A Pattern-Based Test Platform for Families of Smart Health Products

Pedro Almeida[1], João Pascoal Faria[1,2], and Bruno Lima[1,2]
[1]Faculty of Engineering of the University of Porto, Porto, Portugal
[2]INESC TEC, Porto, Portugal
up202003029@edu.fe.up.pt, jpf@fe.up.pt, bruno.lima@fe.up.pt

***Abstract*—One of the most critical ICT application domains is healthcare, where a single failure can lead a patient into a hazardous situation. Due to this, there's a great necessity to ensure that the developed solutions are safe and secure and perform as expected. Smart-Health-4-All (SH4ALL) is a project aiming at accelerating the research, development, commercialization, and dissemination of trustworthy smart health solutions in Portugal. One of the key components of the project is a web platform that supports the generation of integration and system tests for smart health solutions (comprising medical devices, applications, etc.), following a software product line approach. At the domain engineering level, the platform supports the creation of feature models and related test patterns for families of smart health products. At the product engineering level, the platform supports the instantiation of test patterns and the generation of corresponding test scripts ready for execution on specific products under test. This paper presents the aforementioned test platform and test process, and the discovery of test patterns.**

***Keywords*—pattern-based testing; test patterns; smart health; software product lines**

I. Introduction

Health care is one of the domains with the most impact on the population. Its objectives are to treat and promote the health of the population and provide it with a higher quality of life; this is possible to achieve through the means of medical devices.

Medical devices improve society's quality of life by aiding in various aspects such as monitoring, disease prediction, surgery assistance, and more [1]. In recent years, healthcare has been adopting the Internet of Things (IoT); this adoption has set the stage for an improved service for various patients such as the elderly population and chronically diseased patients [2] [1]. With the aid of these interconnected devices, which are constantly interchanging data with one another and with multiple applications and services and perform actions based on that information, it's possible to create so-called smart health solutions that provide better and more personalized services to the end users.

Medical device recalls have seen a substantial increase, in many cases due to software faults; this may be an indicator that testing is not being performed adequately [3]. Since these devices directly impact the lives of the population, they need to undergo a thorough certification procedure. For this purpose, standards such as ISO 13485 require that the organization, the entity that's developing the device and aims to implement this given standard, create quality management systems [4], ISO/IEC 62304 require that manufacturers implement specific procedures during development (based on a medical device hazard class) before they are released to the market [5]. Implementing these certification procedures wrongly may lead to the device never being released to the market.

Testing software requires a lot of effort from the developers and testers, even more in healthcare, where there needs to be an extremely low defect rate before release. This challenge becomes harder in highly configurable and interconnected systems such as IoT systems for health care and Ambient Assisted Living (AAL).

From a business perspective, complying with regulatory requirements and ensuring adequate testing of medical devices and associated applications and services, may be particularly challenging for Small and Medium Enterprises (SME), and constitute a barrier to market entry.

It is in this context that the Smart-Health-4-All (SH4ALL)[1] comes to rescue, aiming at accelerating the research, development, commercialization, and dissemination of trustworthy smart health solutions, comprising medical devices, applications, and services. The goal of this paper is to describe one of the key components of the project: a web platform that supports the generation of integration and system tests for smart health solutions, following a Software Product Line (SPL) approach. At the domain engineering level, the platform supports the creation of feature models and related test patterns for families of products. At the product engineering level, the platform supports the instantiation of test patterns and the generation of corresponding test scripts ready for execution on specific products under test.

Hence, the main contributions of the work described in this paper are:

- a pattern-based test process and support platform, designed to facilitate the generation of integration and system tests for families of products within a well-defined domain, following a software product line approach to promote reuse and consistency (*platform engineering level*);
- examples of a feature model and a catalog of test patterns developed for a family of products in the eHealth and AAL domain (*domain engineering level*);
- examples demonstrating the instantiation of test patterns and the generation of test cases ready for execution for specific products under test (*product engineering level*).

This paper is divided into six sections: section II provides an overview of the pattern-based test process and platform; section III describes a feature model and a set of test patterns that were defined for a subset of the smart health domain;

[1]https://www.smarthealth4all.com/en

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00066

section IV shows how the test patterns can be instantiated and test cases generated for specific products under test, based on examples from the SH4ALL project; section V addresses the evaluation of the test platform; section VI discusses related work; finally, section VII draws some conclusions and points out future work.

II. Test process and platform

This section presents an overview of the pattern-based test process and platform.

A. Test process

The envisioned test process has the objective of reducing the time to market by accelerating the final stages of product development and testing, through the generation of test cases out of selecting and instantiating features from a feature model. This testing process takes inspiration from Software Product Line Engineering (SPLE) [6].

The first distinctive characteristic of this test process is that it makes use of feature models and links between features and related test patterns to derive test cases. The second characteristic is that it comprises two stages that are inspired in the ones present in SPLE – domain and application engineering [6]. At the domain (or family) level, the actors work on defining the feature model and finding test patterns. At the application (or product) level, the actors select and instantiate features to specify concrete products and generate their specific test cases. An overview of the test process can be seen in Figure 1.

The process starts with a *domain analysis*, i.e., the analysis of commonalities and variability within a family of products and associated test needs. Based on that analysis, it is produced a *feature model* and a set of related *test patterns*. The test patterns are linked to features in the feature model, and parameters of the test patterns are linked to attributes of those features, which permits at a later stage the selection and instantiation of features to characterize concrete products and generate corresponding test cases.

Then, for the generation of test cases for specific products under test, the user only has to characterize the product under test in terms of a selection (subset) of features of the feature model and the instantiation (assignment) of concrete values for any attributes associated with the selected features (*product analysis* activity, resulting in a *product configuration* in Figure 1). Based on that information, applicable test patterns are automatically selected and their parameters instantiated, resulting in concrete *test cases* that may be exported as test scripts in Gherkin[2] following the *Given-When-Then* structure.

By segregating the process into two distinct stages, firstly with an initial specification component which involves discovering the patterns and defining the feature models; and lastly, a later one which comprises feature selection and product instantiation for test case generation by the platform, this methodology aims to improve the overall speed to market. At its core this process involves a prolonged initial stage of specification and a faster test testing phase, this latter one theoretically is faster mainly due to the fact that the developers and testers will already have the feature models and test patterns defined and only need to instantiate new products of their ecosystem.

B. Domain model

The domain model of Figure 2 represents the main information entities involved in the test process previously described, their relationships, and attributes. It also serves as a basis for designing the data model of the repository of the test platform.

