutSystems for unit and API testing.
(3) Functionality and Performance are the features that are continuously tested in all LCDPs, and testing of other non-functional requirements is neglected.
(4) Except for Powerapps, the other LCDPs use web technologies to integrate with third-party testing tools. Test data import/export is also supported by Mendix LCTF.
(5) Technical testers and/or developers are in charge of performing tests using the third-party tools integrated with the LCDPs. The citizen developer is only involved in the testing activities supported by LCTFs and also when Automated UI testing is provided by recording tools.
(6) Collaboration is considered in all LCDPs but mainly between technical developers and testers. Nevertheless, Mendix is the only LCDP which offers an easy to use feedback mechanism for collaboration of non-technical developers.
(7) Among different test design techniques, the LCTF of Mendix and OutSystems follows the MBT approach for performing unit tests (and API tests in OutSystems' LCTF) based on the DSLs they use for logic modeling augmented with testing elements, Record and Replay technique is supported by all LCDPs for automated UI testing, mainly through integration with Selenium IDE, AI is only supported by AccelQ third-party testing tool which is integrated with Lightning LCDP, Mendix, PowerApps, and Lightning follow Keyword-driven and DDT techniques, and Quantum and OutSystems use the BDD style.
(8) Among the various artifacts, Mendix and Lightning use system requirements to explicitly map them to the test cases, UI pages are used in all LCDPs for performing UI tests by capturing UI test specifications through recording tools, OutSystems and Mendix use system models, such as microflows and data flows, in their testing process, and they produces test models since they use graphical modeling for designing unit tests.
(9) Mendix's LCTF and AccelQ in Lightning LCDP offer reusable test case templates, importing test data from external files, and definition of reusable test cases to be used in testing of other applications built using the same platform. Besides, reusing of repetitive interactions is supported in OutSystems.
(10) A medium level of automation for test generation is provided in all LCDPs, but mostly by their integrated third-party testing tools.
(11) The final executable test cases are generated in different programming languages such as Java and C#.
(12) The test configuration is automated at a medium level since manual efforts are still needed in some cases.
(13) Distributed test execution is considered in all LCDPs, but mainly in their integrated third-party testing tools. For instance, Mendix and OutSystems leverage Selenium Grid for this purpose.
(14) All LCDPs, except PowerApps, use Selenium Server as their test execution tool; PowerApps uses Microsoft Dynamics 365.
(15) Provider cloud and on-premises are the supported test execution platform in all LCDPs.
(16) Monitoring and visual and textual reporting are provided in all LCDPs. Mendix and OutSystems also analyze the execution traces and offer notes for improvement.

In conclusion, the results of our evaluation reveales that there are specific features for low-code testing that should always be considered and cannot be neglected in the testing component of LCDPs. They are the role of citizen developer and the side effects of her non-programming knowledge in her involvement in testing, the need for high-level automation, and leveraging the cloud. According to them, the next section rephrases the deficiencies in low-code testing in a research-centric approach through providing related work in academy, and proposing opportunities based on them for future work in this area.

5 CHALLENGES AND OPPORTUNITIES

There is a community of people – LCDP developers – who aim at building new low-code development platforms because there are new application domains, features, technologies, customers, and consequently new requirements for the development of new LCDPs. As depicted in Figure 1, these platforms have two main editors in a nutshell, one for building a software system and another for testing that system. Potentially, the citizen developer (i. e., the user of the platform) works with both editors to design the system and the test cases, for example through visual modeling.

The problematic from the testing point of view is that there is no general framework, to be used by LCDP developers, for building the testing component of their intended LCDPs which fully supports low-code testing features. Lack of such framework resulted in the high-dependency of existing LCDPs to technical third-party testing tools which are not usable for citizen developers. Additionally, although some commercial LCDPs propose new low-code testing frameworks, they do not fulfill all low-code testing features, they are not reusable for other LCDPs, and their resources are not accessible publicly.

