Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets

Georg Püschel
Technische Universität Dresden
Georg.Pueschel1@mailbox.tu-dresden.de

Ronny Seiger
Technische Universität Dresden
Ronny.Seiger@tu-dresden.de

Thomas Schlegel
Technische Universität Dresden
Thomas.Schlegel@tu-dresden.de

ABSTRACT
Applications for ubiquitous systems have to be designed to run in contextual environments and on a multitude of software and hardware platforms. To assure their quality in an adequate subset of these static as well as dynamic configurations, variability modeling and model-based testing can be used. In this paper, we present an approach applying Model-based Testing (MBT) and Dynamic Feature Petri Nets (DFPN) to define a test model from which an extensive test suite can be derived. We argue that our method is capable of efficiently modeling context-aware application behavior for test purposes.

Author Keywords
Model-based Testing, Petri Nets, Ubiquitous Systems, Context, Features, Mobile Applications

ACM Classification Keywords
D.2.5 Software Engineering: Testing and Debugging

INTRODUCTION

Mobile and ubiquitous applications are designed and developed to be executed on a multitude of heterogeneous target platforms, i. e., supporting a large number of software and hardware configurations. In addition, they have to adapt to changes within their environment based on the current context model describing "external data, that may influence the application"[1], e.g., location or connectivity information. To support platform independence, software and programming interfaces abstract from platform specific properties. However, the resulting behavior of a system may still differ due to minor implementation differences. Hence, we have to test this specific software within an adequate set of predefined configurations.

The second issue —changing contexts— occurs at runtime. The *system under test* (SUT), depends, e. g., on geospatial data (GPS), adapts to connectivity problems, or changes its graphical interface according to the device's current orientation. In order to test this behavior, we have to keep in mind that these *dynamic* adaptations can occur at almost arbitrary points in time.

To deal with those problems, we designed a workflow for generating test cases, which is depicted in Fig. 1. First, we make use of *feature models* [3], which are a widely-used concept to define commonality and variation in systems, especially in Software Product Lines (SPLs). The static variability of configurations based on software and hardware platform differences can be defined by using feature models spanning a test configuration space. From this space, configurations can either be selected specifically for testing or they can be verified against the feature models' constraints.

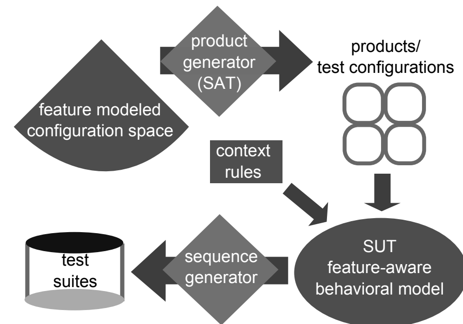

Figure 1. Workflow of test case generation.

Second, we apply *model-based testing* (MBT), which is a means for "the automation of black box tests" [10]. In MBT, models are used to specify the communication between a test environment and the SUT —considered to be a black box— via its interfaces and to generate valid test cases. The main advantages of MBT include a reduction of redundancy, a measurable coverage, and a traceability among artefacts. Thus, we create a test model based on Dynamic Feature Petri Nets (DFPN) to define the SUT's behavior under all possible configurations, and to derive a subset of this model from *context rules*. These context rules associate feature activations or deactivations with actions that have to be executed in order to enable the (de-)activation in the black box SUT. Combining the test model, context rules, and feature models, a generator derives one test suite for each of the application's static configurations.

Overview

The rest of this document is structured as follows: first, we design the features models of an exemplary SUT as well as its Petri net-based behavioral test models. Subsequently, it is shown how parts of our test model are derived from context rules and how test cases are generated. Afterwards, we briefly present an editing tool which implements our approach. In the end, we discuss related work, conclude our own contributions, and outline future research activities.

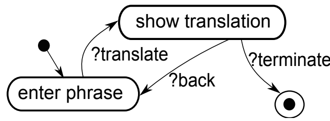

Figure 2. State machine for the TransApp example.

MODELING APPROACH

In this section, we present our modeling strategy. To illustrate our approach, we introduce a brief example from the mobile application domain in the beginning. Fig. 2 shows the state machine for our prototype application called "TransApp", a minimalist translation agent. It consists of two states, the first one enabling the user to enter a sentence and letting the service backend translate it. Second, the user can either terminate the application or return back to translating another phrase. The target language is thereby determined automatically using the system's `Localization` feature and translation is done by the `Translation` feature via local or Internet dictionary look-up. In the following, we systematically build and combine models to define all information necessary for the test case generation.