The entities are grouped into four packages, depicted with distinct colors: `FeatureModels`, `TestPatterns`, `Products`, and `TestCases`. Some of the characteristics of features models were inspired by the schema present in a paper by Shatnawi and Cunningham [7].

A feature model is defined by a set of `Feature`s, related by parent/child relationships, starting from the `root`. An example of a visual representation of a concrete feature model can be consulted in Figure 7. Individual child features may be marked as `optional` or `mandatory` (meaning that the child feature must be selected when its parent is selected). The children of a given feature may also be marked as `alternative` (meaning that exactly one child has to be selected when the parent is selected) or as an `or` (meaning that at least one child has to be selected when the parent is selected). Features may also be related with `requires` and `excludes` relationships (meaning that the selection of the source feature implies or impedes, respectively, the selection of the target feature).

In our approach, some features may in fact represent types of product components (i.e., `Sensor` and `Actuator` in Figure 7). Such features are specially marked in the feature model (attribute `isComponent`). This information is relevant when characterizing concrete product components, starting with the selection of the component type in the feature model.

In our approach, features may also have attributes associated, important for test case generation. For example, a sensor may have a logical identifier and a physical locator; a periodic reading sensor may have a time period associated. In Figure 2, each `Attribute` is characterized by a name, data type, and range (in the case of numeric attributes).

Features may have associated `TestPatterns`, in a many-to-many relationship. Each test pattern is described by a sequence of `TestSteps`, which may, in turn, have substeps, in a recursive manner. Each step may be marked as manual or automatic. Test patterns may be parameterized, i.e., test steps may be described in a generic way by referencing parameters of the test pattern. Later on, for concrete test case generation, such parameters need to be instantiated.

`Attributes` are paired to `Parameters` in a many to many relationship; this is due to the platform being conceived as a tool that is able to define various feature models and link those feature models, in a loosely coupled manner, to patterns. This pairing permits the platform to only select test patterns that are actually used by the features. In the end, once the attributes are instantiated at the product level their values

[2] https://cucumber.io/docs/gherkin/

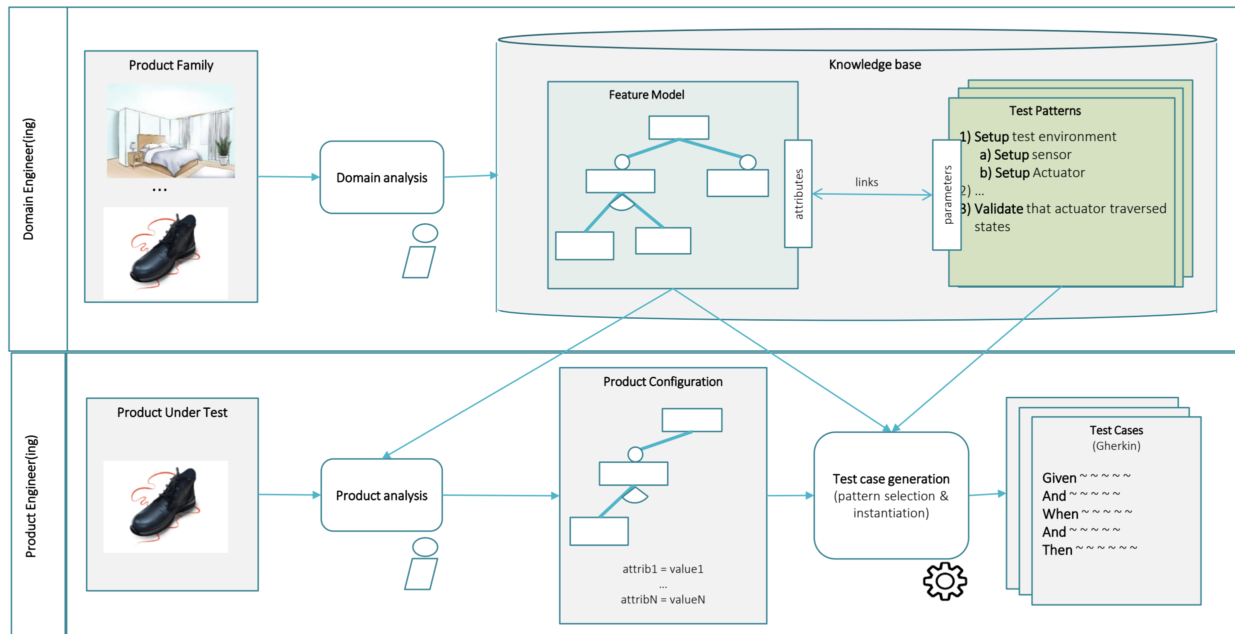

Figure 1. High-level test process (annotated UML activity diagram).

will be automatically assigned to the corresponding parameters through this pairing, and generate test cases.

Whilst manual steps may be described in free text, automatic test steps should be described in a restricted natural language, following appropriate keywords and phrase templates that can be understood by target test automation frameworks that execute test scripts in Gherkin, such as Cucumber[3]. Test steps are also characterized by a type, that will determine the section (*Given*, *When* or *Then*) onto which it is mapped in Gherkin. The currently supported step types are:

- `Setup`: a setup step is normally used to define what parameters are going to be instantiated at the beginning of the test case;
- `Trigger`: a trigger step can be used to indicate if a sensor is going to be triggered;
- `Read`: a reading step is used to read a sensor or actuator value;
- `Validate_equals`: validation step used to check for equality or compliance;
- `For_Each`: this is a special control step to iterate over a list passed as a parameter to the test pattern; in the test generation process, it is expanded into a sequence of steps;
- `During`: this is another control step, to repeat one or more substeps for a configurable amount of time;
- `Check`: a manual validation step; it functions as a checkbox for a tester action;
- `Read_Timestamp`: read a timestamp, to verify response times;
- `Step`: a generic (uninterpreted) step, usually manual.

[3]https://cucumber.io/

In the final conversion of generated test cases into test scripts in Gherkin, the step type determines the section (*Given*, *When*, or *Then*), onto which the test step is mapped. For example, `Setup` steps are typically present at the beginning of the pattern, and are mapped onto the *Given* section in Gherkin (with multiple steps connected by the *And* directive); steps that stimulate or act upon the system under test (such as `Trigger` and `Read`) usually come next, and are mapped onto the *When* section in Gherkin; finally, validation steps are usually done at the end and are mapped onto the *Then* section in Gherkin.

Concrete products under test or product components (class `ProductUnderTest` in Figure 2) are characterized with respect to the feature model by associating the product with the applicable features, and by providing concrete values for the attributes attached to those features.

