TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211X

TUD-FI14-02-April 2014

Püschel, Seidl, Schlegel, Aßmann

Institut für Software- und Multimediatechnik

Using Variability Management in Mobile Application Test Modeling

Technische Universität Dresden
Fakultät Informatik
D–01062 Dresden
Germany
URL: http://www.inf.tu–dresden.de/

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Using Variability Management in Mobile Application Test Modeling

Georg Püschel[1] , Christoph Seidl[1], Thomas Schlegel[2], Uwe Aßmann[1]
[1]Software Technology Group
[2]Software Engineering of Ubiquitous Systems Group
Technische Universität Dresden
{georg.pueschel, christoph.seidl, thomas.schlegel, uwe.assmann}@tu-dresden.de

Abstract: Mobile applications are developed to run on fast-evolving platforms, such as Android or iOS. Respective mobile devices are heterogeneous concerning hardware (e.g., sensors, displays, communication interfaces) and software, especially operating system functions. Software vendors cope with platform evolution and various hardware configurations by abstracting from these variable assets. However, they cannot be sure about their assumptions on the inner conformance of all device parts and that the application runs reliably on each of them—in consequence, comprehensive testing is required. Thereby, in testing, variability becomes tedious due to the large number of test cases required to validate behavior on all possible device configurations. In this paper, we provide remedy to this problem by combining model-based testing with variability concepts from Software Product Line engineering. For this purpose, we use feature-based test modeling to generate test cases from variable operational models for individual application configurations and versions. Furthermore, we illustrate our concepts using the commercial mobile application "runtastic" as example application.

1 Introduction

Smart phones and tablet PCs are a vast growing market where the IT industry brings on new devices and applications (via Apple Store, Google Play, etc.) day by day. Operating systems (OS) are the central differentiating factor of platforms for these applications. Thus, on the one hand, an OS establishes a rather homogeneous environment but, on the other hand the fast evolution has a contrary effect. Each device forms a concrete platform assembled of both a versioned software and a fixed hardware environment. Thus, mobile applications have to support a growing number of platforms and platform versions. Hence, the task of managing variability becomes crucial. In consequence, not only in application development but in test management as well as in test automation engineers have to deal with this variability.

Although the operating system provides abstraction from the underlying platform, the system functions are black boxes and their reliability is not guaranteed. For instance, presentation differs with screen resolution and version of the widget library; context-aware functions depend on availability of specific sensors (e.g., GPS). Hence, platform variability has significant impact on the system under test (SUT). Various properties must be considered

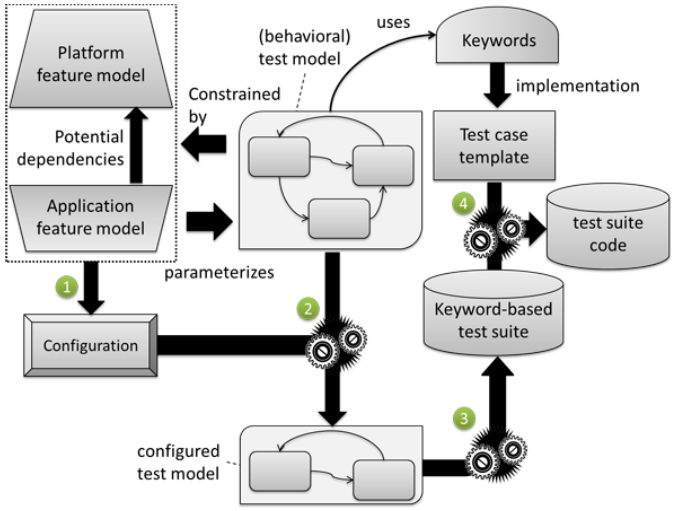


Figure 1: Overview of the workflow of the presented approach.

when specifying test cases and later selecting devices for test execution. Variability has to be covered adequately such that testers can assure a sufficient quality.

For this problem, in the context of *Software Product Lines* (SPLs), concepts and approaches were found to manage variability, create configurations, derive individual software products and test them. SPLs consist of shared assets and differing ones. The commonalities are subject to a *domain analysis* and the variable assets are subject to an *application* analysis. The same scheme is applied in SPL testing (SPLT, [McG07]) where domain and application testing must be distinguished. Domain testing comprises the validation of common features and application testing refines it for the concrete products.

In SPL engineering, models (e.g., *feature models* [KCH$^+$90]) are used to express an SPL's variability. The advantage of this approach is that concrete instances (products) of the SPL can automatically be derived or checked against the model's constraints. Testing also benefits from using models: in *model-based testing* (MBT, [Utt07]), test cases can automatically be derived from operational test models such as state or activity charts. MBT's advantages are *traceability* between generated test artifacts and requirement specifications as well as the higher coverage, which can be reached in combination with test automation. MBT is a black box approach where no code is directly investigated and models do not specify the SUT's inner properties but operations and a protocol of its interface. Hence, variability management in MBT means tailoring the black-box-based models according to configuration decisions on the platform's or application's variability model. The result are test models that can be automatically generated for different configurations.

