“This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884”

Project Number: 813884

Project Acronym: Lowcomote

Project title: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms

Concepts for Testing in Low-Code Engineering Repositories

Project GA: 813884

Project Acronym: Lowcomote

Project website: https://www.lowcomote.eu/

Project officer: Dora Horváth

Work Package: WP4

Deliverable number: D4.3

Production date: 30/11/2020

Contractual date of delivery: 30/11/2020

Actual date of delivery: 30/11/2020

Dissemination level: Public

Lead beneficiary: IMT Atlantique

Authors: Faezeh Khorram, Alessandro Colantoni, Luca Beradinelli, Jean-Marie Mottu

Contributors: The Lowcomote partners

Project Abstract

Low-code development platforms (LCPD) are software development platforms on the Cloud, provided through a Platform-as-a-Service model, which allow users to build completely operational applications by interacting through dynamic graphical user interfaces, visual diagrams and declarative languages. They address the need of non-programmers to develop personalised software, and focus on their domain expertise instead of implementation requirements.

Lowcomote will train a generation of experts that will upgrade the current trend of LCPDs to a new paradigm, Low-code Engineering Platforms (LCEPs). LCEPs will be open, allowing to integrate heterogeneous engineering tools, interoperable, allowing for cross-platform engineering, scalable, supporting very large engineering models and social networks of developers, smart, simplifying the development for citizen developers by machine learning and recommendation techniques. This will be achieved by injecting in LCDPs the theoretical and technical framework defined by recent research in Model Driven Engineering (MDE), augmented with Cloud Computing and Machine Learning techniques. This is possible today thanks to recent breakthroughs in scalability of MDE performed in the EC FP7 research project MONDO, led by Lowcomote partners.

The 48-month Lowcomote project will train the first European generation of skilled professionals in LCEPs. The 15 future scientists will benefit from an original training and research programme merging competencies and knowledge from 5 highly recognised academic institutions and 9 large and small industries of several domains. Co-supervision from both sectors is a promising process to facilitate agility of our future professionals between the academic and industrial world.

Report Executive Summary

The WP4 (Large-scale repository and services for low-code engineering) will produce a repository for low-code artefacts, able to store and enable the retrieval of heterogeneous modelling artifacts. Considering Low-Code Engineering (LCE), the repository will include services for data mining, continuous software engineering (CSE) [1] and quality assurance.

This first report (D4.3 - Month 23) among two (D4.6 - Month 41) presents the "Concepts for Testing in Low-Code Engineering Repositories" covering both the Tasks 4.4 and 4.5:

- Task 4.4: DevOps Support for Low-Code Engineering Platforms, considered by ESR9 (JKU)
- Task 4.5: Cloud-Based Testing Workbench for Low-Code Engineering, considered by ESR10 (IMT)

"The report will cover the requirements to write and configure low-code tests, the specification of the model transformations generating executable test, the consideration of the distribution of low-code artefacts (in particular software and data) on the Cloud. The deliverable will also define how low-code testing integrates in a process for DevOps in low-code engineering." [Grant Agreement]

Whereas this report covers the requirements for supporting testing approaches in low-code development platforms, it is complementary to the report D4.1 (M23) "Low-Code Engineering Repository Architecture Specification".

This report presents a preliminary model-driven approach to represent low-code testing processes and their integration with DevOps-enabled low-code engineering platforms. This work has already been validated in scientific publications [2, 3].

Contents

1 Introduction **5**

2 Low-Code Testing **6**
2.1 Features of Low-code Testing 6
2.1.1 Description of the features . . . 6
2.1.2 The Status of the Testing Component of Mendix LCDP 9
2.2 Challenges and Opportunities 11
2.2.1 The Role of Citizen Developer in Testing 11
2.2.2 The Need for High-level Test Automation 13
2.2.3 Cloud Testing 14

3 Modeling Low-Code Testing Processes and Platforms **16**
3.1 DevOpsML 16
3.1.1 Platform Specification 17
3.1.2 Process Specification 18
3.1.3 Integration Mechanism 19
3.2 Modeling Testing Requirements, Processes, and Low-Code Platforms in DevOpsML 19
3.2.1 Modeling Low-Code Testing Features. 21
3.2.2 Modeling Testing Capabilities of LCDPs. 22
3.2.3 Modeling Continuous Software Engineering Processes. . 24
3.2.4 Platform Requirements Modeling 25
3.2.5 Platform Capabilities Analysis 26
3.2.6 DevOps Process and DevOps Platform Weaving 27

4 Conclusion **28**

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 pre-built 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 [4, 5].

The main user of LCDP, called Citizen Developer, is a domain expert with no programming knowledge [4]. 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.

In all LCDPs, the Citizen developer has a major role in the application development process, but his lack of technical IT knowledge leads to the emergence of new requirements and challenges in the definition of the whole process as well as the concerns of each phase, including testing. There are several success stories of existing commercial LCDPs [6, 5], which demonstrate they overcome some challenges somehow. However, there is a vendor lock-in that confines the existing resources, which prevent both the interoperability and extensibility of their proposals.

In this deliverable, we focus on the identification of the requirements to realize the *testing phase of a DevOps process*, that is customizable for any LCDP. To this end, we initially synthesize the low-code testing concept, by specifying the requirements to write, configure, and execute low-code tests. Hereafter, we introduce DevOpsML as a framework for modeling DevOps processes and platforms. In this deliverable, the framework is particularly used to model the concerns and capabilities identified for low-code testing, their relationships with a commercial LCDP (Mendix, as an illustrative example), and their integration in a DevOps process example.

It is worth mentioning that, the final output of the work presented here, will be two services for Continuous Software Engineering (CSE) and Testing. They are introduced in deliverable 4.1, with their potential functionalities and components, as services integrated into the Low-Code Engineering Repository (LowCodER). In addition to the functionalities offered by a repository (CRUD operations in particular, as considered in deliverable D4.1), testing while following a DevOps process required features from a low-code platform. The main objective is to ultimately offer services that are reusable and/or customizable for a variety of LCDPs, as introduced in the next subsections. The requirements to reach such objective are detailed in this report. However, both of them are under development and their exact features cannot be strictly claimed. We first introduce our motivations for considering Low-code Testing while following a Devops process in the next subsections.

This deliverable is structured as follows:

- Chapter 2 identifies a set of requirements to support low-code testing in LCDPs, based on the low-code concerns in general and what exist in existing LCDPs.
- Chapter 3 introduces DevOpsML, a model-driven conceptual framework for modeling and combining arbitrary DevOps processes and platforms.
- Chapter 4 concludes the report.

2 Low-Code Testing

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 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. Although there are many commercial tools that are already used in the existing LCDPs for testing low-code systems (i.e., systems that are developed using an LCDP) [2], they are not open to access and some challenges remain unsolved, especially when considering the role of citizen developer in testing.

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 [7], 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; we use the term 'low-code testing' to denote the testing approaches considering the low-code context.

In this deliverable, we identified a set of requirements to support low-code testing in LCDPs, based on the low-code concerns in general [4]. We use Mendix as an illustrative LCDP to present which testing components are embedded in such a commercial LCDP (whereas in [2], we conducted an analysis of the testing components of five LCDPs, namely Mendix [8], Power Apps [9], Lightning [10], Temenos Quantum [11], and Outsystem [12]). It highlights what is offered by successful commercial platforms (i.e., their testing capabilities) and what is still missing. They have been selected based on the recent reports of Forrester [6] and Magic Quad Quadrant [5] 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.

Based on the result of our previous analysis, we came up with a set of requirements for writing, configuring, and executing low-code tests that are presented in the section 2.1.

Afterward, the existing challenges of the domain, and their associated difficulties and opportunities are described in a research-centric approach in section 2.2, by providing related work in the state-of-the-art.

2.1 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, along with the possible (may not complete) values for them. They are defined based on the low-code principles as well as the capabilities and deficiencies of the testing components of commercial LCDPs, presented in [2]. These features are indeed the decisions necessary to be made for building a low-code testing component, which is adaptable to the required testing capabilities of a DevOps process using an LCDP.

As a case study, we chose Mendix LCDP to evaluate its testing facilities based on our proposed feature list. Moreover, we will model its provided and required concerns and capabilities in section 3.2.2.

2.1.1 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.

2.1.1.1 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

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.

open through import/export techniques to reuse testing artifacts of other sources, or completely open via web-technologies.

2.1.1.2 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 [2], 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 features 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 features 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 feature 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:

- providing the possibility of reusing test data/test cases of other sources; when the same language, for test data/test case definition, is used in the low-code testing component and in the external source.
- offering reusable test cases from a pre-defined repository; when the low-code testing component offers a set of reusable test cases (stored in a repository), that are previously defined for general objectives
- 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; when the low-code testing component is customized for a specific LCDP, and supports definition of reusable test cases (stored in a repository) for later reuse in that specific LCDP
- 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; when the low-code testing component is generic and therefore applicable in various LCDPs, and also supports definition of reusable test cases that could be stored in a repository to be reused later on

2.1.1.3 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)[1], and programming languages (e. g., Java).

2.1.1.4 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.

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.

2.1.1.5 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 presenting 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.

2.1.2 The Status of the Testing Component of Mendix LCDP

To be consistent all over the report, Mendix only is used as a case study (whereas three other LCDPs have been considered in [2]). Nevertheless, the goal is not to be exhaustive but to illustrate that even a major LCDP does not fulfil Lowcomote project's objectives.

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 [8].

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 [13].

For supporting other kinds of testing, Mendix recommends the use of commercial tools such as SoapUI[2] for automated integration and API testing, Selenium IDE[3] for browser-based UI and acceptance testing, and TestNG [14] for scripting automated tests in Java language [13].

Moreover, there are three quality add-ons provided by Mendix which are not testing tools, but their usage improves the quality of the application:

- *Application Test Suite (ATS)*: ATS is a set of tools built on top of Selenium [15] for embedding test into application lifecycle.
- *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 [16].The features are analyzability, modifiability, testability, modularity, and reusability;

[1]http://www.ttcn-3.org/
[2]https://smartbear.com/product/ready-api/soapui/overview/
[3]https://www.selenium.dev/selenium-ide/

- *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 [17].

To demonstrate how our proposed feature list can be applied for the evaluation of low-code testing components, we use it to organize our analysis result on the Mendix testing component.

1. Mendix proposes a new Low-Code Testing Framework (LCTF) but with limited capabilities. Consequently, it provides integration with third-party testing tools to have reasonable coverage of all testing activities.
2. The LCTF of Mendix is designed only for unit testing, and the other testing scales are supported by its integrated third-party testing tools.
3. Functionality and Performance are the features that are continuously tested in Mendix.
4. Mendix uses web technologies to integrate with third-party testing tools, and it also provides test data import/export mechanism.
5. Technical testers and/or developers are in charge of performing tests using the third-party tools integrated with Mendix. The citizen developer is only involved in the testing activities supported by Mendix LCTF (i.e., unit testing), and also Automated UI testing using Selenium WebDriver.
6. Collaboration is considered in Mendix, but mainly between technical developers and testers. For collaboration with non-technical developers, Mendix offers an easy to use feedback mechanism.
7. Among different test design techniques, the LCTF of Mendix follows the MBT and graphical modeling approach for designing unit tests based on the DSL it uses for microflow modeling (i.e., BPMN language). Moreover, Record and Replay technique is supported for automated UI testing, through integration with Selenium IDE, and Keyword-driven and DDT techniques are used in the integrated third-party testing tools.
8. Among the various artifacts, Mendix LCTF uses i) system requirements to explicitly map them to the test cases, ii) UI pages for performing UI tests through Selenium recording tool, since UI test specifications are captured based on user interaction with UI pages, and iii) system models, including microflow diagrams and domain models, for designing unit tests. Mendix LCTF also produces unit test models (i.e., BPMN models) since it uses graphical modeling for designing unit tests.
9. Mendix LCTF offers reusable test case templates, importing test data from external files, and definition of reusable test cases to be used in the testing of other applications built in the Mendix platform.
10. A medium level of automation for test generation is provided in Mendix, but mostly by its integrated third-party testing tools.
11. The final executable test cases are generated in java (Mendix LCTF uses jUnit library).
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 Mendix, by leveraging Selenium Grid.
14. Mendix uses Selenium Server as its test execution tool.
15. Provider cloud and on-premises are the supported test execution platforms in Mendix.
16. Monitoring and visual and textual reporting are provided in Mendix. It also analyzes the execution traces and offers notes for improvement.

To sum up, the results of our evaluation on the testing components of commercial LCDPs that is presented in [2], revealed that there are specific features for low-code testing that should always be considered and cannot be neglected in the testing component of any LCDP. They are the role of citizen developer, 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 academia, and proposing opportunities based on them for future work in this area.

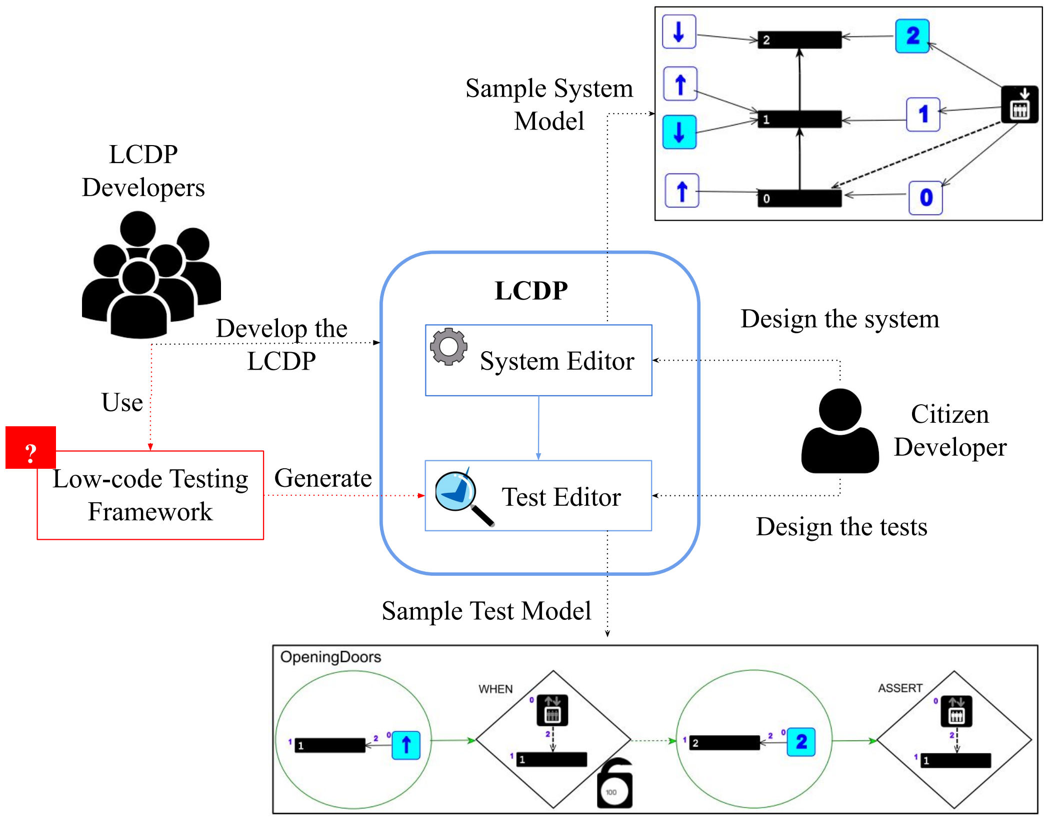


Figure 1: An overview of the main challenges in low-code testing (The model instances are derived from [18])

2.2 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 problem 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 a 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.