Based on that characterization, the applicable test cases for a product under test are automatically generated, by selecting and instantiating (for concrete parameter values) the test patterns associated with the selected product features. Each `TestCase` comprises a sequence of `ConcreteTestSteps`, instantiated (and flattened) from the test steps in the corresponding test pattern. In this process, the hierarchical structure of steps and substeps in a test pattern is flattened into a single sequence; control test steps that use the `For_Each` keyword are also expanded into a flat sequence of steps.

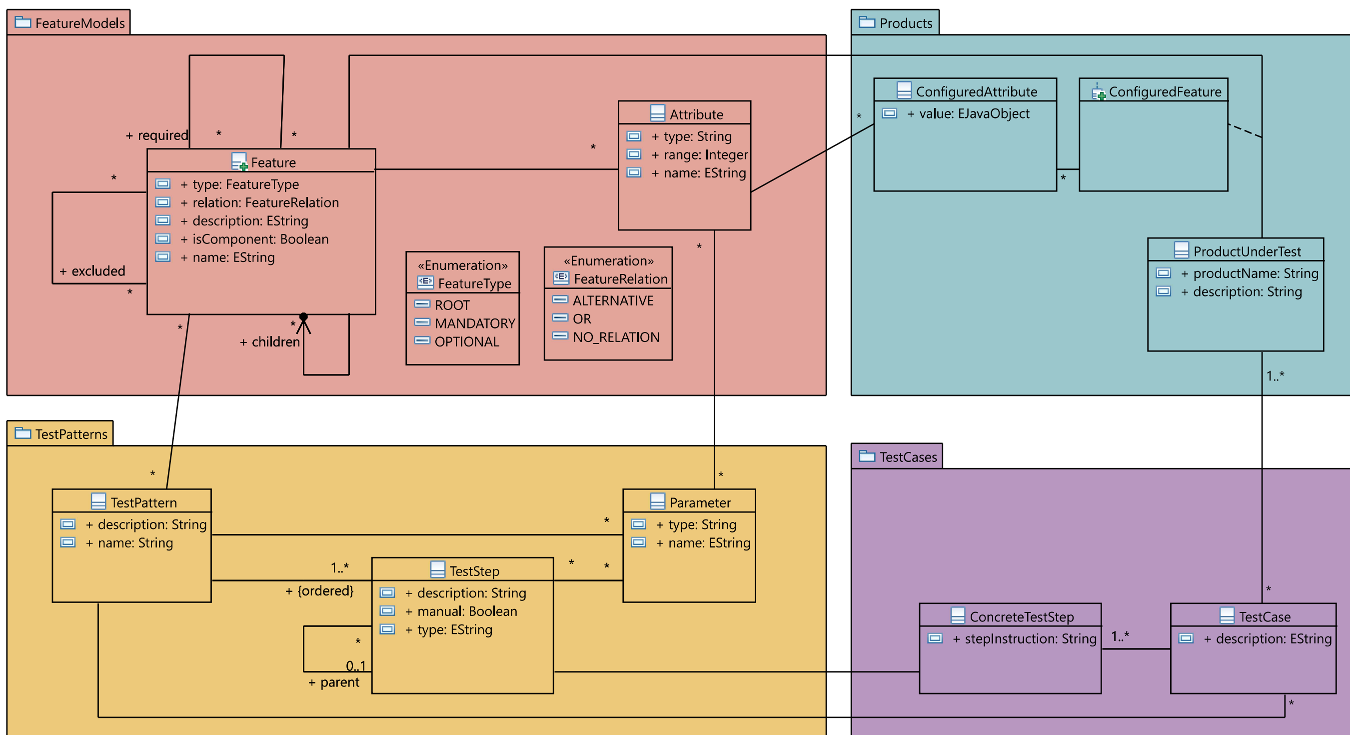

Figure 2. Domain model (UML class diagram with default multiplicity 1).

C. Platform architecture

The test process previously described is supported by a web-based test platform, following a client-server architecture depicted in Figure 3.

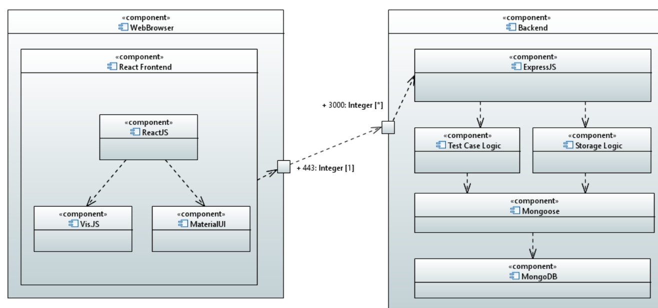

Figure 3. Platform architecture (UML component diagram).

In the frontend, a ReactJS[4] based web page communicates with a RESTFul API developed in ExpressJS[5] in the backend.

To reduce the development effort, two libraries were used: the VisJS[6] library, to draw feature models and build an interactive editor of feature models; the MaterialUI[7], to take advantage of its pre-built UI components, and avoid spending time in manually marking down the CSS of the website.

Finally, to store the data, MongoDB[8] with the Mongoose[9] library were used to store the composite structures; this last decision was made inspired by the work of Shatnawi and Cunningham in [7].

D. Platform user interface

The test platform provides a User Interface (UI) that enables users to perform the different activities of the test process, plus some administrative tasks. The UI is divided into three groups of pages, related to different entities of the domain model: a group of pages for editing and visualizing feature models; another group of pages for editing and visualizing the test patterns; and another group of pages for characterizing products under test and visualizing the generated test cases. The navigation between these pages can be seen in Figure 4.