Altogether, an industrial application of these methods and techniques is desirable in order to improve and automate testing of mobile applications. Therefore, in this paper, we present our test modeling tool [anonymized for blind review] , which aims to face these challenges. Its modeling and processing workflow is illustrated in Figure 1. We start by defining

feature models for the platform and—optionally—for the mobile application as well. The platform's model comprises variability in hardware and operating system capabilities while the application's model specifies the difference between certain versions (e.g., Lite and Pro). The application's variability sometimes depends on the properties of a concrete platform (e.g., available sensors or API methods). The central test model—in our approach a Petri net—supports two variability mechanisms: For one, using constraints, its operational elements can be tailored to be included only for a subset of possible platform/application configurations. Furthermore, a parametric variability mechanism allows to generate configuration-specific test actions. The latter mechanism is based on keywords, which can be mapped to concrete unit test code by using a template. Before the generation process starts, the user has to select a concrete configuration (step 1) that includes the definition of the target platform and the application product. In step 2, this configuration determines which elements of the test model are considered in the generation step 3. In this step, the remaining test model is subject to a reachability analysis that computes all possible execution traces consisting of keywords. Later, in step 4, these keywords are processed by the template-based model-to-text transformation mechanism, which produces code artifacts (e.g., unit tests).

With our process, we are able to produce test cases for multiple mobile platform configurations from one central test model. In this paper, we describe this process in detail and illustrate it in the context of a running example based on the mobile application "runtastic". Thereby, we present the necessary metamodels, notations and how they are applied for the running example.

The remainder of this paper is structured as follows: In Section 2, we introduce a running example. In Section 3, we describe our variability metamodel and an instance for the Android platform. Subsequently, we present the test models in Section 4 and discuss in detail how test cases are derived. In Section 5, we discuss related work before we conclude the paper and outline future work in Section 6.

2 Running Example

In this section, we briefly introduce our running example. We selected a proprietary third-party Android application called "runtastic"—a successful fitness application that was installed more than 52000 times in lite and almost 20000 times in pro version according to Google Play Store statistics[1]. The application's services depend on the available sensor equipment as well as the version the user ordered.

Figure 2 depicts initial and main screens of runtastic. The application allows users to track and evaluate their running accomplishments comprising functionality like time, track, and heart rate recording, evaluation diagrams, track planning and sharing. In addition to the features of the `Lite` version, the `Pro` version of the application adds functionality like voice support among the complete training phase. Our goal is to automatically generate black box test cases for this running example. Black box testing means that we do not

[1] http://play.google.com

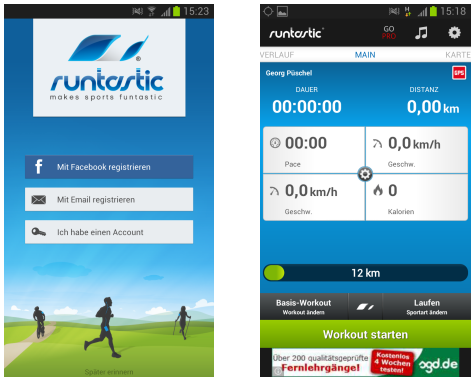


Figure 2: Screenshots of the case study application “runtastic”.

require any access to the application’s source code. Test actions are solely performed on the graphical user interface and validations are performed against it as well.

3 Platform Variability Modeling using Attributed Features

The presented example is implemented for several platforms and platform versions. For testers, these environments have different effects on the SUT due to different variable properties. For instance, the availability of sensors can affect the application’s functionality. In our running example, the recording of running actions fails on devices without GPS. Another example is the layout of the SUT’s graphical user interface, which has to be adapted to different screen sizes depending on the device the application runs on.

All these differences must be taken into account when specifying test cases or test models that are used to generate test cases. When aiming at generation of test cases for individual configurations, the first step is to specify all necessary variable properties of the platform. In SPL research, these variable properties are captured in a variability model, such as a feature model [KCH$^+$90]. This approach benefits from the existence of graphical notations, several extensions (e.g., feature attributes, different cardinality elements, constraints) and the abstraction from implementation details (i.e., the variability mechanism). In this section, we present our problem-specific feature model that enables us to express the platform and application variability.

3.1 Attributed Version-aware Feature Model

The initial base model and notation for feature models was proposed by Kang et al. [KCH$^+$90]. A feature tree is basically a compound tree where nodes are features that can be optional or mandatory in context of their parent feature. In [CHE05], Czarnecki et

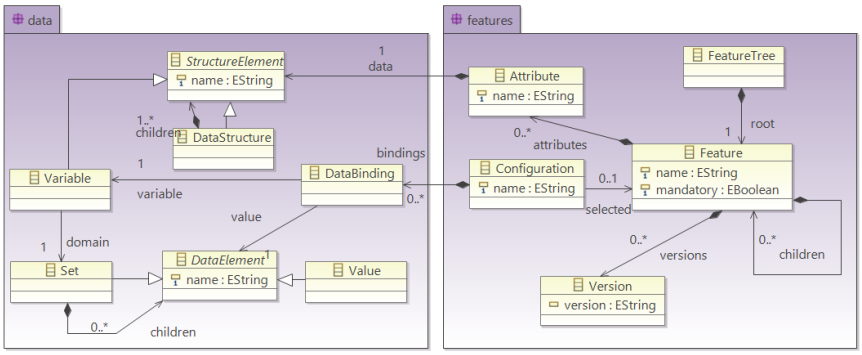


Figure 3: Metamodel for attributed version-aware cardinality-based feature models.

al. added elements to express attributes and additional cardinality constraints for features.