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.

2.2.1 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.

Regarding the main objective of LCDPs, i. e., providing system development facilities for domain experts, Domain-Specific Languages (DSLs) are the underlying theory in the LCDP development; 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 [19]. 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 [20].

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 at the top of 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 [18] 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 the 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 [21]. 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.

Once again, it is not appropriate to propose a solution dedicated to one LCDP and its DSL. Lowcomote Project promotes Low-code Engineering to enable citizen developers in defining test cases, by using a generic testing language which is fairly easy to use for non-technical developers. The language should support automatic test configuration, generation, and execution, since these are the very technical activities of the testing process and consequently have to be relaxed for low-code testing. Additionally, as one of the main objectives of Low-Code Engineering Repositories is offering services that could be reused by various LCDPs, the language being independent from any platform should be compatible with any low-code DSLs thanks to transformations and generations. However, there exist no such testing language.

2.2.1.1 Previous Attempts

The first 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 [22, 23], 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 [18], 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 [24].

TDL is a potential candidate to be the testing language of Lowcomote Testing Framework: it is standard, specifically defined for non-technical testers, offers a high-level of abstraction for test description, and is platform-independent [24]. However, the main shortcoming of TDL is in its execution. TDL is not executable by itself, and for running test cases written in TDL, they have to be first transformed to TTCN-3 code, and then the technical developers have to implement adapters and codecs to make test execution feasible on the intended system under test. Therefore, a high-level of technical aspects should be managed manually, and TDL is currently deficient for low-code testing (regarding the low-code testing features presented in Table 1)

2.2.1.2 Opportunities

Among numerous 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.

Following the difficulties mentioned for direct use of TDL for low-code testing, a good opportunity is to make it executable without relying on TTCN-3 language. We are currently working on the implementation of an execution engine for TDL which will be capable of performing automatic test generation, configuration, and execution.

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, or using a very simple testing language, they need to be assisted on how to correctly design the test models.

2.2.2 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 [25].

Briefly, the results of this query along with the information gathered in [2] 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. MBT can be considered following two different directions:

- **SUT Modeling**: Modeling the System Under Test (SUT) using a Domain-Specific Modeling Language (DSML), and then auto-transforming these system models into the test cases, test scripts, and test data, using transformation engines.
- **Test Modeling**: Modeling the test cases and test data using a test-specific DSML, and then automatically configuring and executing them on the SUT through the DSML execution engine.

It is evident that the engines are the pivotal elements in both approaches, 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. Considering the SUT Modeling approach, 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 [26]. For the Test Modeling approach, TDL and TTCN-3 are the only test-specific DSMLs. However, the former is not executable at all, and the latter requires a high-level of technical and manual effort to become executable.

2.2.2.1 Previous Attempts

According to the classification of MBT approaches into two categories, namely SUT Modeling and Test Modeling, we identify the previous attempts for each one of them.

SUT Modeling: 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 [26]. 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.

Test Modeling: To the best of our knowledge, there is no previous effort on making TDL executable, and for making TTCN-3 less technical, TDL is in fact introduced for this purpose, but only for filling the technical gap in the test design. Therefore, TTCN-3 test generation, configuration, and execution is still a technical and demanding task [24].

2.2.2.2 Opportunities

MBT as SUT Modeling: 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.

MBT as Test Modeling: Implementation of an execution engine for TDL could be seen as the main opportunity, since TDL's strengths fit well with the low-code test design features of table 1.

2.2.3 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 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 [27].

In 2019, Bertolino et al. performed a systematic review of the cloud testing area [27]. 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 [27], 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 [27], all three aspects of cloud testing have to be provided by LCDPs, especially in those of scalable.

2.2.3.1 Previous Attempts

Besides the existing challenges and issues described in [27] for the cloud testing in general, the specific features of low-code introduces new ones. As we described in section 2.2.2, MBT (in its both directions) 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. As far as we know, there is no related work for considering cloud in MBT as Test Modeling, and all the following presented information are the previous attempts in MBT as SUT Modeling.

In MBT, given an abstract picture of the SUT, it is possible to generate many test cases to be executed on the cloud [27]. 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

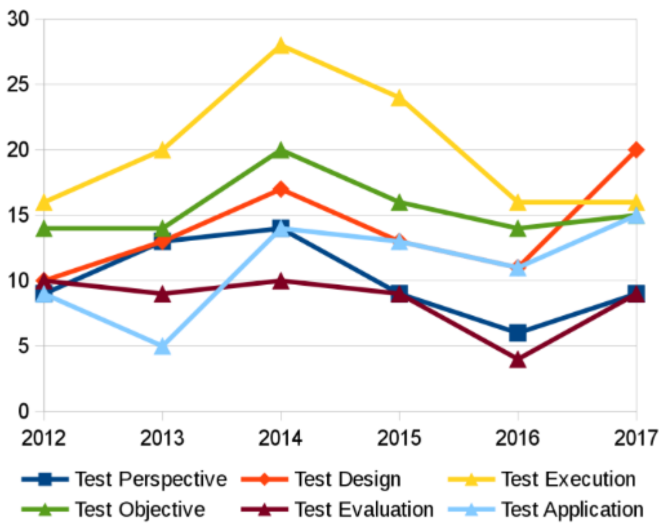


Figure 2: The trends of the areas in cloud testing by year (Taken from [27])

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 [28, 29, 30, 31, 32, 33].

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.

2.2.3.2 Opportunities

We identify 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, which the best potential one is TDL. 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.

3 Modeling Low-Code Testing Processes and Platforms

This Section introduces DevOpsML [3], a model-driven conceptual framework for modeling and combining arbitrary DevOps processes and platforms. Tools along with their interfaces and capabilities are the building blocks of DevOps platform configurations, which can be mapped to software engineering processes of arbitrary complexity.

In this deliverable, we show how the DevOpsML framework can be used to model and combine testing processes and low-code platform specifications to support specific testing features as those outlined in Section 2 (see Table 1).

The rest of the section is organised as follows. Section 3.1 introduces DevOpsML [3], its modeling capabilities for DevOps processes and platforms, and its weaving mechanism for combining process and platform models. Section 3.2 shows DevOpsML in action. In particular, we show how to model and combine i) generic testing features as reusable library of DevOpsML capabilities and concerns, ii) DevOps pipeline including testing steps, iii) testing process requirements for LCDP, and iv) LCDPs as a suitable combination of tools whose provided capabilities satisfy the given testing requirements.

3.1 DevOpsML

Over the last decade, DevOps methods and tools have been successfully implemented and adopted by companies to boost automation and efficiency of the engineering process. The term DevOps was coined in 2009 [34] and became popular among companies and practitioners [35] and, subsequently, among researchers and academia. Jabbari et al. [36] define DevOps as *"[...] a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices."*.

The momentum on DevOps resulted in a flourishing of technological solutions to meet the huge market demands [37]. The side effects of such a rapid evolution was a scattered landscape of technological solutions offering a variety of tools [38] for supporting activities of *continuous-software engineering* (continuous software engineering) [1] processes.

Figure 3 gives a bird's eye view of the problem at hand. There is no "one size fits all" DevOps process that is capable to cope with all the specific goals, strategies, and requirements. Different DevOps process variants exist (e.g., DevSecOps [39] or AIOps [40] to mention just a few). The process variability is reflected on DevOps platforms too, which may aggregate different engineering services (e.g., security mechanisms and AI-augmented services) depending on process needs.

Consequently, the choice of DevOps platforms for specific engineering processes is still an open challenge. In [34], Bordelau et al. investigated and elicited sets of requirements for DevOps frameworks. Among them, they consider also the need for an adequate support for modeling of DevOps engineering *processes*, of the *product* resulting from the process, i.e., the software system, as well as requirements of *resources* (e.g., tools) involved in the accomplishment of development and operations phases. Furthermore, it has to be mentioned that the same DevOps process and platform can be more or less adequate based on different skills of the involved stakeholders, nowadays possibly ranging from skilled engineers to domain experts with no ICT background at all.

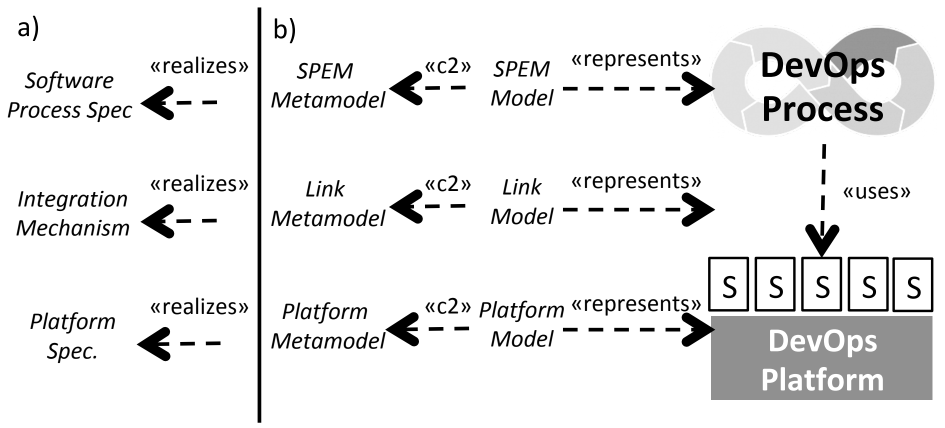


Figure 3: The DevOpsML framework elements (a) and their implementations (b).

In order to tackle this shortcoming, we introduced in [3] DevOpsML, a conceptual framework for modeling and configuring DevOps engineering processes and platforms. With DevOpsML, we approach the problem of DevOps process and platform integration from two directions. First, in a bottom-up approach, existing DevOps platforms can be studied and their characteristics made explicit in so-called platform models. Second, in a top-down approach, we propose the usage of process modeling languages [41] to explain the DevOps processes. The glue between the two directions is a linking language, which explains how the different services offered by the platforms are used by the processes.

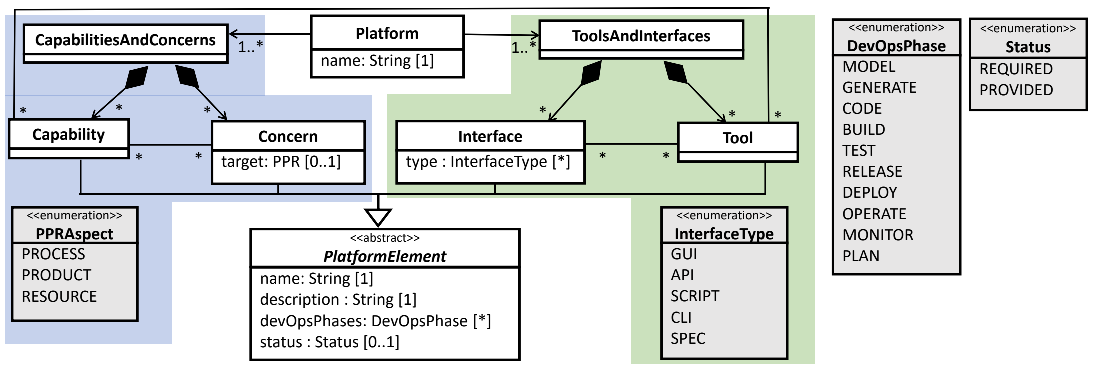


Figure 4: Platform metamodel.

In Section 3.2, we show how the DevOpsML conceptual framework can be adopted to show the integration of low-code testing phases in user-defined DevOps engineering processes supported by configurable low-code platforms. In particular, we will show how DevOpsML can be adopted to model i) the low-code testing features collected in Table 1 (as reusable library of testing capabilities and concerns), ii) a low-code platform offering testing capabilities, and iii) a DevOps process including testing phases requiring such testing capabilities. The resulting process and platform models can be finally integrated via model weaving [42].

The rest of this section details the DevOpsML framework elements depicted in Figure 3 and present some guidelines on how to use DevOpsML.

DevOpsML consists of three elements, namely *software process specification*, *platform specification*, and *integration mechanism*. In our prototypical implementation, available at [43], we implement them following a typical model-driven approach. We separately introduce the DevOpsML framework elements in the following subsections.

3.1.1 Platform Specification

According to the given DevOps definition [36], we expect that a platform configuration is capable to support development and operational activities adopting and realizing MDE principle and practices [42].

In DevOpsML, a platform is a combination of tools, each one providing (or requiring) a set of capabilities addressing engineering concerns of interest, supported by combining tools via explicit interfaces. A platform is expected to satisfy and be able to satisfy the needs of an arbitrary engineering process.

Following typical MDE practices, we provide a *platform metamodel* to create platform models representing a combination of tools, interfaces, capabilities, and concerns. The metamodel with its concepts and relationships is shown in Figure 4 and implemented in Ecore [43]. The following paragraphs detail the content of the platform metamodel.

Tools and Interfaces. This package introduces metaclasses for representing tools and interfaces.

In its broadest meaning, a tool is a resource that helps in accomplishing a work unit of an engineering process. It provides or requires interfaces.

An interface represents the boundary of tools, and consequently, of platforms as a whole, through which interactions among tools and platforms take place, according to their required and provided capabilities. Interfaces play the roles of connectors [44] among platform elements, directly connecting multiple tools and, through them, their capabilities. We consider a predefined but extensible set of interface types, i.e., graphical user interfaces (GUI), application programming interfaces (API), script, command line interfaces (CLI), or more generic specifications (SPEC).

Capabilities and Concerns. This package introduces the concepts of *capability* and *concern*.

For *capability*, we intend a facility enabled by a particular platform configuration for performing a specified activity that will be specified in a separated process model (see Section 3.1.2).

A *concern* is *"a stakeholder's interest that pertains to the development of an application, its operation or any other matters that are critical or otherwise important"* [45].