The page for creating or updating a test pattern is illustrated in Figure 5. The user can enter the global properties of the test pattern (top left), a list of parameters (bottom left, with a name, data type, and description), and a hierarchical list

[4]https://reactjs.org
[5]https://expressjs.com
[6]https://visjs.org
[7]https://mui.com
[8]https://www.mongodb.com
[9]https://mongoosejs.com

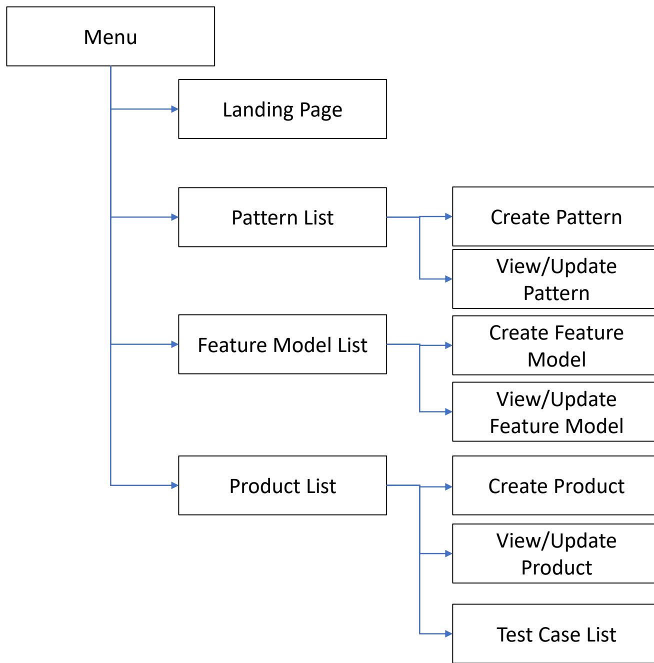

Figure 4. Navigation diagram of the platform.

of steps (right). In the description of the test steps, the test parameters are referenced with the # prefix.

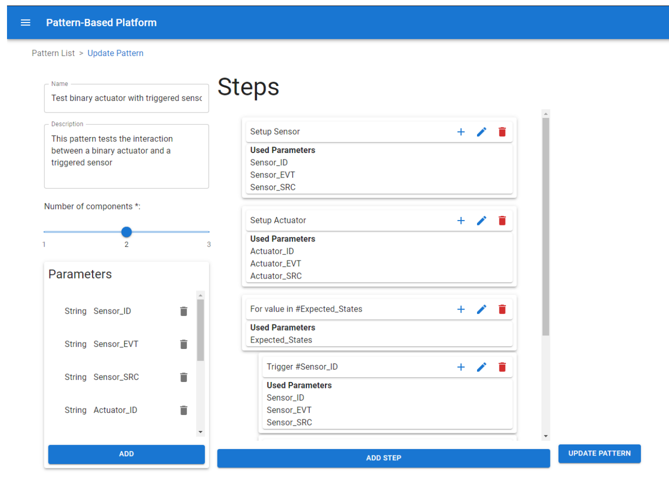

Figure 5. Patterns page UI.

The page for creating or updating a feature model is illustrated in Figure 6. For drawing the feature model, a canvas is used in which features are represented as ellipsis and are connected through straight lines. The user can interact with the page to create or update child features and their attributes, among other operations.

The page for characterizing a product under test is illustrated in Figure 8. The product components are registered and listed in the bottom left, and the features present in each product component are selected from the feature model on

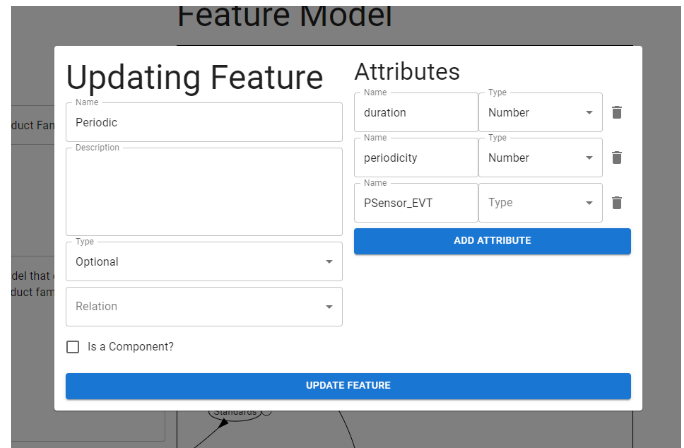

Figure 6. Feature model page UI.

the right. For each selected feature, the user has to instantiate its attributes.

From the list of products under test, the user may request the generation of applicable test cases for a product selected. The test cases are presented in a list with an option to export to Gherkin, as illustrated in Figure 9.

III. Feature Model and Test Patterns

This section describes an excerpt of the feature model and test pattern catalog that were developed for the domain of the SH4ALL project (eHealth and AAL products). The development followed a bottom-up approach, starting from the study of the functionalities and test needs of concrete products from the SH4ALL project, which were iteratively abstracted until a feature model and test patterns were obtained for the underlying domain.

A. Feature model

The process to discover test patterns started with the analysis of the family of products within the scope of the SH4ALL project. These products were analyzed for their functionalities and test needs, based on the products' documentation, interviews with product manufacturers, and domain knowledge. From the documentation, an analysis was made in order to find their functionalities which were registered and attributed to each product and further abstracted until features of a feature model were found and joined together. For example, if we had a periodic temperature sensor we would start with the functionalities of periodic reading and temperature reading; at the end of the process we would have "Sensor", "Reading Mode", "Periodic", "Measure Scale", and "Single" features.

An excerpt of the resulting feature model can be found in Figure 7. It should be noted that the feature model presented in this paper only conveys a few features relevant for demonstrating the definition of related test patterns and the generation of corresponding test cases. A complete feature model for the domain under consideration can possibly have hundreds or thousands of features.

The feature model present in Figure 7 contains the features required for building common IoT systems comprising a set of sensors and actuators that communicate according to relevant standards (such as Bluetooth and HL7 Fhir). Sensors are characterized by a reading mode, which can be either periodic or user-triggered, meaning that readings are produced at regular time intervals or only upon user request, respectively. Sensors are also characterized by the multiplicity of measures produced at each reading (single or multiple). Actuators are characterized by an actuation type, which, in our experiment, can either be binary (with only two states) or linear (in which the actuator's state can go through a range of values).

B. Test patterns

After having the feature model created, the model and its functionalities were further analyzed in order to discover test cases. From this set of test cases, the ones that were common to one or more products and/or are possible to be applied outside the domain of eHealth were abstracted into the test patterns and registered in the catalog below. The pattern catalog was continuously iterated upon and it contains entries for conformance and functional testing of the system's components.

The test patterns are documented following a template suggested by Meszaros and Doble [8] but with slight modifications; each entry's structure is as follows:

- Name: the pattern name which uniquely identifies it;
- Context: the context in which the pattern is to be applied;
- Problem: the problem that the pattern tries to help solve in the context;
- Steps: the sequence of generic steps and sub-steps to achieve the patterns objective; steps can be automatic (marked with the ✿ icon) or manual (marked with the 👐 icon);
- Example: example of product components in which the pattern could be applied.