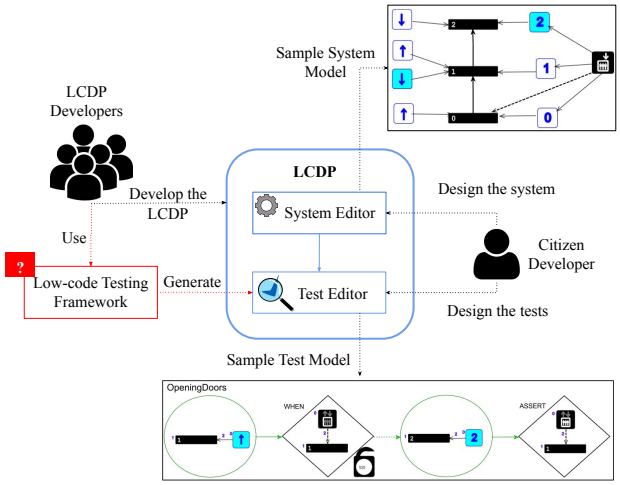

Figure 1: An overview of the main challenges in low-code testing (The model instances are derived from [19])

In the following subsections, we expound the problematic under several challenges, categorized based on the most important features of the low-code context: the role of citizen developer, the need for automation, and the effects of the cloud. Meanwhile, the previous attempts and the potential opportunities to overcome the challenges are also expressed.

5.1 The Role of Citizen Developer in Testing

In a low-code development platform, the citizen developer is responsible for the definition of requirements since she is the expert of the system functionalities. As test cases are mainly derived from the requirements, so she is in charge to define test cases and also to evaluate test results. Therefore, her full involvement in the testing activities, from design to evaluation, is essential for low-code testing but her low-level of technical knowledge causes many challenges, hence new techniques are required.

We previously outlined in Section 2 that LCDPs are built based on specific DSLs. When a citizen developer designs a system in an LCDP, she actually creates instances of the underlying DSL. As she is the domain expert, she can instantiate models from the DSL with low training. If the test cases can be written in the same language as the software (i. e., the same DSL), the citizen developer can design tests with no additional training, so the efficiency increases. An example of this is depicted in Figure 1. The citizen developer designed an elevator by instantiating from a specific DSL. The elevator has one Door, three Floor buttons, two Up buttons, and two Down buttons, and it can stop in three Floors. By using the same DSL augmented with test-specific elements, she designed a test case model to verify the following requirement [19] that we stated in BDD style:

> **GIVEN** the elevator on the first floor with the Up button pressed,
> **WHEN** the elevator's door is closed **AND** the Floor button is pressed on the second floor,
> **THEN** the elevator stays on the first floor **AND** its door becomes open.

Despite the benefits of using the same language for designing software and its tests, especially in the low-code domain, it is rarely used in LCDPs as its implementation resources are limited or are dedicated to specific DSLs and are not reusable. BDD framework of OutSystems LCDP is a successful case of the implementation of such an approach in the real-world. It extends the platform's DSL with testing elements such as Assertions to enable automated Unit and API testing. The test cases are firstly defined textually using Given-When-Then clauses and then each clause is modeled in the same approach as system modeled [24]. As previously described in section 2, OutSystems LCDP uses code generation engines to produce executable code. Consequently, the test models are transformed into Java or C# code to be executed against the system under test.

5.1.1 ***Previous Attempts.*** The mentioned technique i. e., in detail DSL extension with further properties (e. g., testing features) is a language engineering issue that is investigated in several papers. In [20, 21], a framework, named ProMoBox, is introduced that enables DSL engineers to auto-integrate five sub-languages to a given DSL, to support specification and verification of temporal properties of a system modeled using the given DSL. It uses Linear Temporal Logic (LTL) to specify properties and offers a model checking engine plug-able to DSL environments to run and evaluate them.

The ProMoBox framework is restricted to the DSLs whose semantics are described as a rule-based transformation; by this semantics, the system behavior is captured through state changes. Moreover, it is limited to the verification of LTL-based properties. Totally relying on the model checking technique causes the framework's low performance due to high memory usage.

The main issues with the ProMoBox framework were inherited in using model checking. In [19], the framework is adapted to test case generation techniques as it is a valuable alternative to model checking. It proposes an automatic approach to augment a given DSL with testing elements derived from a specific test DSL so that modelers can model functional unit tests in the same language as system models. The model instances in Figure 1 are taken from the running example of this paper.

The testing support of the ProMoBox framework is also restricted to DSLs with rule-based semantics. In addition, it does not support real-time models, other testing scales such as API testing, and distributed test execution since the testing DSL that is used, involves only basic testing elements while there are other testing DSLs covering those of complex. For example, Test Description Language (TDL) is a DSL for high-level test specification that is defined to smooth the transition from system requirements to executable test cases written in TTCN-3; it is itself a test-specific DSL for black-box testing of distributed systems [17].