A platform configuration is expected to offer capabilities to address concerns related to (*i*) the engineering process, (*ii*) the system under study, i.e., the *product* of the engineering process, and (*iii*) of the *resources* (both human and technological ones) required for the correct and convenient process execution and delivery of the engineered product. We introduce the *PPRAspect* enumeration to distinguish among process, product, and resource concerns [46].

Common concepts. The platform metamodel also includes some shared concepts: platform element, status, and DevOps phase.

A platform element is an abstract concept that groups common properties of capabilities, concerns, tools, and interfaces. For all these platform elements, a name and a textual description are mandatory and, together, correspond to the minimal wealth of knowledge required to model platform elements.

Since we intend to model DevOps platforms, we expect that its elements will support typical DevOps process phases (code, build, test, release, deploy, operate, monitor, and plan), which are given as literals of the DevOps phase enumeration. In addition, we aim at applying MDE principles and practices and for this reason, we include model and generate phases, which are recurrent activities in model-driven engineering processes. Each platform element can be associated with many DevOps phases.

It is worth nothing that the given DevOps phase enumeration is not exhaustive and it can be extended, if needed. However, we expect detailed engineering process information to be modeled in separated process models (see Section 3.1.2). Finally, provided or required status can be set for any platform element, providing rationales for composing the platform like technology-wise matching of required and provided tools' interfaces or higher-level evaluation of platforms' capabilities against engineering process or product-related concerns.

3.1.2 Process Specification

Process management is a core concern for software engineering since decades [47, 48] and regards the specification and execution of organizational behaviours, where working units at different levels of granularity are combined in a workflow. Stakeholders with defined roles collaborate to perform the process. Engineering artifacts are produced and manipulated throughout the process.

In DevOpsML, a platform is meant to support model-driven continuous software engineering processes, where activities are expected to manipulate and share MDE artifacts [49], aiming at the highest degree of automation.

In DevOpsML, we assume that a process (model) provides the rationales to choose the elements of a DevOps platform (model), defined as described in Section 3.1.1. For the sake of process specification, DevOpsML needs a software process modeling language (SPML). In [41], a quality model for SPMLs is given.

Table 2: Process modeling capabilities [50] and their support in SPEM

Process modeling capability	**SPEM (MC:Method Content Package, PM:Process with Methods Package)**
A process type is defined by the composition of one or more task types. Each process comprises one or more tasks (P1, P11).	PM:Activity; PM:Task Use
Each task type is created by an actor. An actor may have more than one actor type. An actor that performs a task must be authorized for that task. Actor types may specialize other actor types (P4, P15,P17, P18).	MC:Role Definition; PM:Composite Role; PM:Role Use; PM:Team Profile
Tasks are associated with artifacts used and produced, along with performing actors. For each task type one may stipulate the artifact types which are used and produced (P7, P13, P14)	MC:Task Definition; MC:Work Product Definition; MC:Role Definition; MC:Default Task Definition Parameter; PM:Work Product Use;

For our first prototypical implementation of DevOpsML [43], we choose the Software and Systems Process Engineering Metamodel (SPEM) [51]. SPEM satisfies a minimal set of modeling capabilities that we require for a DevOpsML proof of concept phase. Table 2 provides a list of common capabilities of process modeling languages described in [50] for the sake of a process modeling challenge[4] and maps them to the corresponding concepts defined in SPEM [51]. The complete mapping is available in [43].

Second, we decide to give higher importance to the descriptive capability of SPEM [47] for documentation purposes rather than executability, in which case BPMN or UML Activities (via Foundational UML (fUML) [52]) are more appropriate solutions than SPEM. Moreover, SPEM supports the specification of new processes by separating the definition of reusable process model elements (see Method Content language package in [51]) and their actual uses in processes (see the Process with Methods language package in [51]). In Table 2, we reported the owning package of each mapped concept that we use later in Sections 3.2.4 and 3.2.6 to show DevOpsML in action.

However, it is worth noting that the choice of a particular SPML for process specification is a *variation point of DevOpsML*. Indeed, different SPML can be chosen from existing ones [41] or new ones can be created following software language and model-driven engineering practices [53, 54] to cope with arbitrary SPML's requirements like process modeling capabilities [50] or usability by (non-)technical users. For example, when considering LCDPs [2, 55] as target platforms in DevOpsML, we may expect to reuse a built-in process modeling language and graphical process editors (e.g., microflow by Mendix [8]).

[4]The table report in parenthesis the original IDs (Px) of the process modeling capabilities.

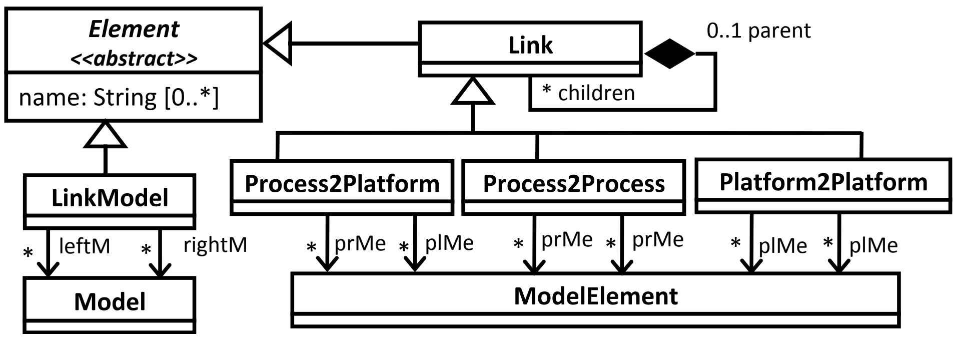


Figure 5: The Linking metamodel.

3.1.3 Integration Mechanism

DevOpsML is a model-driven framework and, as such, different model integration mechanisms are good candidates to provide a model integration capability, such as model weaving and model transformation [42].

For DevOpsML, we choose the model weaving mechanism to link elements from platform and process models. Epsilon Modelink [56] for establishing and editing references between platform and process models.

Figure 5 shows a *linking metamodel* for this purpose. A *link model* contains a collection of hierarchical *links*. Since we do not specify any direction for links, we consider them bidirectional by construction. Each link is weaving sets of model elements belonging to process or platform models[5].

Based on the nature of the woven models, three different types of links can be distinguished: *platform to process*, *platform to platform*, and *process to process*:

- **Process to Platform**: process work units, at any granularity level (e.g., SPEM activities, tasks, steps) can be mapped to platform elements (tools, interfaces, capabilities, concerns).
- **Platform to Platform**: elements from two or more platform models can be linked for user-defined rationales. For example, based on values set for status property, potential (mis)matches can be modeled and used for evaluations (e.g., classification of different platform configurations against concerns' coverage criteria).
- **Process to Process**: elements from two or more process models can be linked for user-defined rationales. In particular, since DevOpsML does not prescribe the use of any process modeling language, these links can be used to relate process models defined with different process modeling languages[6].

3.2 Modeling Testing Requirements, Processes, and Low-Code Platforms in DevOpsML

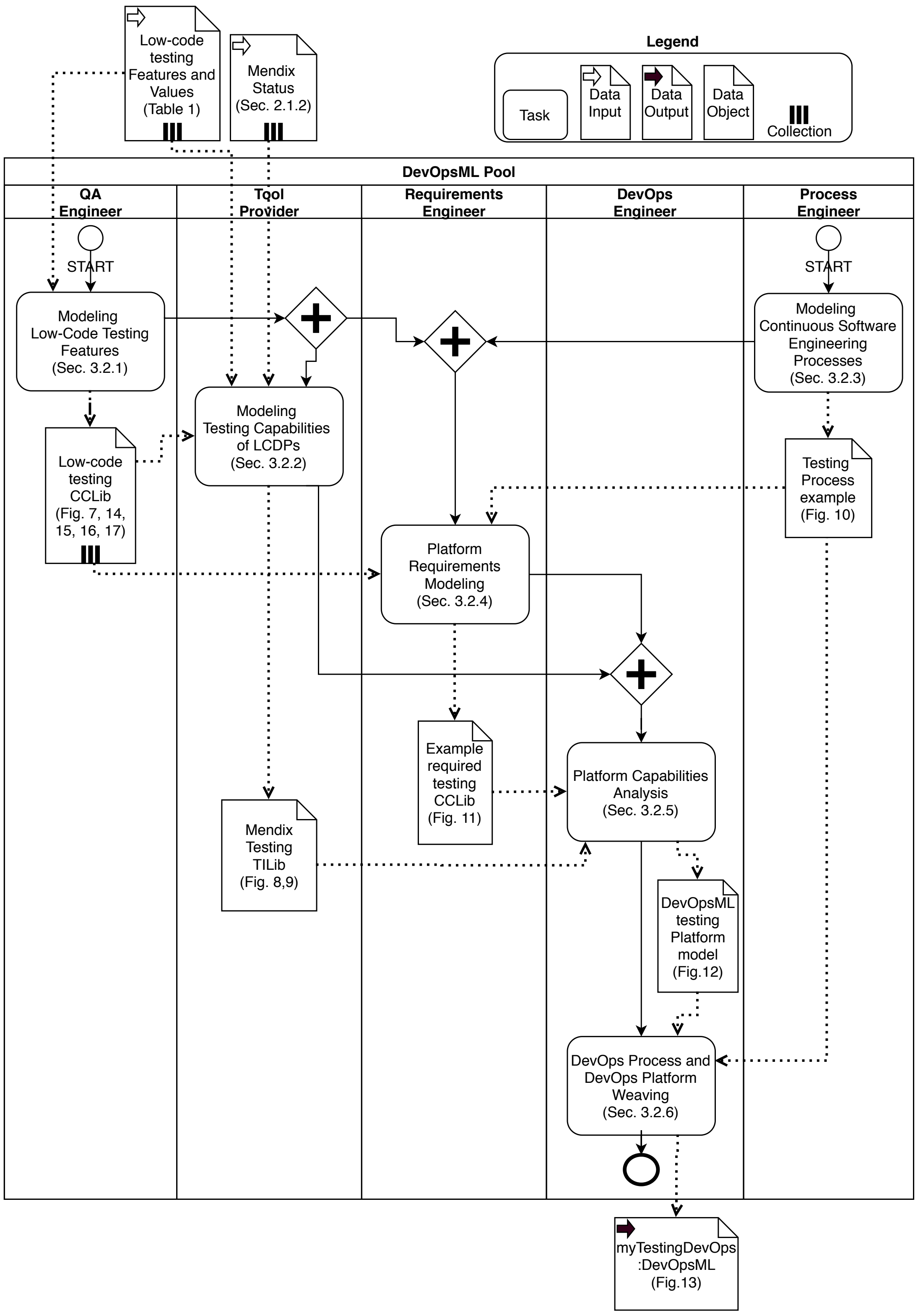
In this section, we adopt the DevOpsML conceptual framework [3] to model the testing features for low-code platforms presented in [2] and reported in Table 1 in Section 2, together with the assessment of support provided by Mendix, one of the five analyzed commercial LCDPs (Section 2.1.2).

In Figure 6, a BPMN-like process diagram is shown depicting the steps and related input/output artifacts, according to the DevOpsML guidelines given in [3]. The DevOpsML-related steps and involved artifacts will be detailed in the following subsections. In particular:

- Section 3.2.1 presents the specification of low-code testing features outlined in Table 1 using the DevOpsML conceptual framework [3]. Testing features are represented as capabilities and concerns (CC) and collected in libraries (CC libraries) conforming to the DevOpsML platform metamodel. This step is expected to be performed by a Quality Assurance Engineer (QA Engineer), whom primary concern is maximizing quality of final product.
- Section 3.2.2 discusses how to model the testing features supported by a LCDP in DevOpsML. We choose to model testing features of the Mendix LCDP [8], in accordance with the information provided in [2] and reported in Section 2.1.2. The Mendix LCDP and integrated third-party tools are collected in tools and interfaces libraries (TI libraries)T conforming to the DevOpsML platform metamodel (Figure 4). This step is expected to be performed by tool providers or tool experts in general, which know tools' capabilities and their available APIs.
- Section 3.2.3 discusses the modeling of DevOps continuous software engineering processes (CSE [1] processes) addressing testing concerns, i.e., including engineering steps (e.g., unit testing) requiring specific testing features. As a result, a continuous process model is obtained, including typical

[5]We keep the Linking metamodel simple on purpose. We do not show here the specialisation of ModelElement metaclass and constraints to distinguish intra/inter-model links.

[6]Technical limitations to these scenarios apply if the integration mechanism does not support heterogeneous modeling spaces [49].



Low-code testing Features and Values (Table 1)
Mendix Status (Sec. 2.1.2)
Legend
Task
Data Input
Data Output
Data Object
Collection
DevOpsML Pool
QA Engineer
Tool Provider
Requirements Engineer
DevOps Engineer
Process Engineer
START
START
Modeling Low-Code Testing Features (Sec. 3.2.1)
Modeling Continuous Software Engineering Processes (Sec. 3.2.3)
Modeling Testing Capabilities of LCDPs (Sec. 3.2.2)
Low-code testing CCLib (Fig. 7, 14, 15, 16, 17)
Testing Process example (Fig. 10)
Platform Requirements Modeling (Sec. 3.2.4)
Example required testing CCLib (Fig. 11)
Platform Capabilities Analysis (Sec. 3.2.5)
Mendix Testing TILib (Fig. 8,9)
DevOpsML testing Platform model (Fig.12)
DevOps Process and DevOps Platform Weaving (Sec. 3.2.6)
myTestingDevOps :DevOpsML (Fig.13)

Figure 6: BPMN diagram of the approach based on DevOpsML [3].

DevOps activities (e.g., continuous integration). A process model is expected to conform to a given process modeling language. In [3], we chose SPEM [51]. This step is expected to be performed by a process engineer, who is responsible for designing, implementing, controlling and optimizing the CSE process.

- Section 3.2.4 model the platform requirements in order to support given testing concerns during a CSE process. A collection of required testing capabilities specific to testing concerns are collected for CC library. This step is expected to be performed by requirement engineers.
- Section 3.2.5 discusses the specification of platform requirements, based on the set of testing capabilities defined in Section 3.2.1. As a result, candidate DevOpsML platform models can be created by integrating elements from CC and TI libraries matching testing requirements. This step is expected to be performed by DevOps engineers, who are able to use the resulting DevOpsML model to guide

the preparation of a working model-driven DevOps engineering environment.

- Finally, Section 3.2.6 discusses the combination of DevOps process with platform models representing LCDPs and integrated third-party tools. Tools and their capabilities are mapped to typical DevOps phases (see Figure 4). This step is expected to be performed by a DevOps engineer,

The engineering services required to support the approach modeled by the BPMN in Figure 6 will be provided by a Continuous Software Engineering (CSE) service as shown in the overall architecture of the Low Code Repository (see Deliverable D4.1).

