TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211X

TUD-FI13-01-Sept. 2013

Püschel, Seidl, Gorzel, Neufert, Aßmann
Software Technology Group

Test Modeling of Dynamic Variable Systems using Feature Petri Nets

TechnischeUniversität Dresden
Nutzerberatung CD
01062 Dresden
Germany
http://tu-dresden.de/cd

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Test Modeling of Dynamic Variable Systems using Feature Petri Nets

Georg Püschel, Christoph Seidl, Mathias Neufert,
André Gorzel, and Uwe Aßmann

University of Technology Dresden, Department of Computer Science
Software Technology Group, D-01062 Dresden
{georg.pueschel, christoph.seidl} @tu-dresden.de,
{mathias.neufert, andre.gorzel} @t-systems-mms.de,
uwe.assmann@tu-dresden.de

Abstract. In order to generate substantial market impact, mobile applications must be able to run on multiple platforms. Hence, software engineers face a multitude of technologies and system versions resulting in static variability. Furthermore, due to the dependence on sensors and connectivity, mobile software has to adapt its behavior accordingly at runtime resulting in dynamic variability. However, software engineers need to assure quality of a mobile application even with this large amount of variability—in our approach by the use of model-based testing (i.e., the generation of test cases from models). Recent concepts of test metamodels cannot efficiently handle dynamic variability. To overcome this problem, we propose a process for creating black-box test models based on *dynamic feature Petri nets*, which allow the description of configuration-dependent behavior and reconfiguration. We use feature models to define variability in the system under test. Furthermore, we illustrate our approach by introducing an example translator application.

1 Introduction

Applications for mobile devices vary heavily in their quality, mostly depending on the vendor's will and ability to assure it. When more competitors with similar alternative products enter the market, they are forced to increase their products' quality as well as the priority of software testing. Test experts can apply model-based testing (MBT) as of «the automation of the design of black box tests» [13]. Thus, they improve the measurable test efficiency in terms of the number of test cases and the degree of test coverage. Black box testing does not require code but a well-defined interface specification, usually consisting of a set of operations and a protocol definition. For mobile software, the interface is mostly graphical as back-end components are often outsourced as Internet services. Traditionally, MBT was designed as a generic approach for arbitrary system types. However, it is desirable to build specialized models for certain domains that let the modeler define problem-specific issues more efficiently.

One issue engineers of mobile software are faced with is a broad heterogenity regarding target platforms (i.e. hardware, software libraries, and operating systems). At the same time, various software platforms are built to run on multiple device types (e.g., smartphones and tablets). In the testing phase, the developed software has to be validated for correct behavior in all these configurations. Due to the black box paradigm, testers cannot directly foresee different effects on the system under test (SUT) between almost-equivalent configurations as hidden differences inside this black box can lead to unpredictable errors. Thus, it is hard to determine configurations to test for adequate coverage.

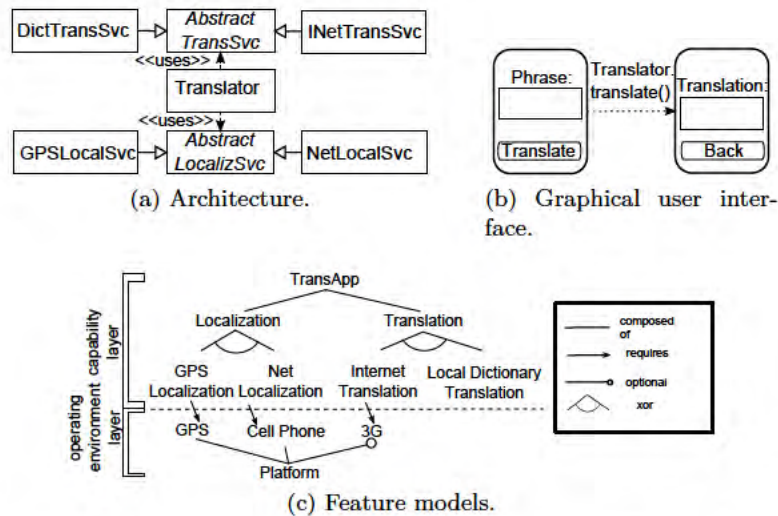
Another problem are changes to the platform's configuration while testing. For instance, the GPS sensor can be switched on or off and the application is then expected to react, e.g., by cancelling a GPS-dependent activity. In consequence, mobile applications testers are faces with a *dynamic variability* challenge as well.