5.1.2 ***Opportunities.*** Among numerous amount of DSLs, there are many, specific for the testing domain. They can be distinguished based on their support for different testing scale (e. g., Unit, Integration, API), testing type (e. g., Performance, Security, Compatibility), application domain (e. g., mobile, web, IoT), and application deployment (e. g., on-premises, cloud, embedded). One solution to the described shortcomings of the state-of-the-art is the support for other testing DSLs (e. g., TDL) in the DSL extension process. Another opportunity that can be taken into account is proposing a generic DSL extension technique that can support different kinds of DSLs (not just DSLs with rule-based semantics). Nevertheless, this generalization reveals specific challenges since the semantics of DSLs could be defined in different ways (i. e., Interpretation and Compilation), and consequently, for the generation of executable test cases and the interpretation of test failures in the model level, various approaches should be followed.

In addition to the DSL-based opportunities, the research areas such as assistant chatbots and recommendation systems are also topics of interest in the alleviation of the challenges related to the role of citizen developer in testing. In other words, as citizen developers do not have the technical knowledge of testing, even if they can model the test cases in the same language as system models, they need to be assisted on how to correctly use test-specific elements during designing of the test models.

5.2 The Need for High-level Test Automation

Many efforts on test automation are conducted so far, as it saves significant time and effort. Test automation enables continuous quality assessment at a reasonable cost, and this is essential for DevOps. Automation is possible on different kinds of tests such as unit, API, and UI functional tests, as well as load and performance non-functional tests.

The upward tendency towards building multi-experience applications also increases the need for the evolution of test automation. LCDPs are specialists for the development of such applications, consequently, test automation is vital in these platforms. Especially, automated API testing is essential in LCDPs as low-code applications use many integrations to other services using APIs. If these integrations are not continuously tested, the application breaks easily.

In low-code testing, a high-level of automation should be provided alongside a low dependency on technical knowledge. Despite that most of the automated testing tools are very technical and they use manual scripting for writing tests, there are some trends followed by them to facilitate this task. To identify the techniques they use, we made an investigation on some of them, selected from [13].

Briefly, the results of this query along with the information gathered in section 3 revealed that the most popular testing techniques aimed at simplicity alongside automation, are **Data-Driven**, **Model-Based**, and **Record and Replay** which are used almost together. The data-driven testing technique provides reusability of test data through its separation from test scripts, while in record and replay technique UI tests are automated by recording user interactions with UI pages. The Model-Based Testing (MBT) technique is applicable for automating tests on any scale, so it is the most comprehensive approach compared to others. Abstraction and automation are its basic objectives and are offered by visual modeling and transformation engines, respectively. It is especially useful when a test run on several deployment options is imperative, which is a considerably important requirement in low-code testing since LCDPs are supposed to auto-generate applications on several platforms from a single system specification.

The first step in using MBT for testing applications in a particular domain is choosing a modeling language based on that domain i. e., indeed a Domain-Specific Modeling Language (DSML), to model the System Under Test (SUT). In the next step, system models are transformed to the test cases, test scripts, and test data, using several transformation engines. These engines are the pivotal elements in providing automation for test implementation, configuration, and execution on several, yet totally different, platforms. As much as automation the engines provide, the efficiency of MBT increases. The crucial role of transformation engines proves the obligation of tool support in MBT. There are numerous MBT languages and tools, each of which adapted to specific domains (and consequently specific DSLs), testing methods, and coverage criteria. Therefore, MBT tool selection is a challenging task [1].

5.2.1 ***Previous Attempts.*** MBT is a growing research field and many papers in this domain are published each year. The latest mapping study on MBT performed by Bernardino et al. illustrates that from 2006 to 2016, approximately 70 MBT supporting tools are proposed by business and academy while some of which are open source [1]. This significant number of tools promotes the opportunity to create a repository of existing MBT tools which can be analyzed for different purposes, but there is no repository so far.

5.2.2 ***Opportunities.*** The model-based testing is addressed in many papers, but it is not specialized for the low-code context. As we mentioned earlier, low-code development platforms are based on particular DSLs and system modeling is inherent in these platforms. Therefore, for the application of MBT in LCDPs, the first step (i. e., selection of a modeling language) is strictly imposed by the platform. Accordingly, for using MBT in the testing component of LCDPs, two modes exist:

(1) If MBT is already applied to the LCDP's underlying DSL and associated tools exist, an appropriate tool has to be selected from the existing pool.
(2) Otherwise, implementation of new MBT tools adapted to the DSL is required.