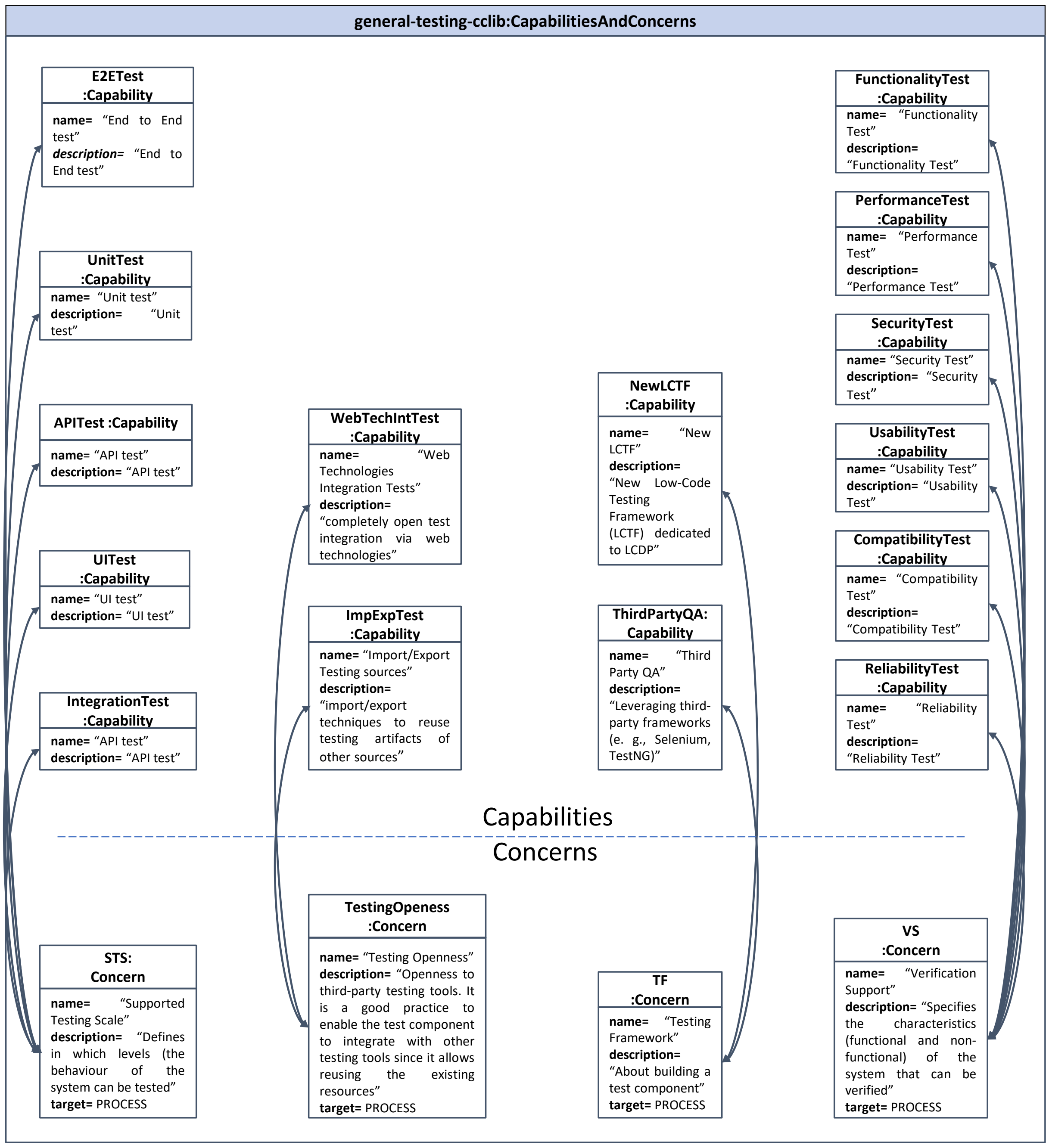

general-testing-cclib:CapabilitiesAndConcerns
E2ETest :Capability
name= "End to End test"
description= "End to End test"
UnitTest :Capability
name= "Unit test"
description= "Unit test"
APITest :Capability
name= "API test"
description= "API test"
UITest :Capability
name= "UI test"
description= "UI test"
IntegrationTest :Capability
name= "API test"
description= "API test"
WebTechIntTest :Capability
name= "Web Technologies Integration Tests"
description= "completely open test integration via web technologies"
ImpExpTest :Capability
name= "Import/Export Testing sources"
description= "import/export techniques to reuse testing artifacts of other sources"
NewLCTF :Capability
name= "New LCTF"
description= "New Low-Code Testing Framework (LCTF) dedicated to LCDP"
ThirdPartyQA: Capability
name= "Third Party QA"
description= "Leveraging third-party frameworks (e. g., Selenium, TestNG)"
FunctionalityTest :Capability
name= "Functionality Test"
description= "Functionality Test"
PerformanceTest :Capability
name= "Performance Test"
description= "Performance Test"
SecurityTest :Capability
name= "Security Test"
description= "Security Test"
UsabilityTest :Capability
name= "Usability Test"
description= "Usability Test"
CompatibilityTest :Capability
name= "Compatibility Test"
description= "Compatibility Test"
ReliabilityTest :Capability
name= "Reliability Test"
description= "Reliability Test"
Capabilities
Concerns
STS: Concern
name= "Supported Testing Scale"
description= "Defines in which levels (the behaviour of the system can be tested"
target= PROCESS
TestingOpeness :Concern
name= "Testing Openness"
description= "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"
target= PROCESS
TF :Concern
name= "Testing Framework"
description= "About building a test component"
target= PROCESS
VS :Concern
name= "Verification Support"
description= "Specifies the characteristics (functional and non-functional) of the system that can be verified"
target= PROCESS

Figure 7: General Capabilities and Concerns libraries. Modeled on General features of Table 1

3.2.1 Modeling Low-Code Testing Features.

One of the very first steps[7] consists in creating libraries of testing capabilities and concerns (CC Lib). The CC library can be created from the information collected in Table 1 on testing. In Table 1 five categories of testing features for LCDP have been identified (General, Test Design, Test Generation, Test Execution, and Test Evaluation), together with a collection of concrete testing features per category (see Table 1).

The following modeling rules have been applied to (manually) create a CC library from the textual descriptions given in Table 1:

[7]The other one is the process modeling, which can be done concurrently, according to the DevOpsML guideline [3]

- For each Category in Table 1, a separated CC library model artifact has to be created. Since we are considering testing capabilities and concerns, we add the suffix `testing-cclib` to the resulting library. For example, a General-Testing CC library model is created containing an instance of the *CapabilitiesAndConcerns* metaclass. The resulting artifact is a platform model of reusable platform elements [3], i.e., a library, conforming to the platform metamodel in Figure 4.
- For each Feature in Table 1, a new instance of the *Concern* metaclass is created with the same name of feature and an optional description. The concern is related to the support of testing process in low-code platforms and for this reason, the value *process* from the PPRAspect enumeration [46] is chosen for the enumerated property *target*.
- For Possible Values in Table 1, instances of the *Capability* metaclass are created with the same name and an optional textual description

The informal mapping rules introduced above are applied during the *Create CC Libraries* step in the BPMN diagram in Figure 6 A separated CC Lib is created for each low-code testing feature category in Table 1. The resulting CC library for the General testing feature category is shown in Figure 7 using an object diagram-like notation. The CC libraries of the remaining testing feature categories are shown in the Appendix.

It is worth noting we explicitly keep undefined the properties *status* and *devOpsPhases* of any capabilities and concerns (available as specialisation of the abstract metaclass *PlatformElement*, see Figure 4) on purpose.

The *status* property is an optional, single-valued property (multiplicity [0..1], see Figure 4), whose allowed values are determined by the literals of the *Status* enumeration. The *status* will be determined later in the BPMN in Figure 6 by requirements engineers and tool providers to model testing features in requirement specifications (see Section 3.2.4) and tool descriptions (see Section 3.2.2), respectively [3].

Similarly, the *devOpsPhases* is an optional, multi-valued property (multiplicity [*], see Figure 4), whose allowed values are determined by the literals of the *DevOpsPhase* enumeration. It is used to bound a particular platform element (i.e., a capability, a concern, a tool, an interface) to one or more typical DevOps phases. Keeping *devOpsPhases* unspecified in the resulting CC library allows to preserve the genericity of the library w.r.t. software processes and tools, which are not yet taken into consideration at this stage of the approach (see Figure 6).

3.2.2 Modeling Testing Capabilities of LCDPs.

In [2], the support to testing features (see Table 1) of five commercial LCDPs is discussed and an evaluation is given to the reader. According to the goal of this deliverable (*"The deliverable will also define how low-code testing integrates in a process for DevOps in low-code engineering"*), those LCPDs represent candidate tools to be integrated in a DevOps platform (see Figure 3) whose services can support a *continuous software process (CSE)* [1].

In this regard, DevOpsML [3] allows the modeling of tools to be integrated in such DevOps platforms, together with provided/required interfaces and provided/required capabilities to address concerns of interests, which are, in our case low-code testing features [2].

Therefore, according to the available documentation sources (e.g., publications [55, 2]) or LCDPs user guides [8, 9, 10, 11, 12]) a model-driven artifact can be (manually) created that conforms to the DevOpsML platform metamodel shown in Figure 4.

Figure 8 shows a DevOpsML Platform Model for the Mendix platform. A similar model can be created for each LCPD and collected in a Tools and interfaces (TI) library to be reused across different configurations of DevOps platforms.

In particular, the following modeling rules have been applied in order to create a TI library for LCDPs as instance of DevOpsML platform metamodel in Figure 4:

- A *ToolAndInterface* element is created as the top-level element of a TI library. In Figure 8 a *mendix-toolset* TI library is created.
- The TI library is populated with Tools, with a name and an optional description. Figure 8 includes the Mendix, Selenium, SoapUI and TestNG Tool elements, which together represent the low-code testing framework (LCTF) provided by Mendix (see Section 2.1.2).
- If available, *Interface* elements are created to represent available tool interfaces (e.g., APIs, graphical user interfaces, command line interfaces, according to the *InterfaceType* enumeration). In the given example, we skipped the modeling of Mendix interfaces for the sake of readability.
- For each Tool in the TI library, the CC libraries of testing features introduced in Section 3.2.1 are imported to model tool-specific provided and required capabilities and concerns. Figure 8 shows the CC libraries imported for Mendix, namely *general-testing-cclib-mendix*, *test-design-testing-cclib-mendix*, *test-generation-testing-cclib-mendix*, *test-execution-testing-cclib-mendix*, *test-evaluation-testing-cclib-mendix*. The same step has to be repeated for Selenium, SoapUI and TestNG tools.
- If a testing feature is supported by a given Tool, the corresponding *Capability* is updated by assigning the literal `provided` to the *status* property. Similarly, if a testing feature is explicitly stated as provided by integration with third-party tools, the corresponding *Capability* is updated by assigning the literal