Our metamodel relies on attributes of features due to the need to specifiy the different variants of numeric properties in the platform environment (e.g., display resolution). Furthermore, we take the continuous evolution of platforms into account by introducing an element to specify the version of a feature.

We provide a formalization for feature models with these capabilities using the metamodel presented in Figure 3. It is separated into two packages: *data* for attribute data structures and values as well as *features* for the feature metamodel itself. The latter one starts with the definition of a `FeatureTree`, which has exactly one root `Feature`. A feature has a `name` and specifies whether it is considered `mandatory` in context of its parent feature. A feature may further possess `children` features. A `version` of a feature allows the definition of functional evolution in time. When selecting a feature for configuration, exactly one version of that feature has to be selected as well. In the case that there is only a single version of a feature, it does not have to be modeled explicitly so that the feature alone may be selected.

As we use attributed feature models, we introduce elements for data structures describing the domain of possible values for a particular attribute. `DataStructures` have a `name` and contain one or multiple `StructureElements`. Valid sub-elements are `DataStructure` itself (to build up a composite tree) and `Variable` for leaf elements. A second composite structure can be built up for value `Sets`. The abstract value type is `DataElement` and has a `name`. Concrete leaf values are represented by instances of `Value`. Sets containing sets implement a test specific requirement: while the root set represents a value domain, sub sets represent *equivalence classes* such that the test modeler is able to abstract from certain value ranges.

Connection between both packages appear on type and instance level. Firstly, a `Feature` can contain multiple named instances of `Attributes` which are association classes each pointing to a `StructureElement`. Secondly, the tester uses `Configurations` to build up concrete products where all variability is resolved. Thus, a configuration refers to a `selection` of features and contains `DataBindings` to bind the concrete values to its

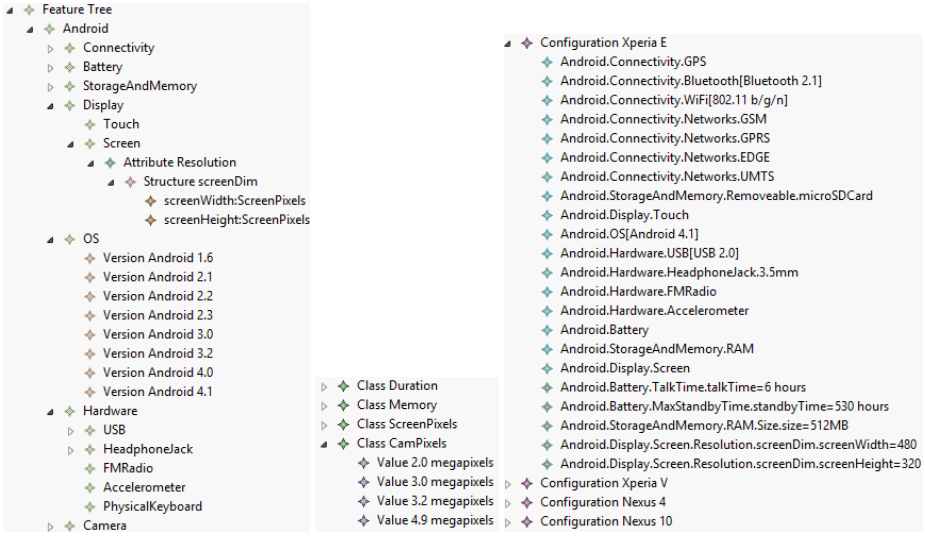


Figure 4: Feature model for the Android platform including concrete configurations of devices.

features' attribute variables. With these classes, we can now specify a real world platform landscape of environments in which the SUT may be tested.

3.2 Feature Model of the Android Platform

Android is Google's operating system for mobile devices. While runtastic was also implemented for a variety of other operating systems, our investigation focuses on Android due to its broad support by a large number of devices. At the time of writing, the maximum version of Android on our devices is 4.1 and there is a large number of devices that were not and will not be updated to this release. Thus, several past versions must be considered in testing to assure that the application runs reliably on these "legacy" systems.