Both mentioned modes promote new challenges and thereupon opportunities, since there is neither a pool of existing MBT tools nor a technique or tool to enable the development of new MBT tools for a given DSL. Discovery and retrieval of appropriate MBT tools based on a set of input features (e. g., application domain, input DSL, testing scale), comparison between different tools based on their features for the same testing scale, and composition of compatible MBT tools especially when they are service-oriented, are a few of use cases of the implementation of such opportunities.

5.3 Cloud Testing

Cloud testing can be defined in three aspects: 1) Testing of the Cloud, meaning functional and non-functional testing of cloud-based applications; 2) Testing in the Cloud refers to leveraging

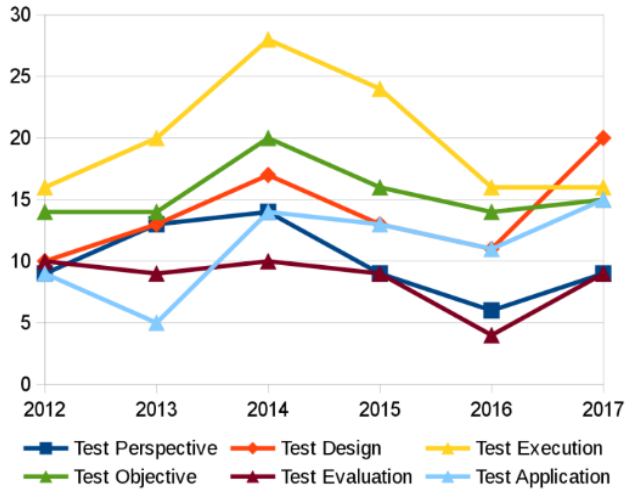

Figure 2: The trends of the areas in cloud testing by year (Taken from [2])

scalable cloud infrastructure, tools, techniques, and computing resources for testing non-cloud applications; and 3) the combination of both which is testing the applications deployed in the cloud by using cloud resources [2].

In 2019, Bertolino et al. performed a systematic review of the cloud testing area [2]. The result of their investigation on 147 papers demonstrated that almost two-third of the state-of-the-art targets the challenges in testing in the cloud, while approximately one-quarter of them target those of in testing of the cloud. Additionally, as can be seen in Figure 2 taken from [2], test design and execution are the most notable areas in cloud testing.

The existing LCDPs are all cloud-based and they support the development of cloud-based applications. Meanwhile, there is an upward trend in low-code context to support the development of large-scale applications, especially for the domains of mobile, web, and Service-Oriented Architectures (SOA) such as Microservices. Overall, as the cloud offers development and maintenance of scalable test infrastructures, and configuration of on-demand scalable resources through cloud virtualization [2], all three aspects of cloud testing have to be provided by LCDPs, especially in those of scalable.

5.3.1 ***Previous Attempts***. Besides the existing challenges and issues described in [2] for the cloud testing in general, the specific features of low-code introduces new ones. As we described in section 5.2, MBT is the most compatible approach with low-code testing. The challenge is how the three paradigms of cloud testing can be provided by MBT techniques and tools.

By MBT, given an abstract picture of the SUT, it is possible to generate many test cases to be executed on the cloud [2]. Several cloud-based MBT frameworks are proposed so far, each of which specialized in different application domains and testing levels.

MIDAS is a cloud-based MBT testing platform for Software-Oriented Architectures (SOA). It supports functional, usage-based, and security testing of individual web services and also their orchestration in SOA applications. Therefore, it provides the third aspect of cloud testing that is testing of the cloud in the cloud. In the MIDAS framework, a new DSL is used for system modeling which is based on the Unified Modeling Language (UML) and the UML Testing Profile (UTP) augmented with SOA-specific features and conditions. Several cloud-based services for test case generation are deployed in MIDAS, each of them uses a distinct test scenario as a basis. Indeed, each service receives a MIDAS DSL model as input and generates test cases for the input model, based on its own test scenario. There are also other services for test case prioritization, scheduling, transformation to the TTCN-3 test code, execution, and arbitration [4, 9–12, 14].

The MIDAS framework only supports testing of SOA applications which are manually modeled using the MIDAS DSL, and which can communicate only via Soap APIs. Besides, the resources for their DSL and the TTCN-3 code generation service are not accessible. These shortcomings lead to its low-level of usage.

5.3.2 ***Opportunities***. We propose the opportunities for cloud-based low-code testing focused on supporting cloud in MBT, according to the opportunities described above for other aspects.