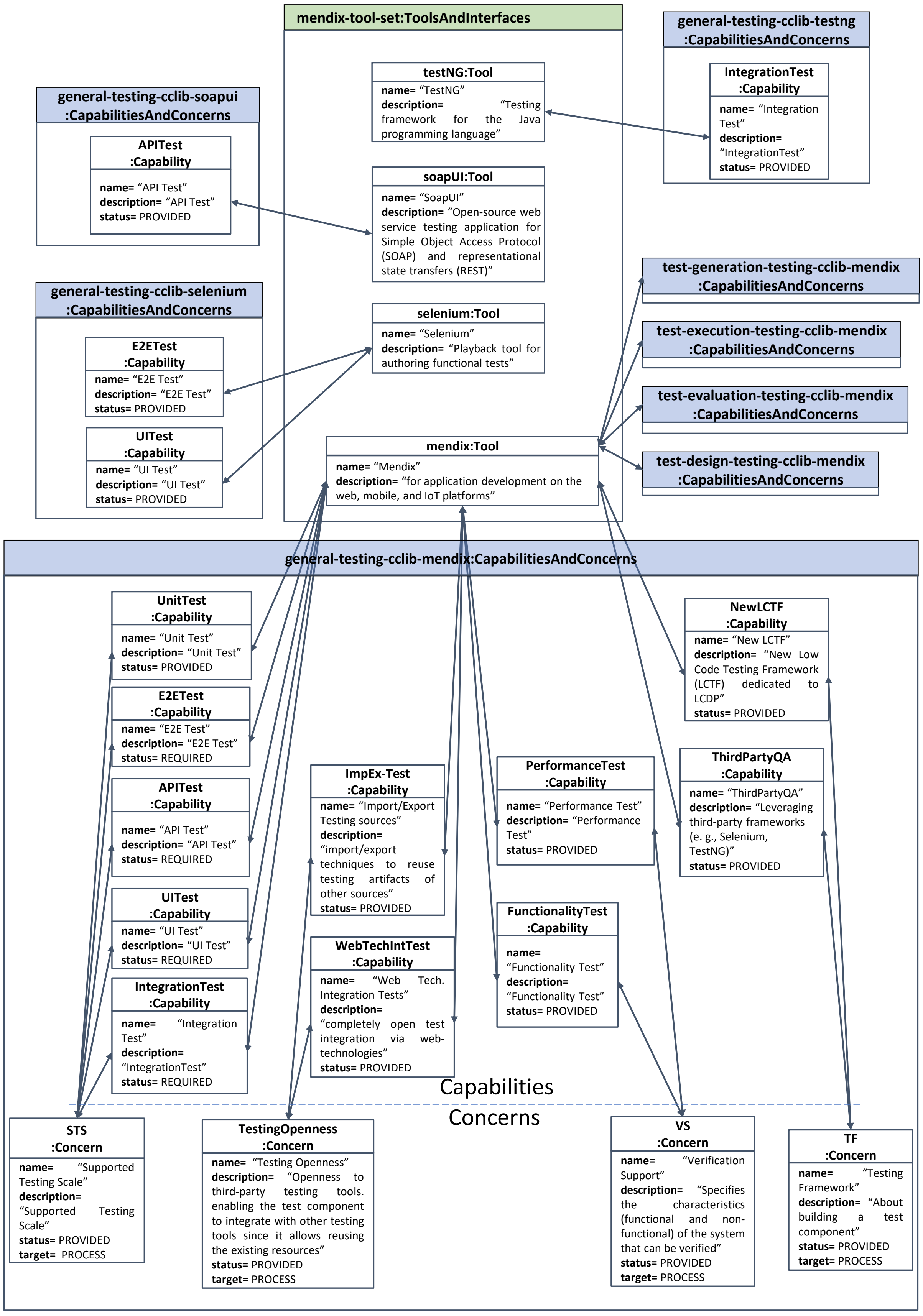
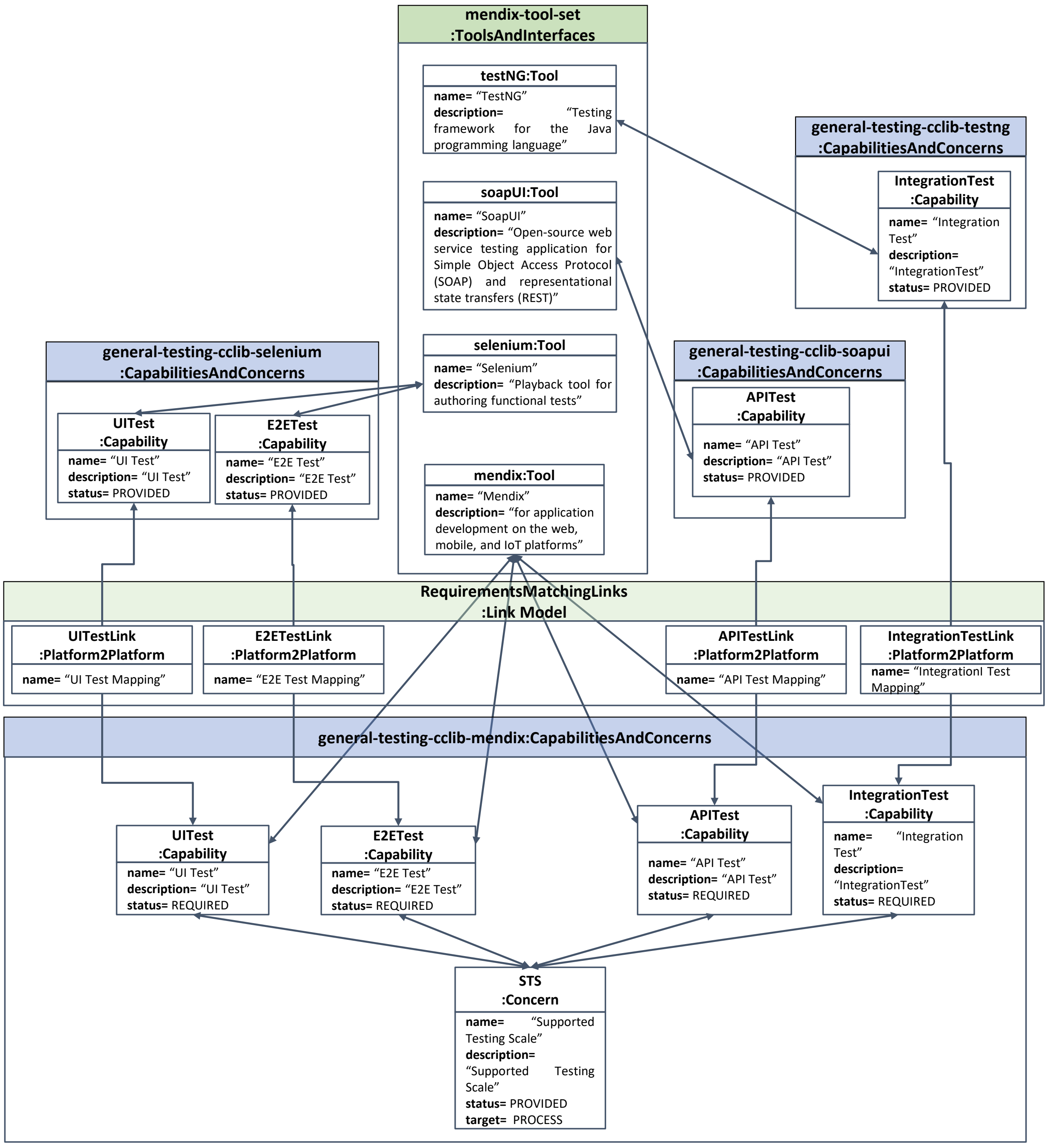

mendix-tool-set:ToolsAndInterfaces
testNG:Tool
name= "TestNG"
description= "Testing framework for the Java programming language"
soapUI:Tool
name= "SoapUI"
description= "Open-source web service testing application for Simple Object Access Protocol (SOAP) and representational state transfers (REST)"
selenium:Tool
name= "Selenium"
description= "Playback tool for authoring functional tests"
mendix:Tool
name= "Mendix"
description= "for application development on the web, mobile, and IoT platforms"
general-testing-cclib-testng :CapabilitiesAndConcerns
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= PROVIDED
general-testing-cclib-soapui :CapabilitiesAndConcerns
APITest :Capability
name= "API Test"
description= "API Test"
status= PROVIDED
general-testing-cclib-selenium :CapabilitiesAndConcerns
E2ETest :Capability
name= "E2E Test"
description= "E2E Test"
status= PROVIDED
UITest :Capability
name= "UI Test"
description= "UI Test"
status= PROVIDED
test-generation-testing-cclib-mendix :CapabilitiesAndConcerns
test-execution-testing-cclib-mendix :CapabilitiesAndConcerns
test-evaluation-testing-cclib-mendix :CapabilitiesAndConcerns
test-design-testing-cclib-mendix :CapabilitiesAndConcerns
general-testing-cclib-mendix:CapabilitiesAndConcerns
UnitTest :Capability
name= "Unit Test"
description= "Unit Test"
status= PROVIDED
E2ETest :Capability
name= "E2E Test"
description= "E2E Test"
status= REQUIRED
APITest :Capability
name= "API Test"
description= "API Test"
status= REQUIRED
UITest :Capability
name= "UI Test"
description= "UI Test"
status= REQUIRED
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= REQUIRED
ImpEx-Test :Capability
name= "Import/Export Testing sources"
description= "import/export techniques to reuse testing artifacts of other sources"
status= PROVIDED
WebTechIntTest :Capability
name= "Web Tech. Integration Tests"
description= "completely open test integration via web-technologies"
status= PROVIDED
PerformanceTest :Capability
name= "Performance Test"
description= "Performance Test"
status= PROVIDED
FunctionalityTest :Capability
name= "Functionality Test"
description= "Functionality Test"
status= PROVIDED
NewLCTF :Capability
name= "New LCTF"
description= "New Low Code Testing Framework (LCTF) dedicated to LCDP"
status= PROVIDED
ThirdPartyQA :Capability
name= "ThirdPartyQA"
description= "Leveraging third-party frameworks (e. g., Selenium, TestNG)"
status= PROVIDED
Capabilities
Concerns
STS :Concern
name= "Supported Testing Scale"
description= "Supported Testing Scale"
status= PROVIDED
target= PROCESS
TestingOpenness :Concern
name= "Testing Openness"
description= "Openness to third-party testing tools. enabling the test component to integrate with other testing tools since it allows reusing the existing resources"
status= PROVIDED
target= PROCESS
VS :Concern
name= "Verification Support"
description= "Specifies the characteristics (functional and non-functional) of the system that can be verified"
status= PROVIDED
target= PROCESS
TF :Concern
name= "Testing Framework"
description= "About building a test component"
status= PROVIDED
target= PROCESS

Figure 8: Mendix provided and required capabilities and concerns

`required` to the *status* property. Figure 8 shows the provided and required capabilities for the Mendix LCDP as evaluated and discussed in Section 2.1.2. The evaluation of testing features of other LCDPs are given in [2]. In this context, Selenium, SoapUI and TestNG tools are third-party tool providing capabilities to Mendix. Finally, if a testing feature, i.e., a capability from the imported testing CC library is not evaluated, i.e., no value is assigned to its *status* property, it is removed from the set of imported elements[8].

- Finally, links among provided and required capabilities of integrated tools, as collected in the Mendix-Toolset TI library, are established by matching required Mendix capabilities to the corresponding provided ones by integrated third-party tools (Selenium, SoapUI and TestNG). In DevOpsML this model weaving [42] step is supported by links collected in a separated Link Model conforming to the linking metamodel introduced in Section 3.1.3. In particular, since the CC library and the TI library depicted in Figures 7 and 8 are platform models conforming to the DevOpsML platform metamodel, the links are instances of the Platform2Platform metaclass. (See Figure 9).



mendix-tool-set :ToolsAndInterfaces
testNG:Tool
name= "TestNG"
description= "Testing framework for the Java programming language"
soapUI:Tool
name= "SoapUI"
description= "Open-source web service testing application for Simple Object Access Protocol (SOAP) and representational state transfers (REST)"
selenium:Tool
name= "Selenium"
description= "Playback tool for authoring functional tests"
mendix:Tool
name= "Mendix"
description= "for application development on the web, mobile, and IoT platforms"
general-testing-cclib-testng :CapabilitiesAndConcerns
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= PROVIDED
general-testing-cclib-selenium :CapabilitiesAndConcerns
UITest :Capability
name= "UI Test"
description= "UI Test"
status= PROVIDED
E2ETest :Capability
name= "E2E Test"
description= "E2E Test"
status= PROVIDED
general-testing-cclib-soapui :CapabilitiesAndConcerns
APITest :Capability
name= "API Test"
description= "API Test"
status= PROVIDED
RequirementsMatchingLinks :Link Model
UITestLink :Platform2Platform
name= "UI Test Mapping"
E2ETestLink :Platform2Platform
name= "E2E Test Mapping"
APITestLink :Platform2Platform
name= "API Test Mapping"
IntegrationTestLink :Platform2Platform
name= "Integrationl Test Mapping"
general-testing-cclib-mendix:CapabilitiesAndConcerns
UITest :Capability
name= "UI Test"
description= "UI Test"
status= REQUIRED
E2ETest :Capability
name= "E2E Test"
description= "E2E Test"
status= REQUIRED
APITest :Capability
name= "API Test"
description= "API Test"
status= REQUIRED
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= REQUIRED
STS :Concern
name= "Supported Testing Scale"
description= "Supported Testing Scale"
status= PROVIDED
target= PROCESS

Figure 9: Linking Mendix third-party required capabilities with capabilities provided by external tools

3.2.3 Modeling Continuous Software Engineering Processes.

Figure 10 shows a DevOps process model created using SPEM as software process modeling language (SPML). In particular, following the DevOpsML guideline given in [3], an high-level DevOps pipeline of

[8]It is worth noting that this approach is suitable only for *import-by-value* scenarios i.e., when the imported libraries are copied in the recipient models. In an *import-by-reference* scenario, deleting an element from a reusable library would cause inconsistencies in other recipient models.

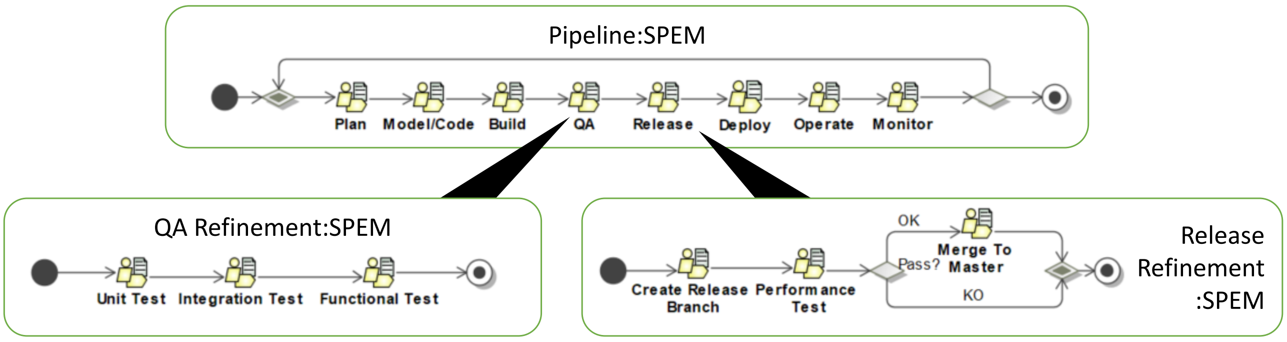


Figure 10: A given DevOps process example

SPEM activities (plan, model/code, build, QA, release, deploy, operate, and monitor) are further refined to show QA and release sub-activities.

In particular, in this deliverable, we are interested in model-driven continuous software engineering (CSE) processes, i.e., engineering processes suitably combining DevOps [36] and MDE [42] principles and practices [1].

Finally, it is worth noting that the considered CSE process (model) can be of arbitrary complexity. The process model is a required input for the requirements engineer to suitably map tools, capabilities and concerns, possibly available in reusable CC and TI libraries, to a specific DevOps phase (by setting the enumerated *devOpsPhase* property, see Figure 11).