Figure 4 depicts excerpts of a feature model capturing the variability for the Android platform along with concrete configurations of individual devices. Top level features of the platform are `Connectivity`, `Battery`, `StorageAndMemory`, `Display`, `OS` (operating system), `Camera`, and `Hardware` (e.g., `USB`, `FMRadio`). Several features are versioned (e.g., in Figure 4 the OS) and attributed (e.g., `Resolution`). For attributes, only four value domains were required—`Duration` (3 - 2180 hours), `Memory` (0 - 256000 MB), `ScreenPixels` (240 - 1920 pixels), and `CamPixels` (2.0 - 12.19 megapixels). We refrained from specifying different domains for attributes with equal dimensions even if the employed domains may contain excess values never used in practical scenarios.

Additionally, the bottom part of Figure 4 depicts an excerpt of platform configurations used by individual devices. The unfolded configuration for the `Xperia E` smartphone contains several features and data bindings (e.g., RAM size is 2048MB).

Due to the large number of features and configurations, we built the variability model for Android automatically by parsing the information on Google's website[2]. Due to this automation, we do not need to alter the model manually if new features or devices appear in order for the information to stay up to date. The model depicted in Figure 4 was retrieved on 02/13/2013.

Note that there are several other feature groupings possible but we decided to organize the feature tree exactly as Google organized its feature configuration. Furthermore, we did not include the CPU in our feature tree because there are too many variants of processors to gain an advantage from using the concrete CPU as discrimination parameter in testing. Several useful limitations may be specified to consider the compatibility definitions of Google for each Android version[3]. To reduce complexity, we neglect this issue for the platform environment in our example. However, such variablity constraints can be constructed, e.g., with propositional logic (cf. Section 4.2).

The overall model contains 36 abstract and concrete (leaf) features, including 16 versions and 7 attributes. Even though this is a rather manageable feature tree, 278 distinct platform configurations were specified on this basis. Hence, the usage of features, attributes, and versions in variability management results in a significant complexity reduction in comparison to mapping each test artifact to concrete configurations. When test cases are generated, the modeler selects a subset of platforms (i.e., concrete devices), instead of creating a new configuration from the feature tree. The utilization of the constructed platform variability model for our running example is presented in the next section.

4 Feature-based Test Modeling

The challenge for testers is to determine whether a specified input produces an expected output. The subject of testing depends on the integration level: single components are tested with unit tests, multiple components require integration testing and, in the end, the fully-integrated system has to be checked in a system test. Due to the the black box perspective employed by MBT, this approach can deal with all these test procedures as single components as well as combinations of multiple components can be perceived as black boxes. A black box provides an interface with several parameterized operations and a protocol definition. The behavior defined by this protocol has to be tested and, thus, needs to be described in an operational model. Furthermore, we can use the aforementioned feature model notation to specify the behavioral consequences implied by different application configurations and test environments (i.e., platform configurations).

In the following, we discuss how the business-related variability that application developers intended can be integrated into our approach by using an extra application feature model. After that, we introduce the operational model employed in our approach as well as the concept to map features to platform functionality.

[2] http://www.android.com/devices
[3] http://source.android.com/compatibility

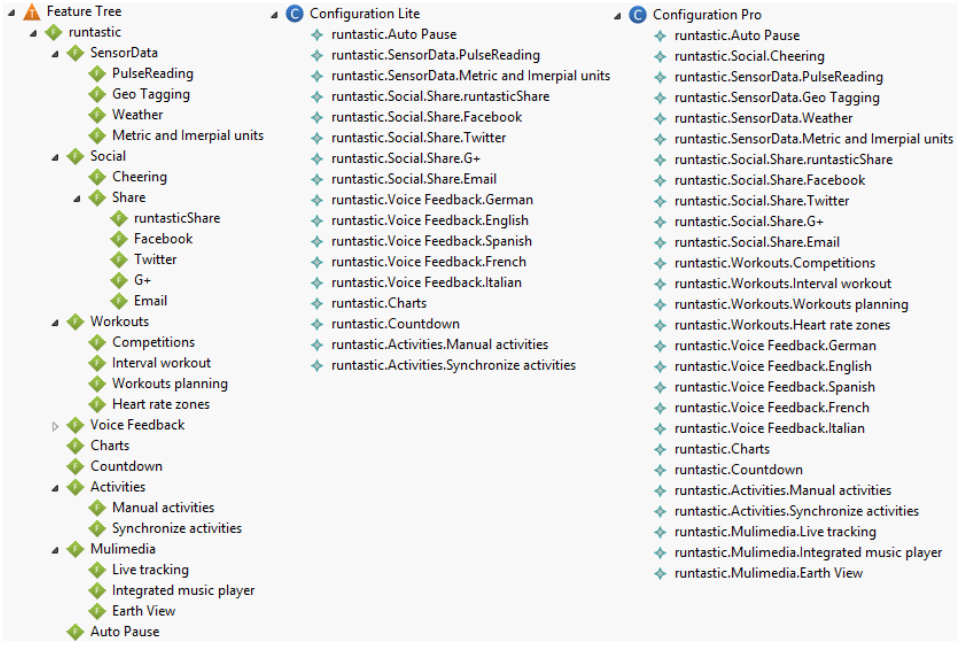


Figure 5: Excerpt from the example application feature model and configurations.