Findings in variability modeling methodology have mainly been driven by software product line (SPL) research [2]. SPLs represent a family of systems that share common assets but differ in some specific parts. A widely used technique for variability modeling in SPL engineering are feature trees [5], which decompose the system's concerns in terms of features connected by a decision tree. Thereby, the tree states several constraints between the features like mandatory existence or mutual exclusion in context of their parent feature node. A huge variety of extensions for feature trees was proposed, e.g., to represent multiplicities or attributes. A subset of features that matches all conditions of a feature tree constitutes a valid configuration (i.e., a product) of the SPL.

As there may be a lot of configurations of a single SPL, testing faces a complexity challenge that has been discussed in the discipline of SPL testing (SPLT). Attemps to reduce complexity, mainly focus on developing approaches on coverage measures and criteria that allow so-called combinatorial testing [8].

SPLT methodology provides means to solve the static part of mobile testing's variability issue such that we can reuse its formalisms to compactly define all possible configurations. The design of dynamically variable systems is a research issue of Dynamic Software Product Lines (DSPLs). A DSPL is an SPL that can be reconfigured at runtime [4]. Therefore, it monitors the environment until a certain trigger is activated that indicates the reconfiguration. Thus, DSPLs are runtime self-adaptive. While there is already a body of knowledge in designing DSPLs, testing approaches are still rare.

In this paper, we will provide remedy to this problem by proposing a process and models that allow the description of reconfigurability in test models. MBT for mobile applications requires concepts for managing static and dynamic variability across platforms and the SUT. The contribution of this paper is an envisioned modeling and generation process plus a metamodel for designing dynamically variable test models. Hence, we define the SUT's variability by using a feature model and later describe how the configuration changes using an extended version of *Dynamic Feature Petri Nets* (DFPN) as test model. The basic DFPN formalism was proposed by Muschevici et al. [9]. It includes behavioral



(a) Architecture.

(b) Graphical user interface.

(c) Feature models.

Fig. 1: Translator sample architecture, GUI and feature models.

entities that can be dependent on features and actively define reconfiguration. The output of our test process are test cases that not only define test operations but also the change that is enforced on the SUT during operation.
This paper is structured as follows: In Sect. 2, we introduce a minimal mobile application example. In Sect. 3, we propose the envisioned process for DFPN-based test modeling. In Sect. 4, we explain how initial configurations are derived. In Sect. 5, we outline the DFPN formalism. In Sect. 6, we describe our model extensions to DFPNs and details of the test process. In Sect. 7, we discuss related work. In Sect. 8, we conclude our approach and outline future work.

2 Minimal Example Case

Fig. 1a depicts the architecture of a minimal example case (UML-like class diagram). The application »TransApp« provides a very basic translation service. It's workflow is illustrated by a graphical user interface (GUI) in Fig. 1b. From the user's perspective, the workflow starts with entering a phrase and clicking the »Translate« button, which triggers the application to translate and print the result in a second form. From this point, the user can click »Back« and enter another phrase to translate.
Basically, there are two ways how variability can affect the user: either the GUI is modified or functionality that is not directly visible to the user gets modified. The latter case is applied in the example. The architecture of TransApp uses either an Internet-based translation service, or—if no connection is available—a local dictionary; the decision depends on the state of connectivity. Translating from local knowledge does produce lower quality results as the database is limited to a smaller word set. In

TransApp's software architecture, variability is handled by inheritance and polymorphism. We can mind a factory pattern, which produces a new object of `INetTransSvc` or `DictTransSvc` for each translation of which are both subtypes of `AbstractTransSvc`. The subtype selection depends on the current connectivity.
To parameterize the translator component, the sample application uses a localization service to retrieve the current country. This information gives an initial intention of the target language and is used to provide higher quality and faster translations. The system may select from two alternatives again: one is based on GPS (`GPSLocalSvc`) data, the other one (`NetLocalSvc`) uses the mobile cell network information for approximating a position.
The resulting feature model is depicted in Fig. 1c. The syntax and semantics of our feature model comply to those of Kang et al. [6]. There are two layers of feature: the upper part identifies the capabilities of the SUT and lower part identifies features of the operation environment required by the SUT's capabilities. The semantics include tree edges for features beeing optional, mandatory, or mutually exclusive (logical xor) in context of their parent feature's selection.
Both translation and localization services are mandatory and for each one an implementation has to be chosen. Alternative implementations conflict so they are mutually exclusive.
We decided only to take operating environments into account that are equipped with GPS and a Cell Phone feature but 3G networks are optional. Several application features depend on the respective operation environment capabilites. For instance, GPS localization can only be used if GPS is provided by the platform.