The next subsections present some of the test patterns defined:

1) Test binary actuator with triggered sensor
2) Test binary actuator with periodic sensor
3) Test linear actuator with triggered sensor
4) Test actuator conformance
5) Test triggered sensor conformance
6) Test periodic sensor conformance

These patterns belong to two different test categories: functional and conformance testing. The first category comprises patterns that involve testing the devices functionalities in an integrated system, such as if an actuator goes to a certain state when a sensor produces a specific reading. The second category is related with testing the conformance of a single device to a given communication standard (such as message formats).

1) Test binary actuator with triggered sensor:

Context: In an IoT system sometimes we have an actuator that only has two states and a sensor that only works as a switch that toggles the states of the actuator.

Problem: We have to guarantee that the sensor and actuator interact appropriately so that the system can fulfill its objectives.

Solution:

1) Prepare system for testing
   a) ✿ Set up location of events for the sensor and the actuator;
   b) ✿ Set up the necessary identifiers for the sensor and the actuator;
   c) Define the actuator expected states to be traversed;
   d) Define the actuator start state;
   e) ✿ Set up the maximum actuator response time;
2) While the actuator state doesn't return to the initial state
   a) Trigger the sensor;
   b) ✿ Verify the trigger time of the sensor and store it;
   c) ✿ Verify the actuator state;
   d) ✿ Verify the actuator's response time and store it;
   e) 👐 Manually register that the actuator was activated;
3) Validate if the actuator only traversed valid states;
4) (if a duration was defined) Calculate the difference between the sensor's trigger time and actuator's response time and verify if it is within the actuator's maximum response time.

Example: This pattern could be applied to a pair of a capacitive sensor and a light bulb. In this case, the sensor is a triggered sensor that simply is touched and turns the light switch on or off.

2) Test binary actuator with periodic sensor:

Context: In an IoT system sometimes we have an actuator that only have two states and a sensor that performs periodic readings and those readings work as a switch that swaps the states of the actuator.

Problem: We have to guarantee that the sensor and actuator interact appropriately so that the system can fulfill its objectives.

Solution:

1) Prepare system for testing
   a) ✿ Set up location of events for the sensor and the actuator;
   b) ✿ Set up the necessary identifiers for the sensor and the actuator;
   c) Define the states and corresponding sensor readings;
   d) ✿ Set up the test duration and reading periodicity;
2) During the test duration at each periodic reading:
   a) Induce a new environment for the sensor;

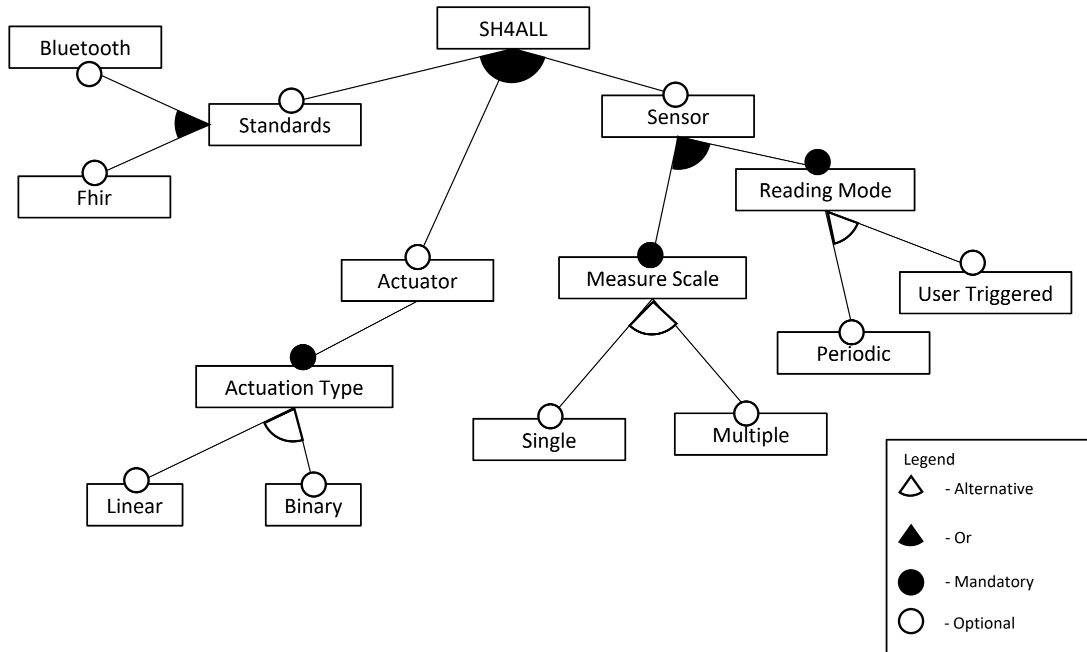

Figure 7. SH4ALL feature model (excerpt).

b) ⚙ Register sensor value and store it;
c) ⚙ Verify the actuator state and store it;
d) ⚙ Register the current time and store it;
e) ✋ Manually register that the system is working as expected;

3) Validate that if for all the sensor readings the actuator was in a valid state;

Example: This pattern could be applied to a heater and a temperature sensor. In this case, the heater is on when the temperature is too low and off when the temperature is high.

3) Test linear actuator with triggered sensor:

Context: In an IoT system sometimes we have an actuator that has a range of possible states and a sensor that is triggered by a user and the value of those readings works as a switch to a certain state of the actuator.

Problem: We have to guarantee that the sensor and actuator interact appropriately so that the system can fulfill its objectives.

Solution:

1) Prepare system for testing
   a) ⚙ Set up location of events for the sensor and the actuator;
   b) ⚙ Set up the necessary identifiers for the sensor and the actuator;
   c) Define the linear actuator's expected value;
   d) Define the linear actuator's start value;
   e) ⚙ Set up the maximum actuator response time;
2) Trigger sensor;
3) ⚙ Verify the trigger time of the sensor and store it;
4) ⚙ Verify the actuator's value;
5) ⚙ Verify the actuator's response time and store it;
6) ✋ Manually register that the actuator was activated;
7) Validate if the actuator was activated to the correct state by a valid value;
8) (if a duration was defined) Calculate the difference between the sensor's trigger time and actuator's response time and verify if it is within the actuator's maximum response time.

Example: This pattern could be applied to a set of buttons that control how much a window's blinds are closed. In this case, each button has a specific length of the blinds assigned.

4) Test actuator conformance:

Context: In an IoT system sometimes its components need to conform to one or more interoperability standards so that they can exchange messages with the system or with external systems.

Problem: We have to guarantee that the actuator only accepts and sends messages according to a certain message schema.