4.1 Application Feature Model

Mobile applications are often released in at least two versions: a `Lite` version and a commercial `Pro` version. Naturally, the latter one provides extended functionality. In case of our running example, e.g., heart rate measuring is only accessible in “runtastic pro”. This type of variability in functionality is mostly due to business considerations such that different groups of users are attracted and to provide an impression of the functionality of an app without having to pay. We can specify the variable capabilities of the SUT inside the feature tree and re-organize it according to a changed business strategy or for newly added features. We then use the features of each version as determination parameters for our test model instead of a singular version distinction.

For our running example, the application feature tree and the two configurations are depicted in Figure 5. As a configuration only consists of statements on selected features and bound variable values, each application configuration can be combined with a platform configuration by simply building a set-wise union of both statements. Thus, the tester can select one of each type before generating a test suite for a specific application version and platform target. Such application feature models are optional and only necessary if we distinguish between multiple business-driven versions.

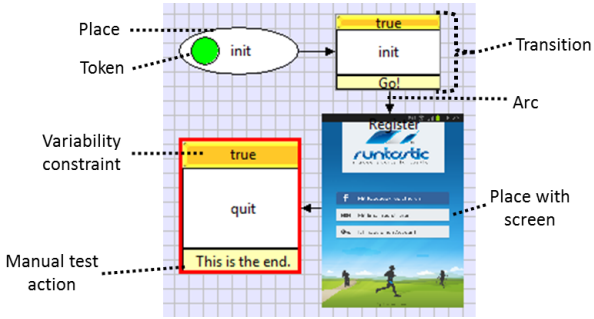


Figure 6: Minimalistic sample model of our Petri-net-based operational model illustrating syntactic elements.

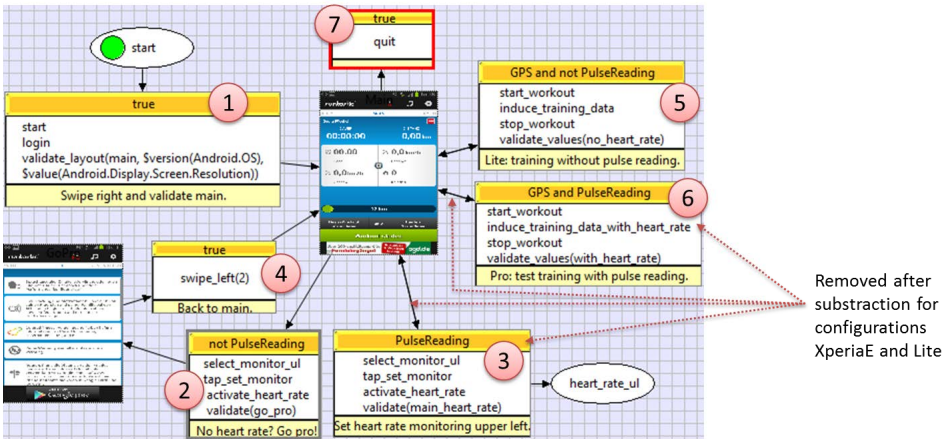


Figure 7: Excerpt of the variable control flow model for the example application “runtastic”.

4.2 Operational Metamodel and Parameterization

As discussed before, an operational model is required to define the black box interface’s behavior. Several types of models exist that potentially can be employed for this purpose: state machines or UML activity diagrams, state charts, or sequence charts provide means for behavior modeling. However, some of these models lack means for parallelism and others are syntactically complex and lack strict semantics. Furthermore, none of the notations includes variability mechanisms that would enable them to be configured by our feature models (cf. Figure 1, step 1). In consequence, we use an alternative formalism based on Petri nets to model behavior in our approach. In order to relate the feature model and the SUT’s behavior, we extended Petri nets by two different variability mechanisms:

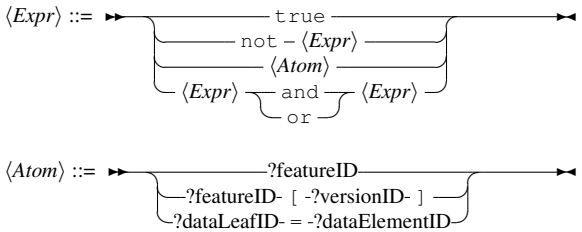
1. By attaching contraints over features on elements in the test model, these elements’ availability gets dependent on the selected configuration. If the constraint is not satisfiable in this configuration, the respective elements are “subtracted” from the

model. This extension is based on Muschevici et al.'s Feature Petri Nets [MCP10].

2. Elements in the operational model are parameterized with references into the feature model such that selection decisions, version names, and attribute values later are appearing in the generated test cases (template mechanism).