3 Envisioned Test Modeling Process

Fig. 2 depicts our overall process. It starts from a feature model as already used in our example. These feature models can describe variability in the software platform (operating system, libraries), sets of devices, and the SUT itself.
In the next step, the complete set of all configurations is computed. For valid configurations, the feature models' constraints must hold. We can set a *combinatorial coverage criterion* [8], which defines an additional propositional constraint on them, so that the amount of valid configurations decreases.
Regarding the example case, a test modeler may preselect the `GPS Localization` feature so that only configurations are considered where the feature is active. The selection of a subset of remaining configurations enables us to measure the test coverage in relation to the complete configuration space.
The resulting configurations each define a possible initial state. This can be used as a parameterization of the DFPN-based behavioral model from which test cases are generated.
DFPNs not only define configuration-dependent behavior, but also specify activation or deactivation of features at runtime. Thus, test modelers

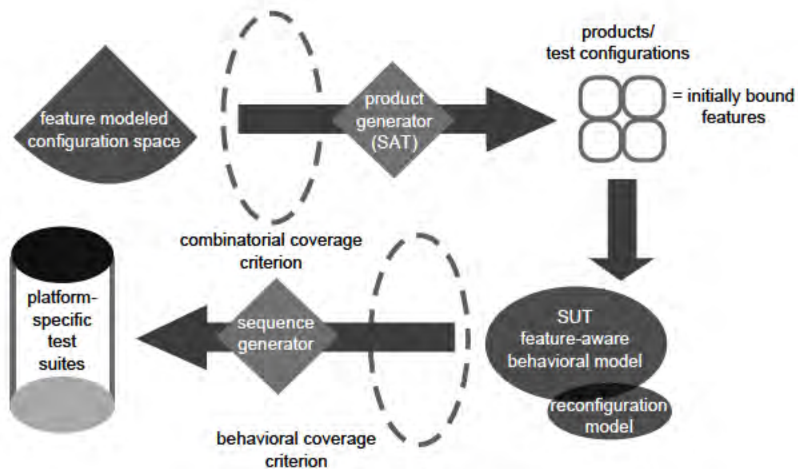


Fig. 2: Overall workflow.

may employ them to define which reconfigurations are possible during runtime. Hence, test modelers create a second DFPN that describes the reconfiguration behavior in parallel to the actual workflow of the application. Both models are used as input to a test case sequence generator, which runs a reachability analysis on the Petri nets. In this process, a *behavioral coverage criterion* can be applied. For Petri-net-based models, several were proposed in the literature [14]. The result of this process is a set of test cases.

4 Derivation of Valid Configurations

Each configuration can deal as initial configuration at deployment time. Thus, we need compute all possible solutions of the feature model's inherent constraints or to validate self-constructed configurations. The latter alternative is feasible if the variant space is too complex to be solved completely. Despite this motivation, a generation of all product configurations would help testers to quantify an analytical *configuration coverage*, i.e., a relation between approached set of test situations and the set of all theoretical valid ones. For instance, statements such as «50% of configurations for software platform P and a set of devices D were tested» could be provided in test service business contracts. For generation and validation, we use a satisfiability (SAT) solver for which our variability models are converted to propositional logic [1]. Existing mobile applications are currently much less complex than desktop software such that the complete exploration of the variability space is feasible in an adequate time with state-of-the-art SAT solvers.

The produced test configurations consist of feature sets that are valid to the feature models' inherent constraints. For instance, the following feature set is a valid configuration in the variant space depicted in Fig. 1c:

$$FS_{sample} = \{TransApp, Localization, Translation,\\ GPS\ Localization, Local\ Dictionary\ Translation,\\ Platform, GPS, Cell\ Phone\}$$

In the next step, test modelers have to define the valid behavior for this configuration and how the configuration may be changed by internally (application workflow) or externally (context). In the following section, DFPNs are introduced, which provide means for modeling such behavior.