Solution:

1) Prepare system for testing:
   - ✿ Set up location of events for the actuator;
   - ✿ Set up the necessary identifiers for the actuator;
   - ✿ Set up the expected actuator message format;
2) (if actuator cannot be activated manually) Set up sensor:
3) (if sensor set up) Trigger sensor;
4) (if sensor not set up) Activate the actuator;
5) Read actuator message;
6) Validate if the actuator message conforms to the expected actuator message format.

Example: This pattern could be applied to any kind of actuator that has to comply with any given messaging standard that defines a message schema.

5) Test triggered sensor conformance:

Context: In an IoT system sometimes its components need to conform to one or more interoperability standards so that they can exchange messages within the system or with external systems.

Problem: We have to guarantee that the triggered sensor only sends messages according to a certain message schema or interoperability standard.

Solution:

1) Prepare system for testing:
   - ✿ Set up location of events for the sensor;
   - ✿ Set up the necessary identifiers for the sensor;
   - ✿ Set up the expected sensor message format;
2) Trigger the sensor;
3) Read the sensor's message;
4) Validate if the sensor's message conforms to the expected sensor message format.

Example: This pattern could be applied to any kind of triggered sensor that has to comply with any given messaging standard that defines a message schema.

6) Test periodic sensor conformance:

Context: In an IoT system sometimes its components need to conform to one or more interoperability standards so that they can exchange messages with the system or with external systems.

Problem: We have to guarantee that the periodic sensor only sends messages according to a certain message schema.

Solution:

1) Prepare system for testing:
   - ✿ Set up location of events for the sensor;
   - ✿ Set up the necessary identifiers for the sensor;
   - ✿ Set up the expected sensor message format;
   - ✿ Set up the test duration;
   - ✿ Set up the sensor reading periodicity;
2) During the test duration at each periodic reading:
   a) Read the sensor message and store it;
3) Validate if all sensor messages conform to the expected sensor message format.

Example: This pattern could be applied to any kind of periodic sensor that has to comply with any given messaging standard that defines a message schema.

IV. Example of test pattern instantiation

In this section, we present an example of instantiation of a test pattern and derivation of corresponding test cases for a concrete product under test.

The product under test is a lightbulb that is turned on or off by the user by touching a capacitive sensor. Hence, it comprises two devices (components): the capacitive sensor (light switch) and the lightbulb (lamp). An applicable test pattern is the first pattern described in section III "Test binary actuator with triggered sensor".

In terms of the feature model of Figure 7, these components are classified as a "User Triggered Sensor" and a "Binary Actuator", respectively, with the following attribute values (for the attributes defined in the feature model):

- Light Switch:
  - SOURCE_ID: "CAPACITIVE_SENSOR_XYZ"; linked to "Sensor Identifier"
  - DATA_SOURCE_URL: "kafka://192.168.1.54:5000"; linked to "Sensor Event Location"
  - ACTION: "USER TOUCHED"; linked to "Sensor Event".
- Lamp:
  - States to be traversed: "ON;OFF"; linked to "Expected traversed states";
  - DATA_SOURCE_URL: "kafka://192.168.1.54:5000"; linked to "Actuator Event Location";
  - SOURCE_ID: "ACTUATOR_Y"; linked to "Actuator Identifier";
  - START_STATE: "OFF";
  - ACTION: "STATE CHANGED to "; linked to "Actuator Event".

Above, we also indicate the name of the test parameter linked to each attribute in our model.

We next illustrate the instantiation of the iterative step of the test pattern, reproduced below.

1) While the actuator state doesn't return to the initial state
   a) Trigger the sensor;
   b) ✿ Verify the trigger time of the sensor and store it;
   c) ✿ Verify the actuator state;
   d) ✿ Verify the actuator's response time and store it;
   e) ⊔ Manually register that the actuator was activated;

The outer step is an abstract iteration step, for iterating over the expected traversed states (in this case, ON and OFF).

During the loop, the inner actions will be repeated in each iteration.

Step a) inside the loop is a manual or automatic test step in which the real or simulated user triggers the sensor (in this case, the capacitive sensor).

Step b) is an automatic test step. It makes use of the parameters “Sensor Identifier”, “Sensor Event Location” and “Sensor Event” to verify at what time the sensor has been triggered.

Steps c) and d) are also automatic test steps. They make use of the parameters “Actuator Identifier”, “Actuator Event Location” and “Actuator Event” to verify the actuator state and response time and store them for later comparison.

Using the attribute values indicated above for the capacitive sensor and lightbulb, the above loop is instantiated and expanded as follows:

- Trigger the sensor “CAPACITIVE_SENSOR_XYZ”;
- Read timestamp of the sensor at “kafka://192.168.1.54:5000” into “Sensor Timestamps”;
- Read the value of the actuator at “kafka://192.168.1.54:5000”, the event was “STATE CHANGED to ON” into “Actuator Values”;
- Read timestamp of the sensor at “kafka://192.168.1.54:5000” into “Actuator Timestamps”;
- Trigger the sensor “CAPACITIVE_SENSOR_XYZ”;
- Read timestamp of the sensor at “kafka://192.168.1.54:5000” into “Sensor Timestamps”;
- Read the value of the actuator at “kafka://192.168.1.54:5000”, the event was “STATE CHANGED to OFF” into Actuator Values”;
- Read timestamp of the sensor at “kafka://192.168.1.54:5000” into “Actuator Timestamps”.

With our platform, using the pattern in Figure 5 and the model in Figure 6, a product derived from the feature model was created and used to instantiate the pattern. The product was characterized with the attribute values previously mentioned. Taking the previous example of the iteration into account, in Figure 9 it can be seen that the values for the iteration [ON;OFF] present in Figure 8 were used to flatten the loop into a series of Gherkin instructions, giving to the user a test script almost ready for execution.

V. Evaluation

To evaluate the developed platform and test patterns, two validation strategies were used: the first strategy involved end-to-end validation testing by the authors, with some products from the SH4ALL project; the second strategy involved validating the platform with users through the means of a guide and receiving feedback after.