Figure 6 depicts a minimalistic sample model instance illustrating all possible elements. The elliptic entity represents a *place* and is marked with an initial *token* (green) as known from conventional Petri nets. The label "init" is the name of the place and has no impact on its semantics. The *arc* to the right connects the place with a *transition*. In Petri nets, source and target of an arc must be of different types, either transition or place. We extended the basic Petri net syntax and semantics of transitions in our formalism. The transition and its representation is separated into three compartments:

a) The upper compartment contains a *variability constraint* in the form of propositional logic over the set of features, versions, and attribute values. The "true" symbol states the sample transition to be unconstrained (despite the control flow arc). The exact syntax is defined in the following syntax diagram:

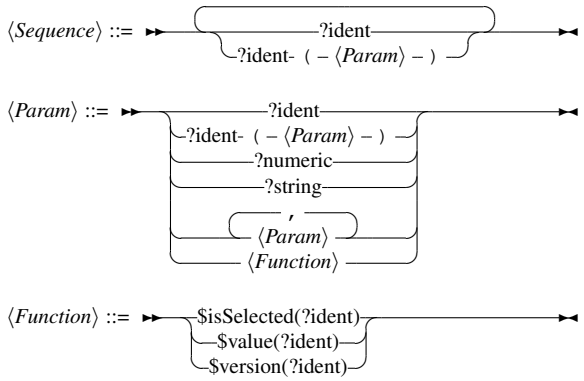


The start symbol $\langle Expr \rangle$ lets us express and/or/not propositional logic with the intuitive semantics and operator priorities. The atoms are feature references (true if feature is selected) either with or without a version, and the atom for comparison of an attribute variable value with a literal value (as reference to a value defined in the value sets/equivalence classes, cf. Section 3). An example expression that is satisfiable on `Xperia E` (cf. Figure 4) is:

```
Android[Android 4.1] and talkTime = 6hours
```

The variability constraint implements the first subtractive variability mechanism: If a constraint cannot be satisfied in the current configuration, the respective transition is removed along with the remaining dangling arcs at test case generation.

b) The middle compartment contains a sequence of parameterized keywords. Keywords are used in automating test execution to abstract from the actual potentially complex execution of single test actions. For instance, a keyword "clickOnLoginBtn" can be used instead of detailed code for the button localization and widget selection directly into the models. Instead, these details are later specified in a separate definition (cf. Figure 1, step 4). Further advantages of this approach are the possibility to reuse the keywords inside the behavior model or rewriting their implementation and their potential platform-dependent semantics without changing the test model. The keywords' syntax is defined as follows:



Our syntax additionally supports several arguments to parameterize the keywords. There can be multiple comma-separated arguments. The first type of arguments are literals (identifiers, numerics, or strings). Moreover, the modeler can use functions to query the currently selected feature configuration. These functions are templates and are substituted at generation time by the generator. Thus, $\$isSelected(\ldots)$ is replaced with $true$ or $false$ depending on whether the referenced feature is selected. Furthermore, $\$version(\ldots)$ and $\$value(\ldots)$ are replaced with the current configuration's included version name of a feature or its value for a feature attribute variable. An example expression for this compartment is:

```
start
validate_widgets(start_screen,$value(screenDim)
```

This expression consists of two keywords: `start` indicates a test action that starts the SUT and `validate_widgets` must later be mapped to an evaluation procedure which checks to layout of the start screen under the given screen dimension.

c) The lowest compartment can be used to enter free text statements. The modeler can use them to either add human-readable notes to the model or to specify manually executable test actions besides the keyword-based ones. Manual actions are required if necessary input or manipulations cannot be automated completely and human interaction is required.

In the lower right of Figure 6, another arc connects this transition to a screenshot-represented place element. This place has semantics completely equivalent to the elliptic representation. The screen-based place was added to provide orientation for test modelers when creating or maintaining the model. In contrast, the elliptic place represents "partial states" that cannot be directly associated with screens. The transition in the lower left of Figure 6 is framed by a red border. It states that a behavioral sequence that leads to this transition generates a test case with each execution of this transition. This element is required because Petri nets do not have a terminal state (as state machines do) and the generator has to determine if the end of a produced trace can be identified as an actual termination.

The operational semantics of our extended Petri net model is defined as follows: A transition is activated if all places that are connected over incoming arcs (i.e., input places) are marked with at least one token. Activated transitions can be executed by consuming one token from each input place and producing one token into each place connected to the transition via an outgoing arc (i.e., output place). If multiple input places are connected to a transition, one token of each input place has to be consumed. At each execution step, a single activated

transition is executed and its keywords are appended to the test case (after performing the keywords' template substitutions). As we require to generate multiple test cases, the generation process is identical to a reachability analysis.

4.2.1 Excerpt from Running Example Model

Figure 7 illustrates how the introduced syntax is applied in our running example. Each transition is numbered and explained in the following:

1) The initial transition is included in arbitrary configurations. Hence, its variability constraint is set to `true`. The keyword sequence starts with `start` and `login` followed by a `validate_layout` action with three arguments: `main` indicates the validated form, `$version` is replaced at generation time with the operating system version and the `value` argument by the configuration's screen resolution. The latter keyword is parameterized and makes the correct layout for the concrete targeted device available to test automation.