5 Introduction to Feature Petri Nets

In this section, we introduce the behavioral metamodel from which test sequences can be generated. The central requirement is that a single model is capable of defining behavior in all configuration states.
Muschevici et al. [10][9] proposed an extension to Petri nets [11], called Feature Petri Nets (FPNs) for modeling dynamically variable systems. A further extension they propose are Dynamic Feature Petri Nets (DFPN), which add the ability to control feature binding at runtime. For test modeling of mobile applications, we use the latter extension. We start with an informal overview of DFPN while the exact formal definition of FPNs and DFPNs can by found in [9]. Fig. 3 depicts an example of a DFPN. Informally, a basic Petri net consists of places (visualized as circles), transitions (black rectangles) and arcs (arrows) connecting mutually places and transitions. Each place can be marked by tokens (black dot). A set of token assignments to places at a discrete point in time is called marking and represents a state of the Petri net. In the operational semantics of Petri nets a state change is applied by letting a transition *consume* tokens from input places and *produce* new ones in output places. Thus, transitions can *fire* if each place in the *input set* of the transition is marked by a token. Petri nets are well-formalized and can be used to model parallelism. In DFPNs, transitions are additionally annotated with a statement with following syntax:

$$\frac{application\ condition\ \varphi}{update\ expression\ u}$$

The *application condition* defines an constraint stating the product configurations under which a transition may fire. Over a set of features F, an application condition is defined as a propositional formula φ with the syntax grammar:

$$»\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi, \text{ where } a \in F« \ [9]$$

Muschevici et al. defined two semantics for FPNs: The first one is *projective*: it changes the Petri net graph by removing all transitions with application conditions not holding in the current configuration. The second semantics definition is operational and reasons on the current configuration which makes it applicable for dynamic reconfigurations as well.

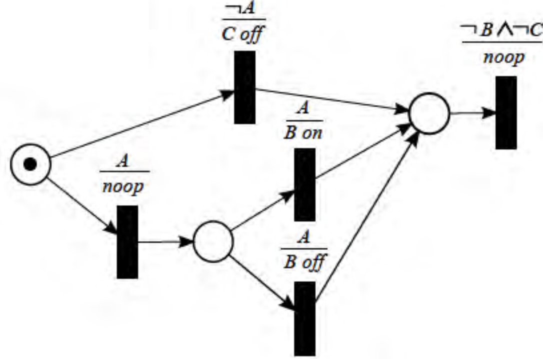


Fig. 3: Dynamic feature Petri net example.

Basically, a transition is activated if its input places are marked and its application condition is true under the current configuration state.
The *update expression* is an operation to manipulate the feature configuration itself. Features either can be switched *on* or *off*. The update expressions' syntax is:

$$\text{»}u ::= \texttt{noop} \mid a\ \texttt{on} \mid a\ \texttt{off} \mid u; u\text{«}[9]$$

where $a \in F$ is a feature and the semantics of **noop** is the identity mapping. As the configuration of a DFPN changes during runtime, here only the operational semantics definition can be applied.
Before executing the behavioral definition, a set of initially bound features has to be defined. In context of our example, this process step was already performed in Sect. 4. In the more formal case of Fig. 3, we assume that the features $\{A, B, C\}$ are selected. Initially, the token may be consumed by two transitions but only the lower one can fire as feature A is activated. For another feature selection where A is `off`, the DFPN would produce a completely different behavior. The transition produces one token so that exactly one of the subsequent transitions can fire. At this point in time, the choice, which of the feature binding operations (B `on` and B `off`) will be executed, is non-deterministical. Depending on this decision, the final transition will either fire or not. At generation time here two different test cases can be produced.
DFPNs satisfy the initially stated requirements: Application conditions provide means for adapting the control flow of the modeled SUT to a specific product configuration and allow modeling the complete behavior of a mobile application for all valid configurations. Further update expressions steer the configuration depending on the control flow.

6 Test Modeling and Generation using DFPNs

When generating test cases from a DFPN, each transition may produce a sequence of test steps while firing. A test step defines an interaction with

the SUT's interface. To consider arbitrary synchronous and asynchronous messaging, we define separate input and output operations:

- `action(X)` produces an output message X, where X may contain a user-defined data term. Modelers are free to adress interfaces and operations, for instance by replacing X with an interface method and respective parameter values.
- `assert(X,Verdict)` retrieves an input message in form of a term from the SUT and verifies whether it matches X. If it does, the test runner continues else it sets the test case's verdict to `Verdict` (e.g., FAIL or ERROR) and aborts test case execution.