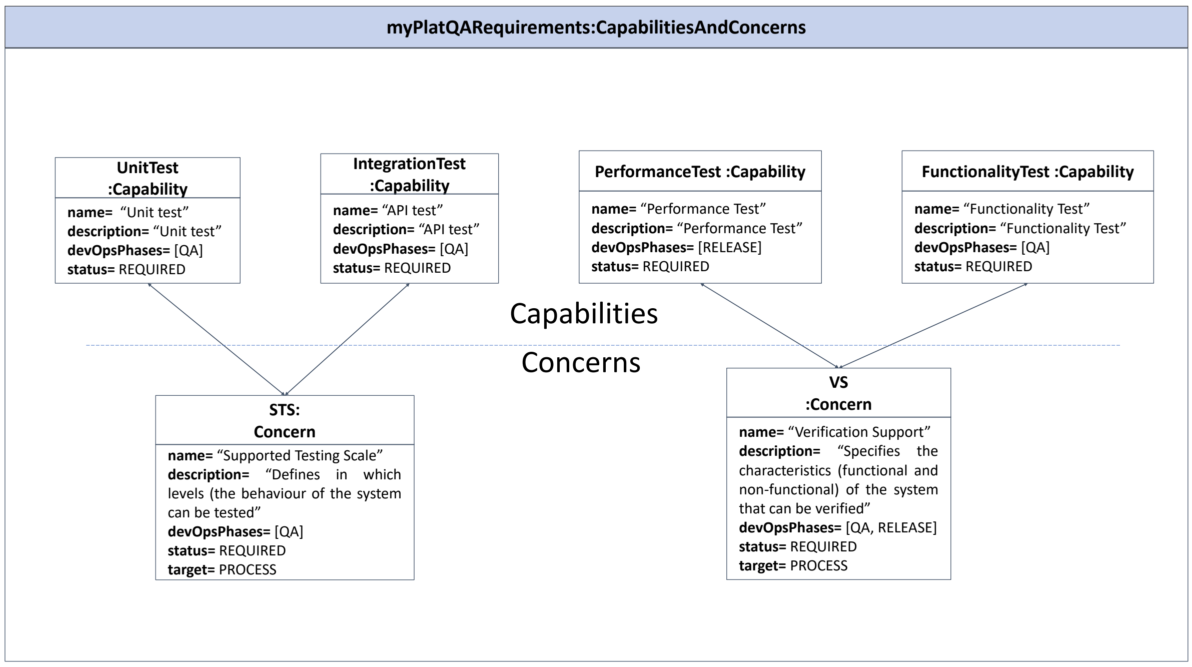

myPlatQARequirements:CapabilitiesAndConcerns
UnitTest :Capability
name= "Unit test"
description= "Unit test"
devOpsPhases= [QA]
status= REQUIRED
IntegrationTest :Capability
name= "API test"
description= "API test"
devOpsPhases= [QA]
status= REQUIRED
PerformanceTest :Capability
name= "Performance Test"
description= "Performance Test"
devOpsPhases= [RELEASE]
status= REQUIRED
FunctionalityTest :Capability
name= "Functionality Test"
description= "Functionality Test"
devOpsPhases= [QA]
status= REQUIRED
Capabilities
Concerns
STS: Concern
name= "Supported Testing Scale"
description= "Defines in which levels (the behaviour of the system can be tested"
devOpsPhases= [QA]
status= REQUIRED
target= PROCESS
VS :Concern
name= "Verification Support"
description= "Specifies the characteristics (functional and non-functional) of the system that can be verified"
devOpsPhases= [QA, RELEASE]
status= REQUIRED
target= PROCESS

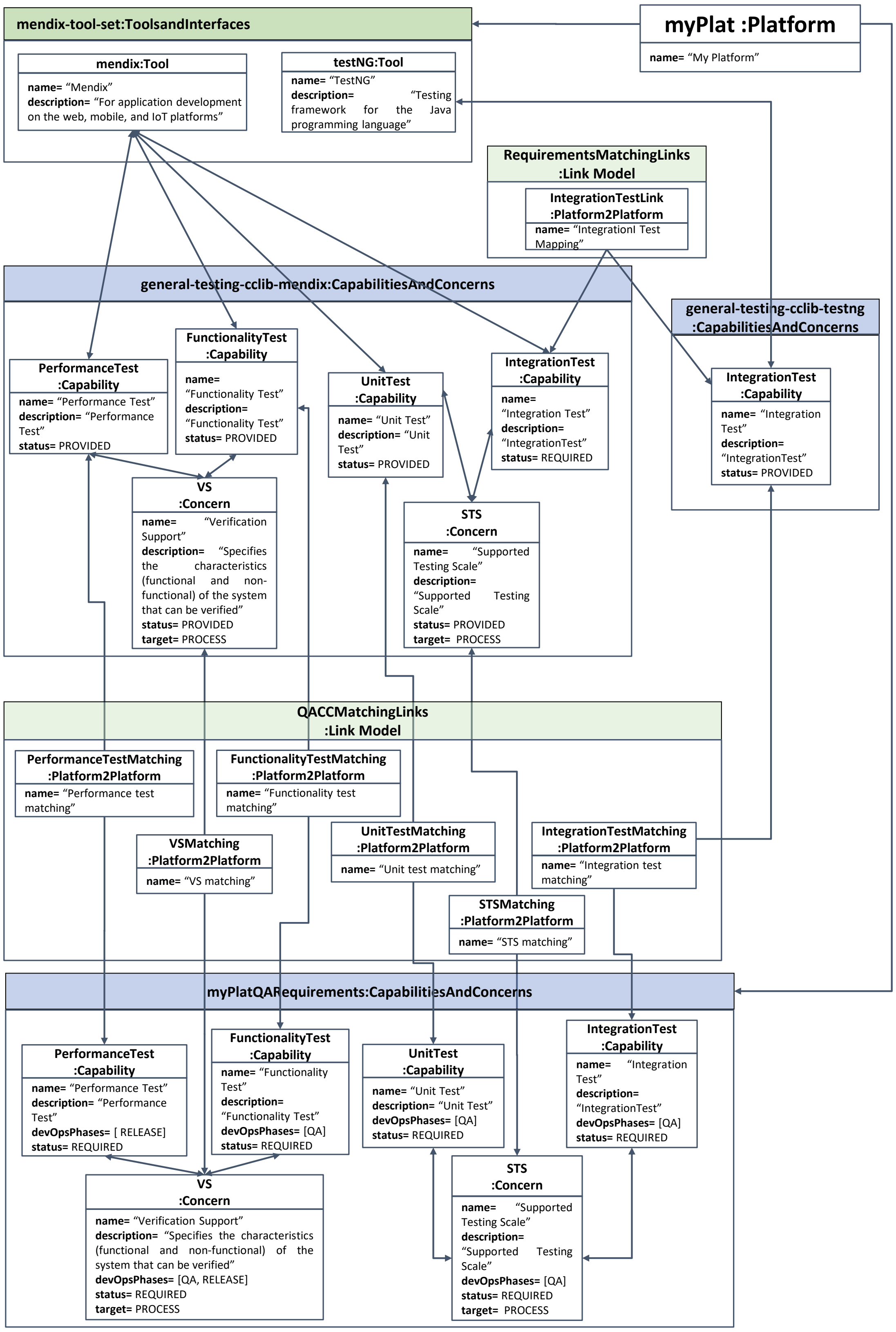
Figure 11: Example of required testing capabilities and concerns for a concrete DevOps. They should be provided by a Requirements Engineer.

3.2.4 Platform Requirements Modeling

In DevOpsML, a requirement specification activity is possible by modeling and collecting required (*status*=`required`) *PlatformElements* (i.e., capabilities, concerns, tools, and interfaces). A coarse-grained categorization of requirements is possible in process, platform, and resource requirements [46], according to the value assigned to *target* property of any *Concern* of interest (see Figure 4).

In order to perform this step, a requirements engineer should rely on i) a existing set of CC libraries, i.e., capabilities and concerns of interests, and ii) process model(s) (see Figure 10). Once this information is available, she can specify process requirements in terms of a collection of *PlatformElements*.

Figure 11 shows a possible requirement specification for low-code testing framework (LCTF) using the CC libraries describing low-code testing features. In particular, we are considering four capabilities, i.e., *UnitTest*, *IntegrationTest*, *PerformanceTest*, and *FunctionalityTest*), which are required in order to address testing concerns (i.e. low-code testing features in Table 2.1), i.e.,testing scale (*STS*) and verification support (*VS*) of the corresponding SPEM activities depicted in the process model (see Figure 10). It is worth noting that, at this step, the links between the platform requirements and a CSE process (Figure 10) are not yet explicitly modeled.



mendix-tool-set:ToolsandInterfaces
mendix:Tool
name= "Mendix"
description= "For application development on the web, mobile, and IoT platforms"
testNG:Tool
name= "TestNG"
description= "Testing framework for the Java programming language"
myPlat :Platform
name= "My Platform"
RequirementsMatchingLinks :Link Model
IntegrationTestLink :Platform2Platform
name= "IntegrationI Test Mapping"
general-testing-cclib-mendix:CapabilitiesAndConcerns
PerformanceTest :Capability
name= "Performance Test"
description= "Performance Test"
status= PROVIDED
FunctionalityTest :Capability
name= "Functionality Test"
description= "Functionality Test"
status= PROVIDED
UnitTest :Capability
name= "Unit Test"
description= "Unit Test"
status= PROVIDED
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= REQUIRED
VS :Concern
name= "Verification Support"
description= "Specifies the characteristics (functional and non-functional) of the system that can be verified"
status= PROVIDED
target= PROCESS
STS :Concern
name= "Supported Testing Scale"
description= "Supported Testing Scale"
status= PROVIDED
target= PROCESS
general-testing-cclib-testng :CapabilitiesAndConcerns
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= PROVIDED
QACCMatchingLinks :Link Model
PerformanceTestMatching :Platform2Platform
name= "Performance test matching"
FunctionalityTestMatching :Platform2Platform
name= "Functionality test matching"
VSMatching :Platform2Platform
name= "VS matching"
UnitTestMatching :Platform2Platform
name= "Unit test matching"
IntegrationTestMatching :Platform2Platform
name= "Integration test matching"
STSMatching :Platform2Platform
name= "STS matching"
myPlatQARequirements:CapabilitiesAndConcerns
PerformanceTest :Capability
name= "Performance Test"
description= "Performance Test"
devOpsPhases= [RELEASE]
status= REQUIRED
FunctionalityTest :Capability
name= "Functionality Test"
description= "Functionality Test"
devOpsPhases= [QA]
status= REQUIRED
UnitTest :Capability
name= "Unit Test"
description= "Unit Test"
devOpsPhases= [QA]
status= REQUIRED
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
devOpsPhases= [QA]
status= REQUIRED
VS :Concern
name= "Verification Support"
description= "Specifies the characteristics (functional and non-functional) of the system that can be verified"
devOpsPhases= [QA, RELEASE]
status= REQUIRED
target= PROCESS
STS :Concern
name= "Supported Testing Scale"
description= "Supported Testing Scale"
devOpsPhases= [QA]
status= REQUIRED
target= PROCESS

Figure 12: Requirement and provisioning matching.

3.2.5 Platform Capabilities Analysis

During this step, tools are evaluated w.r.t. their capabilities to support a given CSE process and to be eventually integrated in a DevOps platform (see Figure 3).

In particular, tools' provided [9] capabilities and concerns are compared with required ones, assessed during the Platform Requirement Modeling step (Section 3.2.4).

The analysis consists in mapping i) `required` *Capabilities* for `process` *Concerns* with ii) *Capabilities* `provided` by candidate Tools.

In DevOpsML, the mapping is realised via model weaving by creating links between the matched model elements.

This step is exemplified by Figure 12. A candidate *Platform* (myPlat) is modeled by combining CC and TI libraries, i.e., separated and reusable Platform Models via Platform2Platform links stored in Link Models. As a result a composite Platform Model is obtained.

In particular, Figure 12 represents a candidate *Platform* myPlat where the Mendix CC library of Figure 8 are linked, via **-Matching* Platform2Platform links, with platform requirements (i.e., `required` capabilities and concerns) of Figure 11.

It is worth noting how the integration test Capability is not directly provided by Mendix (the status of the *IntegrationTest* capability in the *general-testing-cclib-mendix* CC library is `required`) but the integrated third-party tool TestNG. For this reason an additional Platform2Platform link is shown, matching the required and provided capabilities from Mendix and TestNG CC libraries, respectively.

3.2.6 DevOps Process and DevOps Platform Weaving

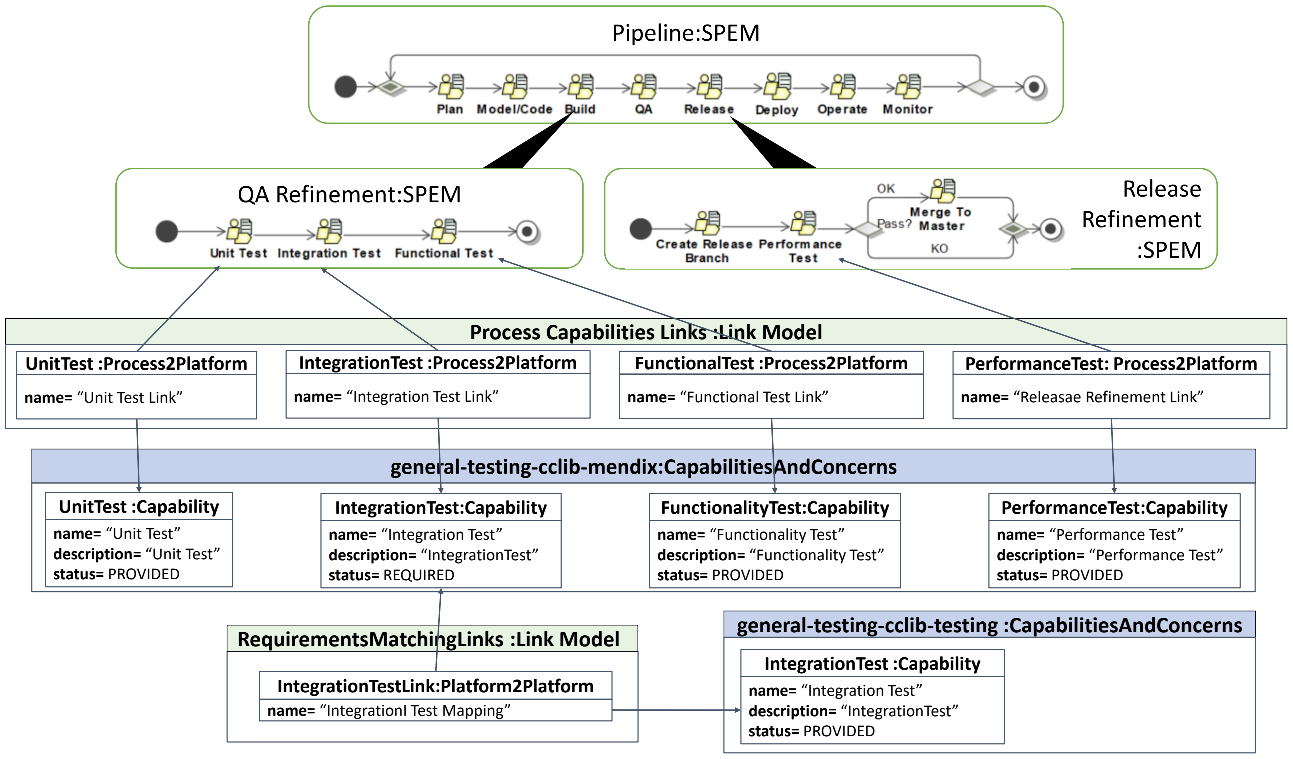

Pipeline:SPEM
Plan
Model/Code
Build
QA
Release
Deploy
Operate
Monitor
QA Refinement:SPEM
Unit Test
Integration Test
Functional Test
OK
Pass?
Merge To Master
KO
Create Release Branch
Performance Test
Release Refinement :SPEM
Process Capabilities Links :Link Model
UnitTest :Process2Platform
name= "Unit Test Link"
IntegrationTest :Process2Platform
name= "Integration Test Link"
FunctionalTest :Process2Platform
name= "Functional Test Link"
PerformanceTest: Process2Platform
name= "Releasae Refinement Link"
general-testing-cclib-mendix:CapabilitiesAndConcerns
UnitTest :Capability
name= "Unit Test"
description= "Unit Test"
status= PROVIDED
IntegrationTest:Capability
name= "Integration Test"
description= "IntegrationTest"
status= REQUIRED
FunctionalityTest:Capability
name= "Functionality Test"
description= "Functionality Test"
status= PROVIDED
PerformanceTest:Capability
name= "Performance Test"
description= "Performance Test"
status= PROVIDED
RequirementsMatchingLinks :Link Model
IntegrationTestLink:Platform2Platform
name= "IntegrationI Test Mapping"
general-testing-cclib-testing :CapabilitiesAndConcerns
IntegrationTest :Capability
name= "Integration Test"
description= "IntegrationTest"
status= PROVIDED

Figure 13: DevOpsML Model: mapping DevOps Process with DevOps platform models.