Configuration Space Feature Models

As discussed initially, one of the most important tasks in test modeling of context-aware systems is to manage variability. In SPL research, this problem has been approached by using feature models, which can also be extended to dependent feature models with the help of relational semantics (more detailed in [7]). Fig. 3 shows the usage of two dependent feature models for TransApp. The feature models are presented as *feature trees*. The upper tree defines the variability of the SUT itself. Apart from the common `core` feature, which contains entities shared among all instances, two abstract features must be selected in all valid products. `Localization` is used to provide an automatic selection of the target language for translation and may either be implemented by using a GPS sensor or by using cellular network data. These specializations are mutually exclusive, i. e., only one of them will be used for localization (marked by an arc between the specialization relationships). For `Translation`, two mutually exclusive options exist as well. Either the system is connected to the Internet and, therefore, an online database is used for looking up the translation, or a local dictionary is consulted.

The lower feature model in Fig. 3 defines the variability space of the underlying system, i. e., of the test environment. The platform consists of a GPS and of a cell phone feature. Internet connection is optional (marked by an empty circle ◯). By defining features as non-optional (standard relationship line with no markings), we do not state that they are mandatory, but that we exclude them from our test configuration set.

Both feature trees are composed by *required* constraints (marked by arrows ↗). For instance, `GPS Localization` requires a `GPS` feature in the underlying system environment. Later on, these relations enable the generator to derive configuration steps leading to a specific test setting.

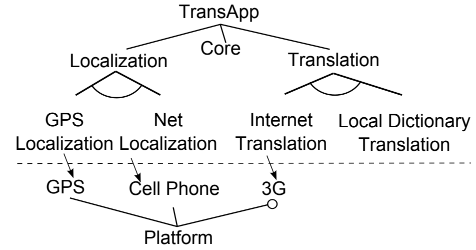

Figure 3. Dependent feature models for TransApp.

Based on the information from the feature models, a set of valid configurations can be generated, e. g.,

$$\{GPS\ Local., GPS, Local\ Dict.\ Translation\}$$

. We can use each configuration as a starting point for a behavioral model defining which features are bound to the system's initial state.

Feature Petri Nets and Test Modeling

Modeling behavior in variable systems can be done by using specific models that define a common behavior space for all selections of features possible. However, many of these metamodels, e. g., modality enriched activity diagrams, have no semantics for feature control. Hence, Muschevici et al. proposed in [5] a Petri net-based model, in which both feature dependent behavior (*Feature Petri Nets*, FPN) and feature controlling behavior at runtime (*Dynamic Feature Petri Nets*, DFPN) can be expressed. While the exact details can be taken from their publications, we give an informal but intuitive definition here and show an exemplary illustration of a DFPN for our test model application in Fig. 4.

First of all, DFPNs are based on basic Petri nets. *Places* (visualized as circles) are partial states which can be connected with *transitions* (black bars) by directed *arcs*. Arcs' starting and ending points have pairwise disjunct types so that they never connect two places or two transitions. A *marking* can be created by putting *tokens* (black dots) into places. A transition is *activated* when all places, which have an outgoing connection with this transition, are marked with tokens. The state of a Petri net can be changed by removing these "incoming" tokens and putting new tokens in all places directly connected by the outgoing arcs from the activated transition. A *trace* is a sequence of states. As multiple places can be marked in the same state, Petri nets can be used to model distributed and parallel processes.

DFPNs adapt the Petri net notion and extend it by marking transitions with *application conditions* and *update expressions*. In Fig. 4, transitions are annotated with these elements in the following way:

$$\frac{\text{application condition}}{\text{update expression}}$$

. An application condition restricts the activation (firing) of transitions to the set of bound (activated) features. At system start, a predefined feature selection is used for initialization. The syntax of a condition follows the grammar

"$\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi$, with $a \in F$" [5]

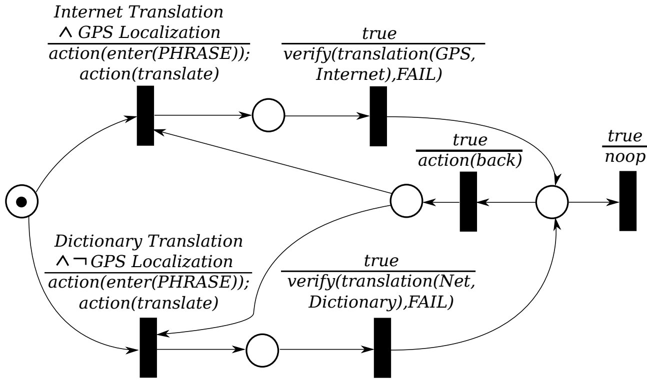

Figure 4. Dynamic Feature Petri Net of our test model.