The message operations can be integrated into a redefinition of the update expressions' syntax. The altered grammar is as follows:

$$u' ::= \texttt{noop} \mid a\ \texttt{on} \mid a\ \texttt{off} \mid action(*) \mid assert(*,*) \mid u';u'$$

where $a \in F$ is a feature in feature set F. Consequently, we also have to redefine the semantics of *update* to

$$FS \xrightarrow{noop} FS$$

$$FS \xrightarrow{action(*)} FS$$

$$FS \xrightarrow{assert(*,*)} FS$$

$$FS \xrightarrow{a\ on} FS \cup a$$

$$FS \xrightarrow{a\ off} FS \setminus a$$

$$\frac{FS \xrightarrow{u_0} FS' \quad FS' \xrightarrow{u1} FS''}{FS \xrightarrow{u_o,u_1} FS''}$$

where $*$ are arbitrary terms and FS is the modified feature set state in a specific point in time. Basically, the *action* and *assert* operations are filtered so that they cannot not affect the DFPN formalism's proofable properties. Further, we redefine the application conditions' grammar for more convenience as

$$\varphi ::= a \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid \neg\varphi \mid true$$

where $a \in F$ is a feature in feature set F. Again redefine the satisfaction semantics: given an application condition φ and a feature set FS, $FS \vDash \varphi$ *iff* $\varphi = true$ or

$$\begin{aligned}
FS &\vDash a && \textit{iff } a \in FS \\
FS &\vDash \varphi_1 \wedge \varphi_2 && \textit{iff } FS \vDash \varphi_1 \textit{ and } FS \vDash \varphi_2 \\
FS &\vDash \varphi_1 \vee \varphi_2 && \textit{iff } FS \vDash \varphi_1 \textit{ or } FS \vDash \varphi_2 \\
FS &\vDash \neg\varphi && \textit{iff } \varphi \nvDash FS
\end{aligned}$$

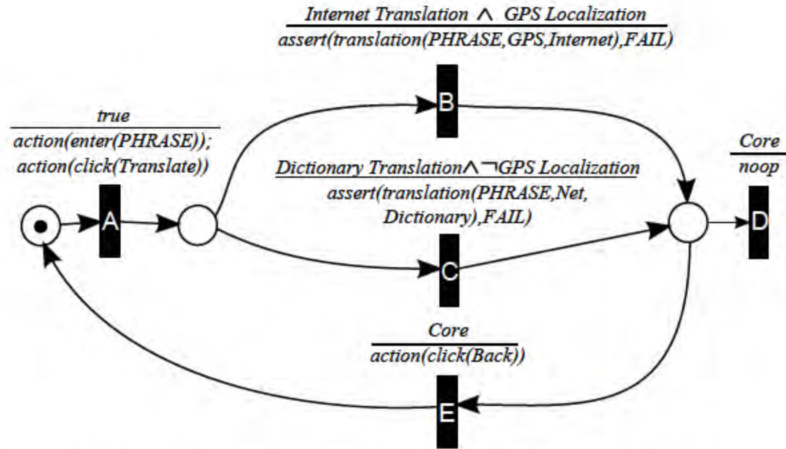


Fig. 4: Excerpt of the manually designed DFPN test model.

In Fig. 4, we applied the defined operations to create a test model for TransApp. The initial state defines a configuration (feature set) that satisfies the constraints of the configuration space variability model (cf. Sect. 4). In the beginning, only transition `A` is activated by a token in its single input place. Each transition defines operations being integrated in the generated test steps (cf. Sect. 6.2). The first action is `enter(PHRASE)`. The executing tester or automatic test driver is required to replace `PHRASE` with an input term from a test database. Afterwards, a click on the «Translate» button is executed by a respective action statement. The subsequent transitions `B/C` define assertions. At test runtime, a function

$$translation(PHRASE, GPS, Internet)$$

has to *oracle* the expected translation output according to a given input phrase and the current configuration. The concrete implementation of those actions and oracles is SUT- and platform-dependent. In testing, data is often organized in test oracle databases that associate input data with expected outputs. Finally, a place is defined where the containing token may be consumed by a termination transition `D` or by a returning transition `E` that states an click action on the «Back» button. This behavioral model can produce test sequences of arbitrary lengths so that an adequacy criterion has to be employed in order to decrease the search space. However, the visualized DFPN is only an excerpt, whereby combinations of features are left out:

$$(Internet\ Localization \wedge Net\ Localization)$$

and

$$(Dictionary\ Translation \wedge GPS\ Localization)$$

.