The first validation was done using two products of the SH4ALL project, namely the MyCareShoe and the Bedroom Module and consisted in going through the entire process of:

- Creating the feature model;
- Defining attributes for the features in the feature model;
- Creating the test patterns;

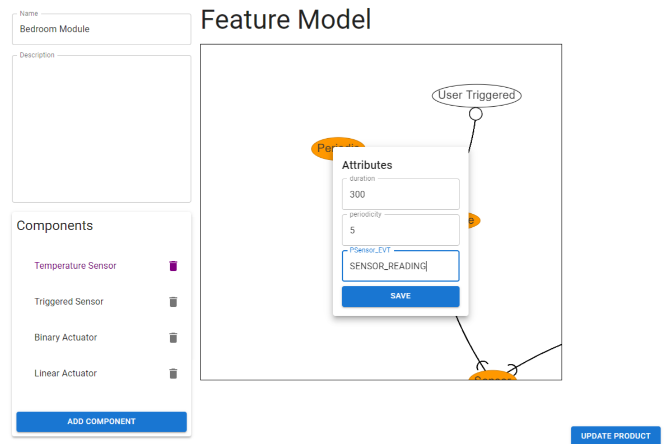

Figure 8. Products page UI.

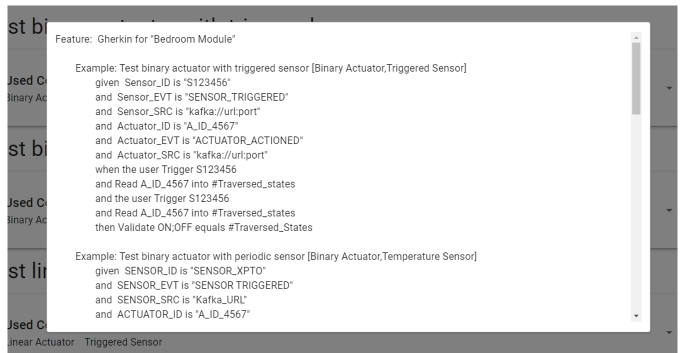

Figure 9. Test cases page UI.

- Linking attributes from the feature model to parameters from the test patterns;
- Characterizing the products by selecting applicable features and instantiating attribute values;
- Generating test cases.

At the end of this first validation process, two metrics were collected: the number of test cases generated and test pattern re-usability. The former is measured by counting the total number of test cases generated, while the latter is measured by viewing if the patterns are reused throughout both products. For example if a pattern is reused in both products it means it was reusable.

The end results were that the platform for the two products generated a total of 16 test cases. The conformance testing patterns ended up being reusable throughout both products due to the fact that, in this experiment, both products had devices requiring conformance testing. Regarding the functional test patterns, for this sample of products, they ended up not being as reusable; this is due to the fact that the products in the end had different functionalities. Bellow, it can be viewed a subset of the generated test cases.

```
Example: Test Binary Actuator With triggered Sensor
    [Led Lights,Capacitive Sensor]
  given Sensor_Identifier is "CAPACITIVE_SENSOR_X"
  and Sensor_EVENT is "Event: Trigger_SENSOR"
  and Actuator_Identifier is "LED_LIGHTS_X"
  and Actuator_EVENT is "Event: Actuator"
  and Start_State is "ON"
  and State_Changes is "OFF;ON"
  when the user Trigger CAPACITIVE_SENSOR_X
  and Read LED_LIGHTS_X into #Traversed_States
  and the user Trigger CAPACITIVE_SENSOR_X
  and Read LED_LIGHTS_X into #Traversed_States
  then Validate #Traversed_States equal OFF;ON

Example: Test binary actuator with periodic reading
    [Led Lights,Temperature Sensor]
  given PSensor_Identifier is "Temperature Sensor"
  and PSensor_Event is "Event: Temp"
  and PActuator_Identifier is "LED_LIGHTS_X"
  and PActuator_Event is "Event: Actuator"
  and Duration is "65"
  and Expected is "undefined"
  and periodicty is 5
  when the loop ends after 65 seconds
  and I Will have Read Temperature Sensor into #
      Metrics 13 times
  and I Will have Read LED_LIGHTS_X into #Metrics 13
       times
  and I Will have Read Timestamp into #Metrics 13
      times
  then Validate #Expected equals #Metrics
```

Listing 1. Example of two text cases generated in the platform.

The second and last validation strategy involved validating the platform with five different users, with a software engineering background from academia and industry. The users received a guide describing the steps that they had to follow in order to generate test cases with the platform. This guide went through the entire aforementioned process, but with only one pattern and a single product. This experiment had the goals of verifying if the platform was usable and to receive feedback.

In the end, the results of this experiment were that the users averaged at around 1 hour and 36 minutes to complete the entire guide, with most of the time being spent in creating patterns and drawing the feature model. The users provided feedback about the platform and suggested some improvements in the UI. The users thought that the UI sometimes wasn't intuitive and suggested changing the location of some elements, the types of some input fields, and the appearance of some buttons. They also suggested displaying some more contextual information.

VI. Related Work

The work related to this paper can mainly be found in three domains: Model-Based Testing (MBT), Software Product Line Testing (SPLT), and Pattern-Based Testing (PBT) applied to eHealth and IoT.

In the domain of MBT, test cases are generated from models of various aspects of the system under test (SUT). The generated test cases can be concrete test cases ready for execution or abstract test cases that require further refinement. MBT approaches applied to the eHealth domain include, for example, the work by Lima and Faria [9], which focuses on certification, Gannous et al. [10], which focuses on testing system attributes, Yu et al. [11], and Bombarda et al. [12], which make use of MBT tecnhiques to model the structure of messages from standards and use those models to perform conformance testing. The main difference with respect to our work is that we generate concrete test cases from test patterns related with recurrent test needs, whereas MBT approaches generate test cases from behavioral models of the SUT. Test generation from models suffers from the test case explosion problem, which is avoided in our approach.

In the domain of SPLT, we have various approaches that follow roughly the same SPLE process. First, are created feature models and associated test generators; these generators can either be models or descriptions in domain-specific languages (DSLs). Then, features and test generators are linked, so that, when features are selected, concrete test cases are generated for those selections. In this domain, we can mention the works by Wang et al. [13], Bucaioni et al. [14], Lity et al. [15], Gebizli and Sözer [16], and Weißleder and Lackner [17]. Comparing those approaches to ours, the main difference is that our generators aren't models or DSLs but instead test patterns, with the advantage already mentioned of avoiding the test case explosion problem.

Finally, in the domain of pattern-based testing for IoT systems, we found the works of Pontes et al. [18] [19] and Cunha et al. [20]. In a first iteration, Pontes et al. developed a framework for pattern-based testing called Izinto and cataloged five initial test patterns for IoT. Later, Cunha et al. took Izinto as a base and created a UI for that framework to simplify testing for less technical users. Our work builds upon the pattern catalog of Pontes et al. by contributing with six test patterns to the literature. Our current work also followed a similar approach to Pontes et al. and Cunha et al. by constructing a pattern-based test platform that will aid in testing IoT devices, with the main difference being that in their work the test patterns are hard coded in the platform as parameterized test methods written in an extension to JUnit, whereas in our approach the test patterns are defined by a domain expert in a textual language and are transformed to test scripts in Gherkin to run on concrete products under test.