The approach introduced by MIDAS i. e, model-based testing as a service and providing cloud-based services for different testing activities, is very interesting to be continued for other testing DSLs. One considerable opportunity could be the generation of a comprehensive framework that auto-generates test-specific services for a given DSL. In that case, the opportunities written in the previous sections can be seen as different parts of this framework which in total leads to a cloud-based low-code testing framework.

6 CONCLUSION

Due to the high trend toward low-code domain and limited academic resources for low-code testing, we performed several analyses in this article. We initially discovered the testing facilities embedded in five well-known commercial LCDPs. Afterward, we proposed a set of 16 features for low-code testing which can be used as criteria for comparing several low-code testing components, and as a guideline for LCDP developers in building new ones. Accordingly, we organized the result of our analysis on the testing component of selected LCDPs based on the proposed feature list.

Our investigations lead us to the identification of existing challenges in low-code testing. We redefined them from a research point of view by providing the state-of-the-art in three main categories including, the role of citizen developer and her low-level technical knowledge in the testing activities, the importance and consequently the challenges in offering high-level test automation, and leveraging the cloud for executing tests alongside supporting testing of cloud-based applications.

For each category, we also propose opportunities for future research in low-code testing, such as DSL extension with testing elements, customization of MBT for low-code testing, and supporting the cloud in MBT approaches.

As our future work, we will initially work on the challenges of the first category. At the moment, we are defining a running example to show how the DSL extension algorithm would work in practice and how different kinds of DSLs (i. e., interpreted and compiled) could be supported. Afterwards, we will implement the generic extension algorithm, so different system DSLs can be extended

with various test DSLs. Finally, supporting the cloud, and building tools for generating cloud-based low-code testing component for a given LCDP, are in our future research plan.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska Curie grant agreement No 813884.

REFERENCES

[1] Maicon Bernardino, Elder M Rodrigues, Avelino F Zorzo, and Luciano Marchezan. 2017. Systematic mapping study on MBT: tools and models. *IET Software* 11, 4 (2017), 141–155.

[2] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. 2019. A Systematic Review on Cloud Testing. *Comput. Surveys* 52, 5 (2019), 1–42.

[3] Cédric Beust and Hani Suleiman. 2007. *Next generation Java testing: TestNG and advanced concepts*. Pearson Education.

[4] Steffen BHerbold and Andreas Hoffmann. 2017. Model-based testing as a service. *International Journal on Software Tools for Technology Transfer* 19, 3 (2017), 271–279.

[5] Jason Bloomberg. 2018. *Low-Code/No-Code? HPaPaaS? Hereś What Everybody Is Missing*. Retrieved July 21, 2020 from https://www.forbes.com/sites/jasonbloomberg/2018/07/30/low-codeno-code-hpapaas-heres-what-everybody-is-missing

[6] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. Model-driven software engineering in practice. *Synthesis lectures on software engineering* 3, 1 (2017), 1–207.

[7] International Organization for Standardization. 2011. ISO/IEC 25010:2011-Systems and software engineering- Systems and software Quality Requirements and Evaluation (SQuaRE)- System and software quality models. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 Available in electronic form for online purchase at https://www.iso.org/standard/35733.html.

[8] Martin Fowler. 2010. *Domain-specific languages*. Pearson Education.

[9] A. D. Francesco, C. D. Napoli, M. Giordano, G. Ottaviano, R. Perego, and N. Tonellotto. 2014. A SOA Testing Platform on the Cloud: The MIDAS Experience. In *2014 International Conference on Intelligent Networking and Collaborative Systems*. 659–664.

[10] Alberto De Francesco, Claudia Di Napoli, Maurizio Giordano, Giuseppe Ottaviano, Raffaele Perego, and Nicola Tonellotto. 2015. MIDAS: a cloud platform for SOA testing as a service. *International Journal of High Performance Computing and Networking* 8, 3 (2015), 285–300.

[11] S. Herbold, A. De Francesco, J. Grabowski, P. Harms, L. M. Hillah, F. Kordon, A. Maesano, L. Maesano, C. Di Napoli, F. De Rosa, M. A. Schneider, N. Tonellotto, M. Wendland, and P. Wuillemin. 2015. The MIDAS Cloud Platform for Testing SOA Applications. In *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–8.

[12] Steffen Herbold, Patrick Harms, and Jens Grabowski. 2017. Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. *International Journal on Software Tools for Technology Transfer* 19, 3 (2017), 309–324.