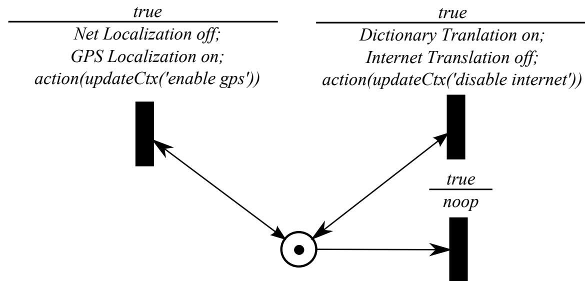

Figure 5. Extract of the context feature model.

where F is the active feature set. We redefined this for reasons of convenience to

$$\varphi ::= a \mid (\varphi \land \varphi) \mid (\varphi \lor \varphi) \mid \neg\varphi \mid true$$

. In Fig. 4, for instance at the beginning, the decision depends on the activation state of the GPS feature, which was determined in advance.

Furthermore, update expressions control the (de-)activation of a feature a (`on`/`off`). Muschevici et al. defined their grammar as:

$$\text{“}u ::= \texttt{noop} \mid a\ \texttt{on} \mid a\ \texttt{off} \mid u; u\text{” [5]}$$

. We added two statements, namely `action` and `verify`, to this notation and extended it to

$$u' ::= u \mid \texttt{action(x)} \mid \texttt{verify(x,v)} \mid u'; u'$$

where x is a term and v is a verdict $\in \{PASS, FAIL\}$. This allows test modelers to define black box interface interactions with the help of arbitrary message patterns (e. g., send-receive, receive-send), whereas an `action(x)` operation indicates to send data to the SUT and `verify(x,v)` operations receive and immediately verify data from it.

For our example case (cf. Fig. 4), we use update expressions to navigate through the application and to validate that a translation was executed correctly. The verified terms may be interpreted as equivalence classes for the test data. By using DFPNs in combination with our extensions, we are able to model feature activation dependent test cases and feature controlling steps.

Defining Feature Dynamics

As presented in Fig. 4, we did not use `on` and `off` operations directly on features in the test model. However, context is the actual driver for changing feature states with respect to the targeted application and system types. We connect both operations with *context rules*, having the form

$$\varphi(\texttt{on}|\texttt{off}) \Rightarrow (\texttt{action(x)}|\texttt{verify(x,v)})$$

and use those to generate and insert a parallel context controlling branch into our test models, as depicted in Fig. 5.

For example, the context rule

$$GPS\ Localization\ \texttt{on} \Rightarrow action('enable\ GPS')$$

would produce the upper left transition. An action performed as a consequence of a feature activation or deactivation must lead to this indirectly controllable operation. For instance, if we perform this test manually, such an operation would contain an instruction for the tester to place the device in a location within GPS satellite range. A more automatic interpretation would be to execute an update over an ontologically modeled context (e. g., by using SPARQL queries).

Using the feature models defined initially, we can derive that it is necessary to deactivate `Net Localization`, because both features exclude each other mutually. The context branch contains exactly one central place, so that in every simulation step only one of its transitions may be activated. Through this parallelism and the feature activation states, the test model and the context branch are able to interact with each other.

Generation of Test Cases

To generate test cases, we have to perform a reachability analysis for the Petri net, i. e., a complete simulation of the combined DFPN consisting of the test model and the context branch to derive all possible traces, each of which corresponding to a specific test case. Additionally, we need to filter direct feature control operations (`on`/`off`), keeping only actions and verifications with runtime relevance. As the latter ones do not influence the (D)FPN execution semantics, we claim that all provable properties of the Petri net also hold true for our extensions.

TOOL SUPPORT

The implementation of our approach is in ongoing development. The *Mobile Application Test Environment (MATE*[1]*)* prototype is depicted in 6. The editor is based on Eclipse and provides a toolset for DFPN-based test models (**1**), the creation of context rules (**2**), and of feature models (**3**). With the help of these models, specific test cases and classes of test cases can be derived to achieve a large coverage of the feature and test space. MATE is also capable of test suite generation and execution. Additionally, an interface is provided to use technology and platform specific test drivers.