The last step of the BPMN in Figure 6 concerns the combination of the configured platform with the supported DevOps process. The resulting artifact is a DevOpsML Model [3] depicted in Figure 13.

The Process Model and Platform Model(s) are combined again through the model weaving integration mechanisms provided in DevOpsML [3], i.e., with a Link Model (*Process Capabilities Links*), where four *Process2Platform* links match the Mendix testing capabilities (both directly provided or provided via third-party tools) with the elements of the process model, in our case SPEM activities.

[9] directly or via integrated tools

4 Conclusion

Low-code domain is more and more popular but there exist limited academic resources for low-code testing. Testing is a mandatory step of any development process. The integration of DevOps principles and practices in low-code engineering platforms (LCEP) is gaining attention by the research community since LCE is both concerned by (i) the development of low-code software and (ii) its operation on the cloud and using dedicated repository (cf deliverable D4.1).

We performed several analyses considering testing requirements in LCDPs, published them [2], and reported them in this deliverable. 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 Mendix LCDP based on the proposed feature list.

We outlined the first prototype of DevOpsML, a model-driven framework for modeling DevOps processes and platforms and their weaving. In this deliverable, we show how the DevOpsML framework can be used to create and combine testing processes and low-code platform specifications to support these testing features.

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.

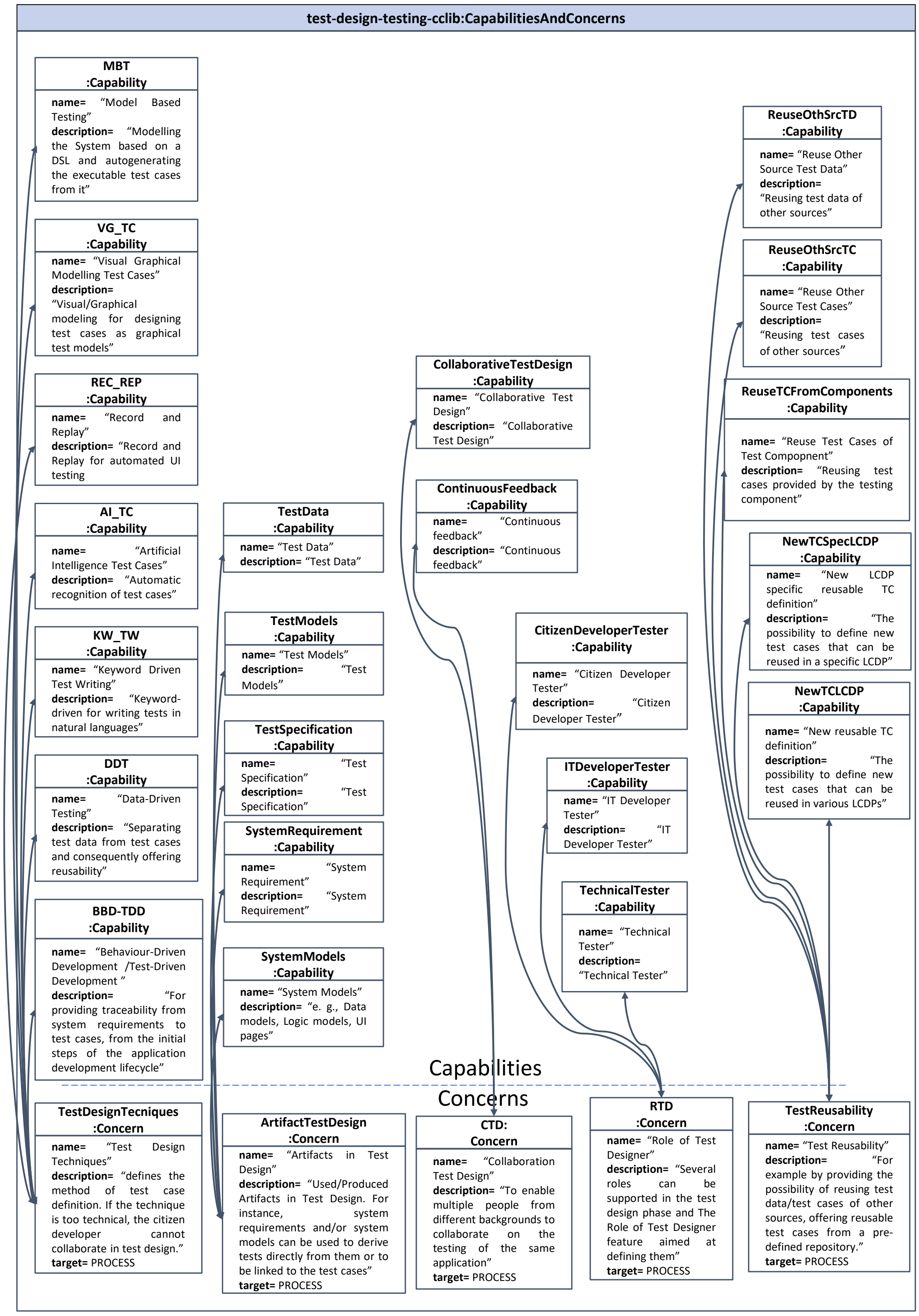
As our future work, we will initially work on the challenges of the first category. At the moment, we are defining a running example to show how the DSL extension algorithm would work in practice and how different kinds of DSLs (i. e., interpreted and compiled) could be supported. Afterwards, we will implement the generic extension algorithm, so different system DSLs can be extended with an appropriate test DSLs satisfying LCE objectives [4]. Finally, supporting the cloud focusing on the operation part of DevOps, and building tools for generating cloud-based low-code testing component for a given LCDP, are in our future research plan.

Appendix

Capabilities and Concerns libraries.

This section contains the Capabilities and Concerns libraries generated following the instructions given in Section 3.2.1.

In Figure 14 we represent the CC library `test-design-testing-cclib`. This library is the model of the features under the category *Test Design* in Table 1.



test-design-testing-cclib:CapabilitiesAndConcerns
MBT :Capability
name= "Model Based Testing"
description= "Modelling the System based on a DSL and autogenerating the executable test cases from it"
VG_TC :Capability
name= "Visual Graphical Modelling Test Cases"
description= "Visual/Graphical modeling for designing test cases as graphical test models"
REC_REP :Capability
name= "Record and Replay"
description= "Record and Replay for automated UI testing
AI_TC :Capability
name= "Artificial Intelligence Test Cases"
description= "Automatic recognition of test cases"
KW_TW :Capability
name= "Keyword Driven Test Writing"
description= "Keyword-driven for writing tests in natural languages"
DDT :Capability
name= "Data-Driven Testing"
description= "Separating test data from test cases and consequently offering reusability"
BBD-TDD :Capability
name= "Behaviour-Driven Development /Test-Driven Development "
description= "For providing traceability from system requirements to test cases, from the initial steps of the application development lifecycle"
TestData :Capability
name= "Test Data"
description= "Test Data"
TestModels :Capability
name= "Test Models"
description= "Test Models"
TestSpecification :Capability
name= "Test Specification"
description= "Test Specification"
SystemRequirement :Capability
name= "System Requirement"
description= "System Requirement"
SystemModels :Capability
name= "System Models"
description= "e. g., Data models, Logic models, UI pages"
CollaborativeTestDesign :Capability
name= "Collaborative Test Design"
description= "Collaborative Test Design"
ContinuousFeedback :Capability
name= "Continuous feedback"
description= "Continuous feedback"
CitizenDeveloperTester :Capability
name= "Citizen Developer Tester"
description= "Citizen Developer Tester"
ITDeveloperTester :Capability
name= "IT Developer Tester"
description= "IT Developer Tester"
TechnicalTester :Capability
name= "Technical Tester"
description= "Technical Tester"
ReuseOthSrcTD :Capability
name= "Reuse Other Source Test Data"
description= "Reusing test data of other sources"
ReuseOthSrcTC :Capability
name= "Reuse Other Source Test Cases"
description= "Reusing test cases of other sources"
ReuseTCFromComponents :Capability
name= "Reuse Test Cases of Test Compopnent"
description= "Reusing test cases provided by the testing component"
NewTCSpecLCDP :Capability
name= "New LCDP specific reusable TC definition"
description= "The possibility to define new test cases that can be reused in a specific LCDP"
NewTCLCDP :Capability
name= "New reusable TC definition"
description= "The possibility to define new test cases that can be reused in various LCDPs"
Capabilities
Concerns
TestDesignTecniques :Concern
name= "Test Design Techniques"
description= "defines the method of test case definition. If the technique is too technical, the citizen developer cannot collaborate in test design."
target= PROCESS
ArtifactTestDesign :Concern
name= "Artifacts in Test Design"
description= "Used/Produced Artifacts in Test Design. 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"
target= PROCESS
CTD: Concern
name= "Collaboration Test Design"
description= "To enable multiple people from different backgrounds to collaborate on the testing of the same application"
target= PROCESS
RTD :Concern
name= "Role of Test Designer"
description= "Several roles can be supported in the test design phase and The Role of Test Designer feature aimed at defining them"
target= PROCESS
TestReusability :Concern
name= "Test Reusability"
description= "For example by providing the possibility of reusing test data/test cases of other sources, offering reusable test cases from a pre-defined repository."
target= PROCESS

Figure 14: Test Design Capabilities and Concerns libraries. Modeled on Test Design features of Table 1

In Figure 15 we represent the CC library `test-generation-testing-cclib`. This library is the model of the features under the category *Test Generation* in Table 1.

In Figure 16 we represent the CC library `test-execution-testing-cclib`. This library is the

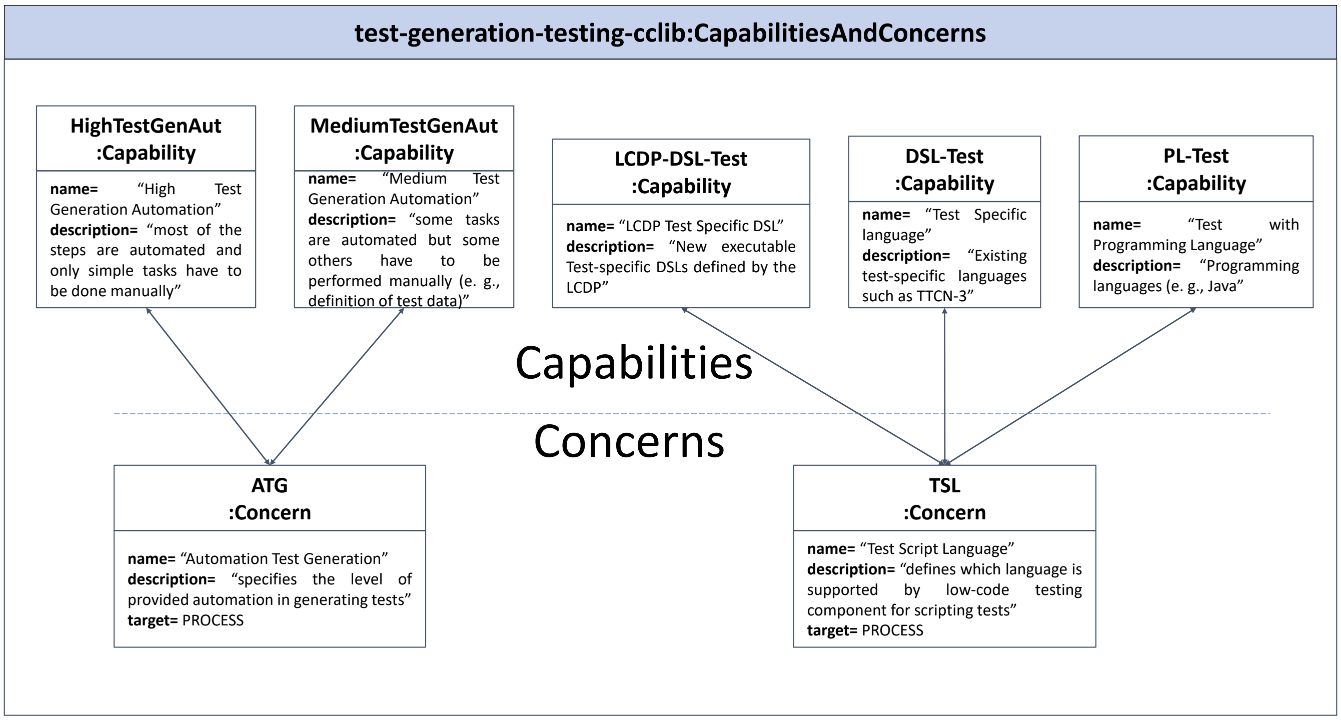

test-generation-testing-cclib:CapabilitiesAndConcerns
HighTestGenAut :Capability
name= "High Test Generation Automation"
description= "most of the steps are automated and only simple tasks have to be done manually"
MediumTestGenAut :Capability
name= "Medium Test Generation Automation"
description= "some tasks are automated but some others have to be performed manually (e. g., definition of test data)"
LCDP-DSL-Test :Capability
name= "LCDP Test Specific DSL"
description= "New executable Test-specific DSLs defined by the LCDP"
DSL-Test :Capability
name= "Test Specific language"
description= "Existing test-specific languages such as TTCN-3"
PL-Test :Capability
name= "Test with Programming Language"
description= "Programming languages (e. g., Java"
Capabilities
Concerns
ATG :Concern
name= "Automation Test Generation"
description= "specifies the level of provided automation in generating tests"
target= PROCESS
TSL :Concern
name= "Test Script Language"
description= "defines which language is supported by low-code testing component for scripting tests"
target= PROCESS

Figure 15: Test Generation Capabilities and Concerns libraries. Modeled on Test Generation features of Table 1

model of the features under the category *Test Execution* in Table 1.