[13] Joachim Herschmann, Thomas Murphy, and Jim Scheibmeir. 2019. *Magic Quadrant for Software Test Automation*. Technical Report.

[14] Lom Messan Hillah, Ariele-Paolo Maesano, Fabio De Rosa, Fabrice Kordon, Pierre-Henri Wuillemin, Riccardo Fontanelli, Sergio Di Bona, Davide Guerri, and Libero Maesano. 2017. Automation and intelligent scheduling of distributed system functional testing. *International Journal on Software Tools for Technology Transfer* 19, 3 (2017), 281–308.

[15] AccelQ Inc. 2020. *ACCELQ is on Salesforce App Exchange*. Retrieved July 21, 2020 from https://www.accelq.com/Salesforce-Test-Automation

[16] Kony Inc. 2020. *Leading Multi Experience Development Platform*. Retrieved July 21, 2020 from https://www.kony.com/

[17] Philip Makedonski, Gusztáv Adamis, Martti Käärik, Finn Kristoffersen, Michele Carignani, Andreas Ulrich, and Jens Grabowski. 2019. Test descriptions with ETSI TDL. *Software Quality Journal* 27, 2 (2019), 885–917.

[18] Tapan Maniar et al. 2020. *Test Studio*. Retrieved July 21, 2020 from https://docs.microsoft.com/en-us/powerapps/maker/canvas-apps/test-studio

[19] Bart Meyers, Joachim Denil, István Dávid, and Hans Vangheluwe. 2016. Automated testing support for reactive domain-specific modelling languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. Association for Computing Machinery, 181–194.

[20] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. 2014. ProMoBox: a framework for generating domain-specific property languages. In *International Conference on Software Language Engineering*. Springer, 1–20.

[21] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay. 2020. A Framework for Temporal Verification Support in Domain-Specific Modelling. *IEEE Transactions on Software Engineering* 46, 4 (2020), 362–404.

[22] Microsoft. 2020. *The world needs great solutions, Build yours faster*. Retrieved July 21, 2020 from https://powerapps.microsoft.com/

[23] Dan North et al. 2006. Introducing bdd. *Better Software Magazine* (2006).

[24] Outsystems. 2020. *How to Automate Unit Testing and API Testing*. Retrieved July 21, 2020 from https://success.outsystems.com/Documentation/How-to_Guides/DevOps/How_to_Automate_Unit_Testing_and_API_Testing

[25] Outsystems. 2020. *Innovate with No Limits*. Retrieved July 21, 2020 from https://www.outsystems.com/

[26] John R Rymer and Rob Koplowitz. 2019. *The Forrester Wave™: Low-Code Development Platforms For AD&D Professionals*. Technical Report.

[27] Salesforce. 2020. *Build apps on the Customer 360 Platform with no-code tools and take your CRM to the next level*. Retrieved July 21, 2020 from https://www.salesforce.com/products/platform/lightning/

[28] Salesforce. 2020. *Testing Apex*. Retrieved July 21, 2020 from https://developer.salesforce.com/docs/atlas.en-us.226.0.apexcode.meta/apexcode/apex_testing.htm

[29] Selenium. 2020. *Selenium automates browsers. That's it!* Retrieved July 21, 2020 from https://www.selenium.dev/

[30] Mary Shaw. 2002. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer* 4, 1 (2002), 1–7.

[31] Mendix Technology. 2020. *Microflows*. Retrieved July 21, 2020 from https://docs.mendix.com/refguide/microflows

[32] Mendix Technology. 2020. *Quality Add-ons Guide*. Retrieved July 21, 2020 from https://docs.mendix.com/addons/

[33] Mendix Technology. 2020. *Testing*. Retrieved July 21, 2020 from https://docs.mendix.com/howto/testing/

[34] Mendix Technology. 2020. *Where Thinkers become Makers*. Retrieved July 21, 2020 from https://www.mendix.com/

[35] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019) (CEUR Workshop Proceedings (CEUR-WS.org))*. Eindhoven, Netherlands. https://hal.archives-ouvertes.fr/hal-02363416

[36] Matt Trevathan. 2020. *Introducing Quantum Testing Framework*. Retrieved July 21, 2020 from https://basecamp.temenos.com/s/article-detail/a042K00001GieQ9QAJ/introducing-quantum-testing-framework

[37] Paul Vincent, Kimihiko Lijima, Mark Driver, Jason Wong, and Yefim Natis. 2019. *Magic Quadrant for Enterprise Low-Code Application Platforms*. Technical Report.