2/3) The main screen shows four monitoring sections (upper/lower left/right). For each, the user can select the monitored quantity (e.g., time, pace, heart rate). At first, the user has to select one of the monitoring sections (in the example case, the upper left one [in some keyword abbreviated `ul`]). After tapping on the set monitor button, a list of available quantities appears. Depending on the configuration, the application sets the monitor to the selected quantity or forwards the user to the "Go Pro" screen where features are listed that are only available in the Pro version. This behavior is modeled by both transitions (2) and (3), which are available or not depending on feature `PulseReading`. In case of transition (3), the `validate` keyword checks if the GUI shows the correct form with the new heart rate monitor available. Afterwards, there are tokens in the main screen place and in `heart_rate_ul`. The latter place indicates a partial state in the test model to mark that the heart rate monitor is set to the upper left section.

4) To move back to main screen, the keyword `swipe_left` is parameterized with the numeric argument 2 indicating this action to be executed twice.

5/6) The last two transitions illustrate how a training recording is to be modeled. The (non-manual) recording can only be performed if GPS is available. If GPS is not available, runtastic issues a warning to the user, which is not depicted in this excerpt. Again, depending on feature `PulseReading`, the transitions behave differently. At the beginning of recording, the user taps on `start_workout`. Then, training data from sensors (GPS, heart rate, time etc.) is induced. In test execution automation, this has to be done automatically (by manipulating the device using mock data) or by carrying the device on a physical test track. Subsequently, the workout is stopped and measured values are validated.

7) A `quit` action is executed to terminate the test (indicated by the red border).

As this example shows, both our variability mechanisms can play different roles in test models. Using this model, test cases can be generated. The remainder of the section discusses how keyword-based test cases are produced and afterwards mapped to concrete test case implementations.

4.3 Test Case Generation and Automation

As this paper focuses on test modeling, we just give a brief description of the test case generation and automation process. Generation from Petri nets is basically identical to reachability analysis where all sequences of execution are recorded, each sequence resulting in a test case. Due to the variablity constraints in our extended formalism, a preliminary filtering step has to be executed where transitions are removed that are not satisfiable in the current configuration. The steps of the generation process are:

1. The user selects a subset of the provided platform configurations (cf. Section 3).
2. For each selected platform do the following:
   - 2.1 For each application configuration do:
     - 2.1.1 Join both the platform and application configuration.
     - 2.1.2 Subtract to unsatisfiable transitions and afterwards dangling arcs from the operational model according to the statements of the joined configuration.
     - 2.1.3 Create a new test suite from all sequences of transitions by reachability analysis and record their template-processed keyword sequences as test cases until a coverage criterion is fulfilled.

The coverage criterion plays an important role in the test case generation as it limits the search in the model's behavioral space, which is potentially infinite (due to loops—in Petri nets also due to unbound production of tokens). In the literature, criteria such as state, branch, and path coverage for state machines/charts and transition or place coverage for Petri nets were proposed. Instead of discussing them here, we refer to the respective literature, e.g., [DAGCH08].

If we perform above steps for our running example, e.g., the `Xperia E` platform configuration can be selected in step (1), and further the `Lite` or `Pro` version with all included features and variable value bindings is added in step (2.1.1). The resulting configuration is the union of both. For `Lite` it contains `GPS` but no `PulseReading` feature such that the transitions that are only satisfiable with `PulseReading` are removed from the operational model (cf. Figure 7) in step (2.1.2). Afterwards, in step (2.1.3), test cases are generated. A sample test case is:

$$\begin{aligned} &start \vdash login \vdash validate\_layout(main, \text{“Android 4.1”},\\ &screenDim = \{screenWidth = 480, screenHeight = 320\})\\ &\quad \vdash select\_monitor\_ul \vdash activate\_heart\_rate\\ &\quad \vdash validate(go\_pro) \vdash swipe\_left(2) \vdash quit \end{aligned}$$

At this point, the generated test cases are still abstract and implementation-independent. The last task is to interpret the test cases and map them to platform-specific test code using a template-based model-to-text transformation. The concept of templates is widely used in many keyword-driven test automation systems such as Selenium[4]. For the discussed Android platform, a template must basically produce a JUnit[5] class with one test method for each test case. A well-known template language is XPand[6]. The following listing shows briefly how an XPand-based template can be used:

```
<<DEFINE suite(String activity) FOR test::TestSuite>>
<<FOREACH cases AS c ITERATOR ic>>
<<FILE activity + "TestCase" + ic.counter1 + ".java">>
package test;

import android.test.ActivityInstrumentationTestCase2;
import my.test.auomtation.framework.*;

public class <<activity>>TestCase<<ic.counter1>> extends
     ActivityInstrumentationTestCase2<<<activity>>> {
        public <<activity>>TestCase<<ic.counter1>>() {
                super(<<activity>>.class);
        }

        //generated test sequence
        public void runTest(){
                <<FOREACH c.steps AS s>><<EXPAND term FOR s.keyword>>;
                <<ENDFOREACH>>
        }

        //keyword implementations
        private void swipe_left(int count){
                TestAutomationFramework.swipeLeft(count);
        }

        private void validate(String what){
                if(what.equals(main)){
                        Assert.assertTrue(solo.searchText("Start workout"));
                        //...
                }
        }

        /** ... **/
}
<<ENDFILE>>
<<ENDFOREACH>>
<<ENDDEFINE>>
```