RELATED WORK

Our research is related to and influenced by a broad range of research fields. Ichiro Satoh [9] proposed an emulator-based approach for network migration simulation in mobile

[1] **`http://www.quality-mate.org/`**

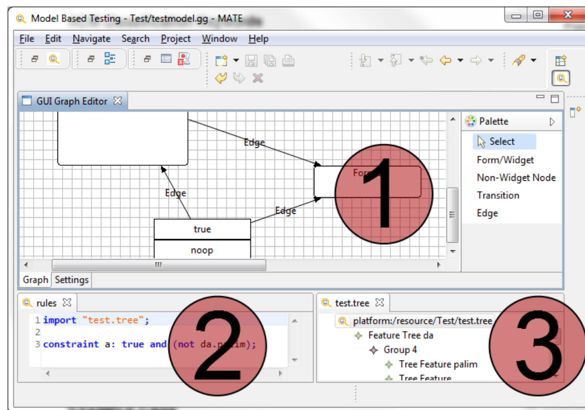

Figure 6. MATE – Mobile Application Test Environment.

computing, which is one aspect of context, while we actually focus on several ones. A more recent work [11] integrated context changes into manually constructed test cases.

Fernandes et al. integrated feature and context modeling in [2] and used specific notations (UbiFEX and Odyssey-FEX). In [12], White and Schmidt proposed using feature models for mobile application design, too. They base their claim on the need for variability management. Similarly, Ridene and Babier manage variability of mobile systems explicitly in test models. For this purpose, they designed a domain specific modeling language [6].

Considering behavioral models for dynamically variable systems, Muscevici et al. compared their DFPN approach to several other Petri net extensions, e. g., self-modifying [8] or reconfigurable [4] Petri nets and stated that they can be mapped to each other.

CONCLUSION AND FUTURE WORK

By applying our approach, it is feasible to link variability in mobile and ubiquitous systems with behavioral test models to generate test suites for different configurations and to change context settings at runtime. To achieve this, we used feature models to define variability of different systems under test and test environment dimensions. With the help of context rules, test engineers can map runtime-dynamic features to test steps necessary to activate or deactivate (bind/unbind) features.

The test model itself is based on Dynamic Feature Petri Nets (DFPN) enriched with a generated parallel branch derived from the information given in the context rules. We claim that this approach provides an expressive means to efficiently create test models for dynamically variable systems, for instance in the ubiquitous and context-aware domain. Compared with related research, we achieve a large and measurable coverage of the configuration space including a variety of context properties, high levels of reusability and automation through modeling, and an extended formalization of test scenarios.

With regard to future work, we intend to evaluate MATE in an industrial case study. In that process, we will measure the effort of creating test models and also its benefit for test projects in comparison to the conventional manual process. Future research problems include coverage constraints that address variability, the introduction of traceability mechanisms, and the design of expressive graphical and textual modeling languages to support test experts.

Acknowledgments

This research has received funding within the projects #100084131 and #100098171 by the European Social Fund (ESF) and the German Federal State of Saxony as well as T-Systems Multimedia Solutions GmbH.

REFERENCES

1. Dalmau, M., Roose, P., and Laplace, S. Context aware adaptable applications - a global approach. *CoRR abs/0909.2090* (2009).
2. Fernandes, P., Werner, C., and Murta, L. Feature modeling for context-aware software product lines. In *Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (San Francisco, CA, USA, 2008)* (2008).
3. Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. Feature-oriented domain analysis (FODA). Tech. rep., Software Engineering Institute, Carnegie Mellon University, 1990.
4. Llorens, M., and Oliver, J. Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. *Computers, IEEE Transactions on 53*, 9 (2004), 1147–1158.
5. Muschevici, R., Clarke, D., and Proenca, J. Feature petri nets. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)* (2010).
6. Ridene, Y., and Barbier, F. A model-driven approach for automating mobile applications testing. In *Proceedings of the 5th European Conference on Software Architecture*, ACM Press (2011), 9.
7. Rosenmüller, M., Siegmund, N., Thüm, T., and Saake, G. Multi-dimensional variability modeling. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)* (2011).
8. Rüdiger, V. Self-modifying nets: A natural extension of Petri nets. In *Automata, Languages and Programming*, vol. 62 of *Lecture Notes in Computer Science*, Springer (1978), 464–476.
9. Satoh, I. A Testing Framework for Mobile Computing Software. *IEEE Transactions on Software Engineering 29*, 12 (2003), 1112–1121.
10. Utting, M. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.
11. Wang, H., and Chan, W. K. Weaving context sensitivity into test suite construction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, IEEE Computer Society Press (2009), 610–614.
12. White, J., and Schmidt, D. C. Model-driven product-line architectures for mobile devices. *Journal On The Theory Of Ordered Sets And Its Applications* (2008), 9296–9301.