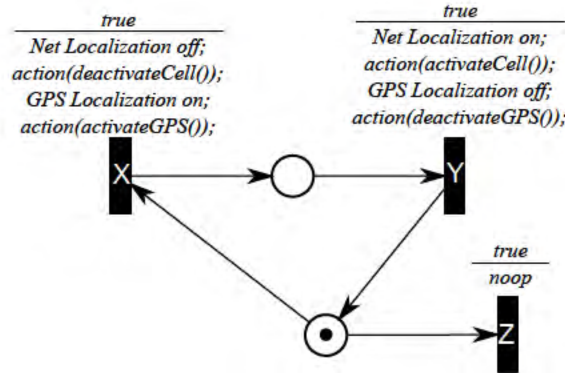


Fig. 5: Reconfiguration DFPN.

6.1 Defining Reconfiguration

With the specified operations, we can either define feature binding explicitly (by using update expressions) or assume the feature selection is static. For dynamic reconfigurations, we now add a parallel DFPN model. The parallelism results in a non-deterministical ordering. The combined behavior model of application workflow and reconfiguration each ordering results in one test case.
An example model is depicted in Fig. 5. All reconfigurations are unrestricted concerning the current configurations (i.e. all conditions are set to `true`). The model contains two reconfiguration transitions `X` and `Y` plus an extra transition `Z` to terminate the execution. The latter one is required as a test case is only produced if no further execution is possible. For this, the finalizing transition removes the token. Each reconfiguration transition (de-)activates features and directly appends an action that has to be generated to manipulate the tested system. For instance, if the cell net localization is switched of, this information has to be propagated to the test interface.
An additional challenge to the test model is to keep track whether the reached configurations are valid with regard to the feature model. In Fig. 1c, both localization types were defined as mutual exclusive. That means, after a transitions is executed and all of its reconfiguration actions were performed, the generator has to check if the reached configuration is valid. By defining such an extra reconfiguration DFPN, the test model is able to restrict the operational order of reconfigurations.

6.2 Test Suite Generation

After we modeled the central DFPN workflow and defined an additional reconfiguration DFPN, test suites can be generated. As an initial configuration (cf. Sect. 4) states assumptions on the deployed system, for each configuration a separate test suite is produced. Based on this initial feature selection we simulate the combined DFPNs (SUT workflow and

reconfiguration DFPN), so that each trace corresponds to a test case. The simulation is identical to a reachability analysis. A DFPN *transition occurence*

$$(M_i, FS_i) \xrightarrow{t_i} (M_{i+1}, FS_{i+1})$$

where (M_i, FS_i) are tupels of markings and features, changed by transitions t_i that can be mapped to a test sequence $s(t_i)$. Given the update expression $u(t)$ mapping $s(t)$ is defined as

$$\begin{aligned} s(t) &= noop && \textit{iff } u(t) = a \texttt{ on } \textit{or } u(t) = a \texttt{ off} \\ s(t) &= u(t) && \textit{otherwise} \end{aligned}$$

where $a \in F$. From a DFPN trace

$$(M_0, FS_0) \xrightarrow{t_0} (M_1, FS_1) \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} (M_n, FS_n)$$

we are now able to create a test case consisting of test steps

$$s(t_0); s(t_1); \ldots; s(t_n)$$

. Informally, only actions and assertions remain in the created test sequence. If we assume that an initial configuration with activated `Internet` feature is selected, a sample test case for TransApp is as follows:

```
1 action(deactivateCell()); //X
2 action(activateGPS()); //X
3 action(enter(PHRASE)); //A
4 action(click(Translate)); //A
5 assert(translation(PHARASE,GPS,Internet),FAIL); //B
```

Comments in each line denote the transitions that produced the respective test action. In the example, the produced generation path first traversed the reconfiguration net and subsequently applied the transitions of the workflow.

7 Related Work

The integration of variability management and testing on mobile devices was investigated by Ridene and Babier [12]. By using a domain specific modeling language (DSML), they specify variability decisions on technical properties of the device configuration. Their DSML is basically an extension to sequence diagrams such that test cases are defined in form of message sequences between involved objects. Thus, the search space is much smaller than in our completely operational model. However, this also makes Ridene et al.'s model less powerful as there are, for instance, no loops.