Assuming that each test suite consists of one or multiple test cases, for each case a new file is created. Each test case consists of steps reflecting the keywords shown above. Those keywords now get listed in the `runTest` method. Their implementation then is done by employing the automation framework to control the graphical user interface or to run assertions over its elements. The template makes use of the Android testing framework. The resulting test cases are tailored for specific platforms by our test generation process.

[4] http://www.seleniumhq.org/

[5] http://junit.sourceforge.net/

[6] http://www.eclipse.org/modeling/m2t/?project=xpand

5 Related Work

The means used in our work are based on the findings of software product line testing (SPLT) research [McG07]. An important point is that we do not assume the tested software to be explicitly designed as an SPL. Instead, we are aware of multiple implementations and the SPL domain is based on a common set of requirements these implementations have to fulfill and have to be validated against. In this process, we model a common (shared) behavior and use refining interpretation by templates to create product-specific tests. Thus, we have to tackle some of the central challenges of SPLT as well: the potentially large number of test cases for variable components and variability inherent to test cases.

Lamancha et al. [LUdG09] proposed a model-driven SPLT approach completely based on standards, especially OVM, UML (including the UML Testing Profile [UML-TP]), Query View Transformation (QVT) and the OMG Model-Driven Engineering (MDE) process. After modeling OVM variation points and variants, these are mapped to UML classes and fragments in sequence charts such that the QVT transformation is able to tailor the model according to the selected variants and produce UML-TP-based test cases. Furthermore, Olimpiew and Gomaa showed in [OG05] and [OG09] how their PLUS approach can be used to construct mappings between features, sequence charts, activity diagrams, use cases, test specifications and other model elements and how test cases can be generated from this information. As mentioned, we avoided to use UML as Petri nets have equally well-defined semantics and an easily comprehensible and extensible syntax.

Another approach explicitly targeting software families was proposed by Bertolino and Gnesi in [BG04]. Their PLUTO methodology uses *tags* to parameterize textual use-case-based test cases. Tags can be *alternative*, *parametric*, or *optional*. Moreover, the authors use *categories* to configure the tests. In comparison to our approach, PLUTO does not support test case generation from an operational (i.e., operational) model.

6 Discussion and Future Work

Our central approaches are attributed feature models as well as a subtraction-based and a parameterization-based variability mechanism, which were used to tailor an operational test model for a specific platform configuration and application product. The concepts presented in this paper enable testers to deal with the variability in single and between multiple platform environments and generate test cases automatically from a single operational model.

For our future work, the long-term target is to produce and execute test cases semi-automatically. Furthermore, we aim to extend our test modeling approach with an concept for validating dynamically adapting systems. Thus, it could be beneficial to use dynamic SPLs (DSPLs, [HHPS08]) synergetically with static feature mechanisms discussed in this paper.

Acknowledgment This work has been funded with the projects #100084131 and #100098171 (VICCI) by the European Social Fund (ESF).

References

[BG04] Antonia Bertolino and Stefania Gnesi. PLUTO: A Test Methodology for Product Families. In *Lecture Notes in Computer Science. SpringerVerlag Heidelberg. 3014*, pages 181–197. Springer, 2004.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[DAGCH08] Junhua Ding, Gonzalo Argote-Garcia, Peter J Clarke, and Xudong He. Evaluating test adequacy coverage of high level petri nets using spin. In *Proceedings of the 3rd international workshop on Automation of software test*, pages 71–78. ACM, 2008.

[HHPS08] S. Hallsteinsen, Mike Hichey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *IEEE Computer*, pages 93–95, 2008.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[LUdG09] B.P. Lamancha, M.P. Usaola, and I.G.R. de Guzman. Model-driven testing in software product lines. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 511 –514, sept. 2009.

[McG07] John D. McGregor. Testing a Software Product Line. In *PSSE*, pages 104–140, 2007.

[MCP10] Radu Muschevici, Dave Clarke, and J. Proenca. Feature Petri Nets. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, volume 2. Springer, 2010.

[OG05] Erika Mir Olimpiew and Hassan Gomaa. Model-based testing for applications derived from software product lines. In *Proceedings of the 1st international workshop on Advances in model-based testing*, A-MOST '05, pages 1–7, New York, NY, USA, 2005. ACM.

[OG09] Erika Mir Olimpiew and Hassan Gomaa. Reusable Model-Based Testing. In Stephen H. Edwards and Gregory Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering*, volume 5791 of *Lecture Notes in Computer Science*, pages 76–85. Springer Berlin Heidelberg, 2009.

[Utt07] Mark Utting. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.