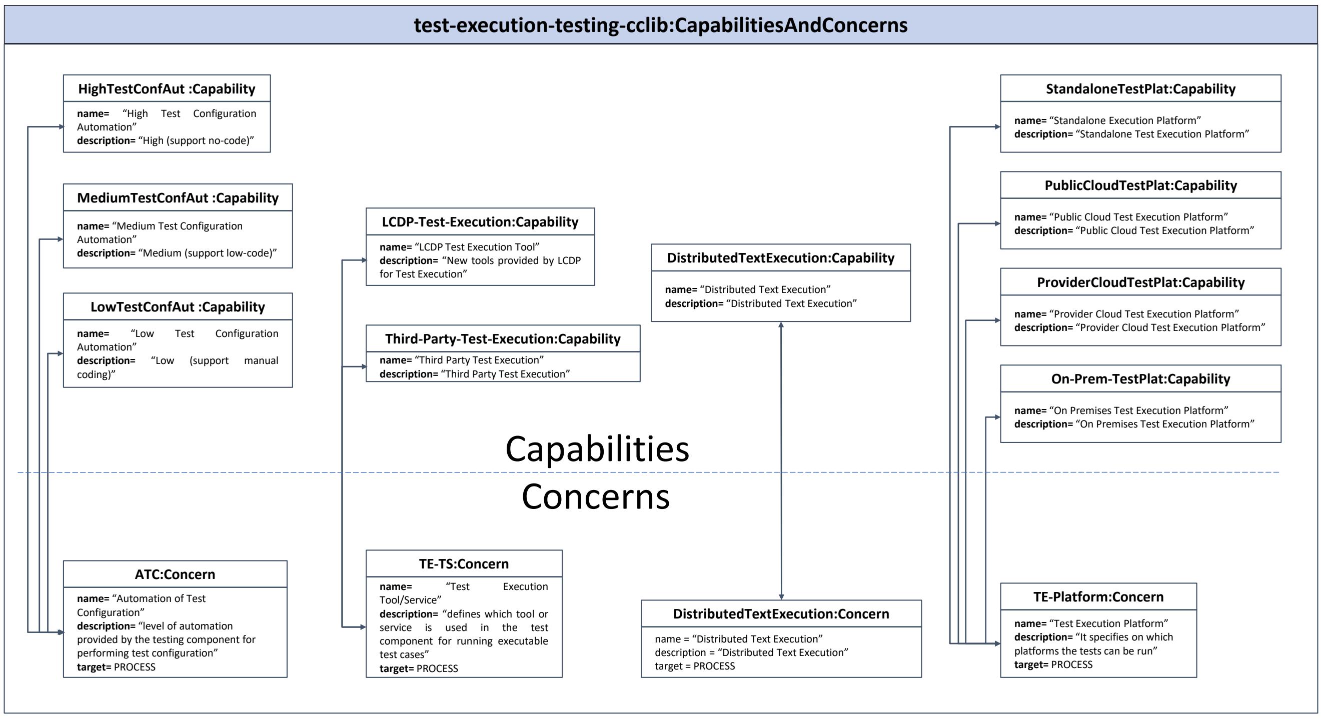

test-execution-testing-cclib:CapabilitiesAndConcerns
HighTestConfAut :Capability
name= "High Test Configuration Automation"
description= "High (support no-code)"
MediumTestConfAut :Capability
name= "Medium Test Configuration Automation"
description= "Medium (support low-code)"
LowTestConfAut :Capability
name= "Low Test Configuration Automation"
description= "Low (support manual coding)"
LCDP-Test-Execution:Capability
name= "LCDP Test Execution Tool"
description= "New tools provided by LCDP for Test Execution"
Third-Party-Test-Execution:Capability
name= "Third Party Test Execution"
description= "Third Party Test Execution"
DistributedTextExecution:Capability
name= "Distributed Text Execution"
description= "Distributed Text Execution"
StandaloneTestPlat:Capability
name= "Standalone Execution Platform"
description= "Standalone Test Execution Platform"
PublicCloudTestPlat:Capability
name= "Public Cloud Test Execution Platform"
description= "Public Cloud Test Execution Platform"
ProviderCloudTestPlat:Capability
name= "Provider Cloud Test Execution Platform"
description= "Provider Cloud Test Execution Platform"
On-Prem-TestPlat:Capability
name= "On Premises Test Execution Platform"
description= "On Premises Test Execution Platform"
Capabilities
Concerns
ATC:Concern
name= "Automation of Test Configuration"
description= "level of automation provided by the testing component for performing test configuration"
target= PROCESS
TE-TS:Concern
name= "Test Execution Tool/Service"
description= "defines which tool or service is used in the test component for running executable test cases"
target= PROCESS
DistributedTextExecution:Concern
name = "Distributed Text Execution"
description = "Distributed Text Execution"
target = PROCESS
TE-Platform:Concern
name= "Test Execution Platform"
description= "It specifies on which platforms the tests can be run"
target= PROCESS

Figure 16: Test Execution Capabilities and Concerns libraries. Modeled on Test Execution features of Table 1

In Figure 17 we represent the CC library `test-evaluation-testing-cclib`. This library is the model of the features under the category *Test Evaluation* in Table 1.

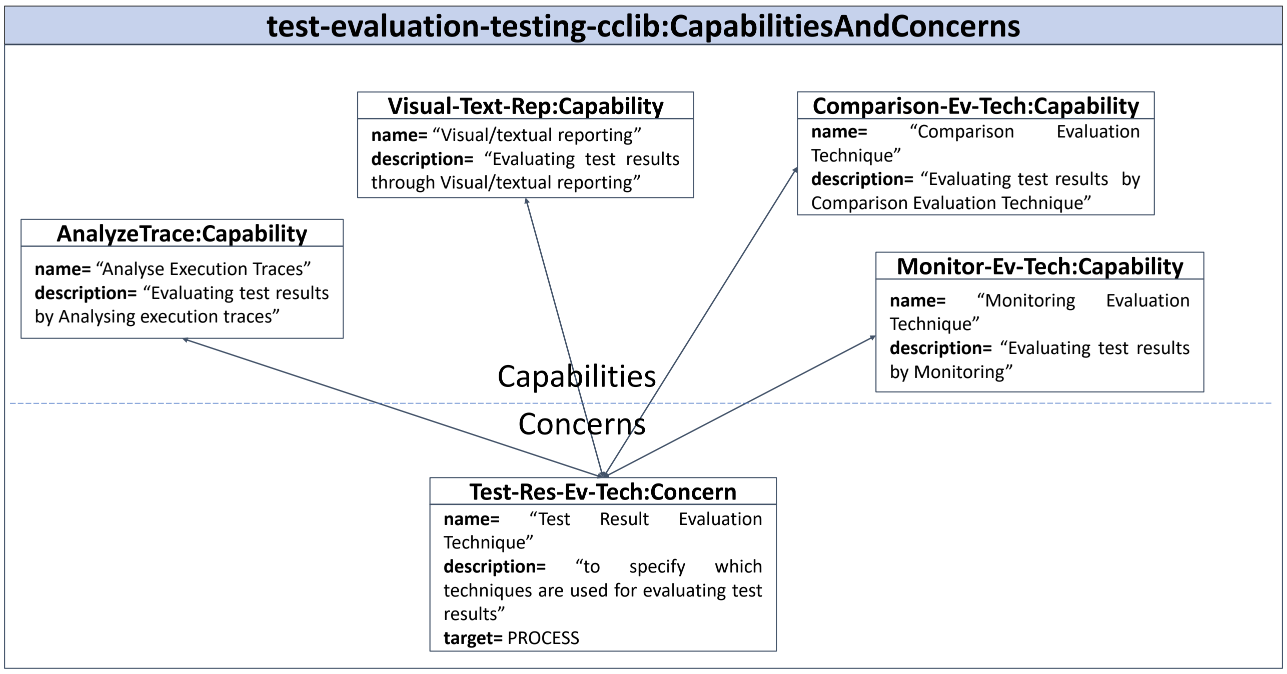

test-evaluation-testing-cclib:CapabilitiesAndConcerns
Visual-Text-Rep:Capability
name= "Visual/textual reporting"
description= "Evaluating test results through Visual/textual reporting"
Comparison-Ev-Tech:Capability
name= "Comparison Evaluation Technique"
description= "Evaluating test results by Comparison Evaluation Technique"
AnalyzeTrace:Capability
name= "Analyse Execution Traces"
description= "Evaluating test results by Analysing execution traces"
Monitor-Ev-Tech:Capability
name= "Monitoring Evaluation Technique"
description= "Evaluating test results by Monitoring"
Capabilities
Concerns
Test-Res-Ev-Tech:Concern
name= "Test Result Evaluation Technique"
description= "to specify which techniques are used for evaluating test results"
target= PROCESS

Figure 17: Test Evaluation Capabilities and Concerns libraries. Modeled on Test Evaluation features of Table 1

References

[1] Jokin Garcia and Jordi Cabot. Stepwise adoption of continuous delivery in model-driven engineering. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 19–32, Cham, 2019. Springer International Publishing.

[2] Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. Challenges and opportunities in low-code testing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. Devopsml: towards modeling devops processes and platforms. In Esther Guerra and Ludovico Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 69:1–69:10. ACM, 2020.

[4] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 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, July 2019.

[5] Paul Vincent, Kimihiko Lijima, Mark Driver, Jason Wong, and Yefim Natis. Magic quadrant for enterprise low-code application platforms. Technical report, February 2019.

[6] John R Rymer and Rob Koplowitz. The forrester wave$^{\text{TM}}$: Low-code development platforms for ad&d professionals. Technical report, March 2019.

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

[8] Mendix Technology. Where thinkers become makers, 2020.

[9] Microsoft. The world needs great solutions, build yours faster, 2020.

[10] Salesforce. Build apps on the customer 360 platform with no-code tools and take your crm to the next level, 2020.

[11] Kony Inc. Leading multi experience development platform, 2020.

[12] Outsystems. Innovate with no limits, 2020.

[13] Mendix Technology. Testing, 2020.

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

[15] Selenium. Selenium automates browsers. that's it!, 2020.

[16] International Organization for Standardization. Iso/iec 25010:2011-systems and software engineering- systems and software quality requirements and evaluation (square)- system and software quality models, 2011. Available in electronic form for online purchase at https://www.iso.org/standard/35733.html.

[17] Mendix Technology. Quality add-ons guide, 2020.

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

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

[20] Mendix Technology. Microflows, 2020.

[21] Outsystems. How to automate unit testing and api testing, 2020.

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

[23] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay. A framework for temporal verification support in domain-specific modelling. *IEEE Transactions on Software Engineering*, 46(4):362–404, 2020.

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

[25] Joachim Herschmann, Thomas Murphy, and Jim Scheibmeir. Magic quadrant for software test automation. Technical report, November 2019.

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

[27] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A systematic review on cloud testing. *ACM Computing Surveys*, 52(5):1–42, 2019.

[28] A. D. Francesco, C. D. Napoli, M. Giordano, G. Ottaviano, R. Perego, and N. Tonellotto. A soa testing platform on the cloud: The midas experience. In *2014 International Conference on Intelligent Networking and Collaborative Systems*, pages 659–664, 2014.

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

[30] 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. The midas cloud platform for testing soa applications. In *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, 2015.

[31] Steffen Herbold, Patrick Harms, and Jens Grabowski. 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):309–324, 2017.

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

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

[34] Francis Bordeleau, Jordi Cabot, Juergen Dingel, Bassem S. Rabil, and Patrick Renaud. Towards modeling framework for devops: Requirements derived from industry use case. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 139–151, Cham, 2020. Springer International Publishing.

[35] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6), November 2019.

[36] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Gartner. Gartner says by 2016, devops will evolve from a niche to a mainstream strategy employed by 25 percent of global 2000 organizations, 2015. https://tinyurl.com/y556a8moU, last accessed on 28/08/20.

[38] Necco Ceresani. The periodic table of devops tools v.2 is here, June 2016. https://blog.xebialabs.com/2016/06/14/periodic-table-devops-tools-v-2/, last accessed on 28/08/20.

[39] Håvard Myrbakken and Ricardo Colomo-Palacios. DevSecOps: a multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination*, volume 770, pages 17–29. Springer, Springer Verlag, 2017.

[40] Yingnong Dang, Qingwei Lin, and Peng Huang. AIOps: Real-world challenges and research innovations. *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: ICSE-Companion*, pages 4–5, 2019.

[41] Julián Alberto García-García, José Gonzalez Enríquez, and Francisco José Domínguez Mayo. Characterizing and evaluating the quality of software process modeling language: *Comparison of ten representative model-based languages. Comput. Stand. Interfaces*, 63:52–66, 2019.

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

[43] Linz BISE Institute, JKU. Devopsml, 2020. https://github.com/lowcomote/devopsml/tree/1.2.2, last accessed on 28/08/20.

[44] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[45] Klaas van den Berg, J.M. Conejero, and R Chitchyan. *AOSD Ontology 1.0: Public Ontology of Aspect-Orientation*. Number AOSD-E in AOSD-Europe-UT-01. AOSD Europe, 5 2005. AOSD-Europe-UT-01.

[46] Miriam Schleipen and Rainer Drath. Three-view-concept for modeling process or manufacturing plants with AutomationML. In *Proceedings of ETFA 2009 - 2009 IEEE Conference on Emerging Technologies and Factory Automation*, pages 1–4. IEEE, 2009.

[47] E. Breton and J. Bezivin. Process-centered model engineering. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 179–182. IEEE, Sep. 2001.

[48] Marlon Dumas, Wil M Van der Aalst, and Arthur H Ter Hofstede. *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, 2005.

[49] Dragan Djurić, Dragan Gašević, and Vladan Devedžić. The tao of modeling spaces. *Journal of Object Technology*, 5(8):125–147, November 2006.

[50] João Paulo A. Almeida, Adrian Rutle, Manuel Wimmer, and Thomas Kühne. The MULTI process challenge. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 164–167. IEEE, 2019.

[51] OMG. Spem, 2008. https://www.omg.org/spec/SPEM/About-SPEM/, last accessed on 28/08/20.

[52] OMG. Semantics of a foundational subset for executable uml models, 2018. https://www.omg.org/spec/FUML/, last accessed on 28/08/20.

[53] Benoît Combemale, Ralf Lämmel, and Eric Van Wyk. Slebok: The software language engineering body of knowledge (dagstuhl seminar 17342). In *Dagstuhl Reports*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[54] Loli Burgueño, Federico Ciccozzi, Michalis Famelis, Gerti Kappel, Leen Lambers, Sébastien Mosser, Richard F. Paige, Alfonso Pierantonio, Arend Rensink, Rick Salay, Gabriele Taentzer, Antonio Vallecillo, and Manuel Wimmer. Contents for a Model-Based Software Engineering Body of Knowledge. *Software and Systems Modeling*, 18(6):3193–3205, 2019.

[55] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020.

[56] Dimitrios Kolovos, Louis Rose, Richard Paige, and Antonio Garcıa-Domınguez. The epsilon book. *Structure*, 178:1–10, 2010.