A second approach to dynamic system testing was developed in context of the DiVA project [3][7]. In DiVA, variability models were used as well. To test a specific system, two phases were defined: In the first «early» phase, instances of a context model were generated and associated with partial solutions of the variability model. Thus, it is possible to test

whether environment changes correctly effect the system's adaptation mode. In the second «operational» phase, the context changes are sequentially applied to the system at runtime. The DiVA approach is quite powerful as it additionally includes the relation between context and system configuration. However, DiVA provides no operational model that allows to explicitly control the task workflow and the order of changes in a synchronized manner as our approach does.

8 Conclusion and Future Work

In this paper, we presented an approach to employ feature-based variability modeling for mobile application testing. For this purpose, we designed a test modeling process for dynamically variable systems, starting from creating a feature model that defines a test configuration space. These models are widely reusable in various test projects where platform-configuration at runtime affects the system under test. In the next step, a SAT-solver generates concrete configurations that are initial system states.
Feature Petri nets are suitable to model dynamically adapting behavior. Enriched with test-specific output labels, test cases can be derived from traces (i.e., traces of the reachability analysis). We sustained syntax and semantics of FPNs, so that property proofs (e.g., liveness) on their original formalism will still hold for our approach.
The major advantage of our approach is the reduction of effort. Test modelers neither have to specify a test model for each configuration nor for each adaptation mode. Both static and dynamic variability are expressed in the feature models and extended DFPNs.
For future work, a requirement is the definition of suitable formal adequacy criteria. They have to be composed from combinatorial and Petri net criteria. Another point are attributes that we aim to integrate in the variability model. Therefore, finite domain variables can be attached to the feature model. As variable assignments state additional information on the DSPL state, they need additional syntax and semantics for FPN conditions and DFPN manipulation actions.
To focus industrial evaluations, a tooling implementation for test modeling and test suite generation from DFPNs is required. Therefore, our Eclipse-based *Mobile Application Test Enironment*[1] *(MATE)* will be extended and used in a case study.

Acknowledgement This research was funded within the projects #100084131 and VICCI #100098171, by the European Social Fund (ESF) and Federal State of Saxony, and by our industrial partner T-Systems Multimedia Solutions.

References

1. Batory, D.: Feature models, grammars, and propositional formulas. In: Software Product Lines. vol. 3714, pp. 7–20. Springer (2005)

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

2. Czarnecki, K., Østerbye, K., Völter, M.: Generative programming. In: Object-Oriented Technology ECOOP 2002 Workshop Reader. pp. 15–29. Springer (2002)
3. Dehlen, V., Solberg, A.: DiVA methodology (DiVA deliverable D2.3) (2010)
4. Hallsteinsen, S., Hichey, M., Park, S., Schmid, K.: Dynamic Software Product Lines. IEEE Computer pp. 93–95 (2008)
5. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA). Tech. rep., Software Engineering Institute, Carnegie Mellon University (1990)
6. Lee, K., Kang, K., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (ed.) Software Reuse: Methods, Techniques, and Tools, Lecture Notes in Computer Science, vol. 2319, pp. 62–77. Springer Berlin Heidelberg (2002), http://dx.doi.org/10.1007/3-540-46020-9_5
7. Maaß, A., Beucho, D., Solberg, A.: Adaptation model and validation framework – final version (DiVA deliverable D4.3) (2010)
8. McGregor, J.: Testing a Software Product Line. Tech. Rep. December, Carnegie Mellon University (2001)
9. Muschevici, R., Clarke, D., Proenca, J.: Feature Petri Nets. In: Proceedings of the 14th International Software Product Line Conference (SPLC 2010). vol. 2. Springer (2010)
10. Muschevici, R., Proença, J., Clarke, D.: Modular Modelling of Software Product Lines with Feature Nets. In: Software Engineering and Formal Methods. pp. 318–333. Springer (2011)
11. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Technische Hochschule Darmstadt (1962)
12. Ridene, Y., Barbier, F.: A Model-driven Approach for Automating Mobile Applications Testing. In: Proceedings of the 5th European Conference on Software Architecture. p. 9. ACM Press (2011)
13. Utting, M.: Practical Model-based Testing: A Tools Approach. Morgan Kaufmann (2007)
14. Zhu, H., He, X.: A Methodology of Testing High-level Petri Nets. Information and Software Technology 44(8), 473–489 (2002)