VII. Conclusions and Future Work

This paper contributed with a novel pattern-based test process and support platform, designed to facilitate the generation of integration and system tests for families of products, following a software product line approach to promote reuse and consistency.

In the proposed approach, a feature model and a set of recurrent test patterns related to the features in the feature model are initially defined by a domain specialist. Then, for each product under test, the product specialist only has to characterize the product under test with respect to the defined feature model. Based on the product characterization and the catalog of test patterns, concrete test cases are derived automatically (by a selection and instantiation process from

the test patterns) and exported as test scripts in Gherkin ready for execution.

The approach was illustrated for a family of smart health products, within the scope of the SH4ALL project. Six test patterns were presented, three in the domain of functional testing and the remaining in the domain of conformance testing. Based on these test patterns, twelve test cases were automatically generated for two products of the product family.

In future work, we plan to extend the catalog of test patterns and validate the approach with further products of the SH4ALL project.

ACKNOWLEDGMENTS

This work is a result of project SMART-HEALTH-4-ALL - Smart medical technologies for better health and care, with reference POCI-01-0247-FEDER-046115, co-funded by the European Regional Development Fund (ERDF), through the Operational Programme for Competitiveness and Internationalization (COMPETE 2020) and the Lisbon Regional Operational Programme (LISBOA 2020), under the PORTUGAL 2020 Partnership Agreement.

REFERENCES

[1] A. S. Yeole and D. R. Kalbande, “Use of internet of things (iot) in healthcare: A survey,” in *Proceedings of the ACM Symposium on Women in Research 2016*, ser. WIR ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 71–76. [Online]. Available: https://doi.org/10.1145/2909067.2909079

[2] R. H. Ekpo, I. P. Osamor, V. C. Osamor, T. N. Abiodun, A. O. Omoremi, M. O. Odim, and O. Oladipo, “Understanding e-health application utilizing internet of things (iot) technologies,” in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2019, pp. 1465–1469.

[3] C. Software, “The role of software in medical device failures,” Available at https://www.criticalsoftware.com/multimedia/critical/pt/xvhdnSb3o-CSW_White_Paper_The_Role_of_Software_in_Medical_Device_Failures.pdf, Accessed last time in October 2021, 2021.

[4] I. O. for Standardization, “Medical devices — Quality management systems — Requirements for regulatory purposes,” Geneva, CH, Standard, Mar. 2016.

[5] ——, “Health software – Software life cycle processes,” Geneva, CH, Standard, Mar. 2016.

[6] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: Foundations, principles, and techniques*, ser. Software Product Line Engineering: Foundations, Principles, and Techniques, 2005, pp. 1–467. [Online]. Available: www.scopus.com

[7] H. Shatnawi and H. C. Cunningham, “Encoding feature models using mainstream json technologies,” in *Proceedings of the 2021 ACM Southeast Conference*, ser. ACM SE ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 146–153. [Online]. Available: https://doi.org/10.1145/3409334.3452048

[8] G. Meszaros and J. Doble, *A Pattern Language for Pattern Writing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997, p. 529–574.

[9] B. Lima and J. P. Faria, “A model-based approach for product testing and certification in digital ecosystems,” in *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2016*, 2016, pp. 199–208, cited By :1. [Online]. Available: www.scopus.com

[10] A. Gannous, A. Andrews, and L. Alhazzaa, “Robustness Testing of Safety-critical Systems: A Portable Insulin Pump Application,” in *Proceedings - 2020 International Conference on Computational Science and Computational Intelligence, CSCI 2020*, 2020, pp. 1736–1742. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85113388343&doi=10.1109%2FCSCI51800.2020.00322&partnerID=40&md5=cbcaec1545c42ca3720c133a5ec8d2b9

[11] L. Yu, Y. Lei, R. N. Kacker, D. R. Kuhn, R. D. Sriram, and K. Brady, “A General Conformance Testing Framework for IEEE 11073 PHD’s Communication Model,” in *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments*, ser. PETRA ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2504335.2504347

[12] A. Bombarda, S. Bonfanti, A. Gargantini, M. Radavelli, F. Duan, and Y. Lei, “Combining Model Refinement and Test Generation for Conformance Testing of the IEEE PHD Protocol Using Abstract State Machines,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11812 LNCS, pp. 67–85, 2019. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85075672459&doi=10.1007%2F978-3-030-31280-0_5&partnerID=40&md5=7213798343659fe8b695a7378240b46d

[13] S. Wang, S. Ali, T. Yue, and M. Liaaen, “Using feature model to support model-based testing of product lines: An industrial case study,” in *2013 13th International Conference on Quality Software*, 2013, pp. 75–84.

[14] A. Bucaioni, F. D. Silvestro, I. Singh, M. Saadatmand, H. Muccini, and T. Jochumsson, “Model-based automation of test script generation across product variants: a railway perspective,” in *2nd ACM/IEEE International Conference on Automation of Software Test*, May 2021. [Online]. Available: http://www.es.mdh.se/publications/6219-

[15] S. Lity, M. Lochau, I. Schaefer, and U. Goltz, “Delta-oriented model-based spl regression testing,” in *2012 Third International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*. IEEE, 2012, pp. 53–56.

[16] C. S. Gebizli and H. Sözer, “Model-based software product line testing by coupling feature models with hierarchical markov chain usage models,” in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2016, pp. 278–283.

[17] S. Weißleder and H. Lackner, “Top-down and bottom-up approach for model-based testing of product lines,” *arXiv preprint arXiv:1303.1011*, 2013.

[18] P. M. Pontes, B. Lima, and J. P. Faria, “Test patterns for IoT,” in *A-TEST 2018 - Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Co-located with FSE 2018*, 2018, pp. 63–66. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85061771105&doi=10.1145%2F3278186.3278196&partnerID=40&md5=518106579f24e48c32f07c922e18ab85

[19] ——, “Izinto: A pattern-based IoT testing framework,” in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 125–131. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85060969368&doi=10.1145%2F3236454.3236511&partnerID=40&md5=81f32ef3b2e3880d31931055943032a1

[20] H. D. Q. Cunha, “Low-code solution for iot testing,” 2019.