Context-Aware DSPLs: Model-Based Runtime Adaptation for Resource-Constrained Systems

Karsten Saller*
Real Time Systems Lab
TU Darmstadt
saller@es.tu-darmstadt.de

Malte Lochau
Real Time Systems Lab
TU Darmstadt
lochau@es.tu-darmstadt.de

Ingo Reimund
Real Time Systems Lab
TU Darmstadt
ingo.reimund@es.tu-darmstadt.de

ABSTRACT

Dynamic Software Product Lines (DSPLs) provide a promising approach for planning and applying runtime reconfiguration scenarios to adaptive software systems. However, applying DSPLs in the vital domain of highly context-aware systems, e.g., mobile devices, is obstructed by the inherently limited resources being insufficient to handle large, constrained (re-)configurations spaces. To tackle these drawbacks, we propose a novel model-based approach for designing DSPLs in a way that allows for a trade-off between precomputation of reconfiguration scenarios at development time and on-demand evolution at runtime. Therefore, we (1) enrich feature models with context information to reason about potential context changes, and (2) specify context-aware reconfiguration processes on the basis of a scalable transition system incorporating state space abstractions and incremental refinement at runtime. We illustrate our concepts by means of a smartphone case study and present an implementation and evaluation considering different trade-off metrics.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management—*Adaptive Software Systems*; D.2.9 [**Software Engineering**]: Management—*Software Configuration Management*

General Terms

Algorithm, Design, Performance

Keywords

DSPL, Feature Models, Adaptive Systems, Contexts, State Space Reduction

*This work has been funded by the DFG as part of the CRC 1053 MAKI

SPLC 2013 workshops August 26 - 30 2013, Tokyo, Japan

1. INTRODUCTION

Feature models provide a comprehensive formalism for specifying commonality and variability among the different members of a family of similar (software) products organized in a software product line (SPL) [14]. A *feature* constitutes (1) a product characteristic, i.e., a system property relevant for some stakeholder as identified during domain engineering, as well as (2) a product configuration parameter for deriving stakeholder-specific product variants during application engineering [7].

Fig. 1 shows a sample feature model in FODA notation [14] for a sample smartphone product line case study that serves as a running example throughout this paper. The feature

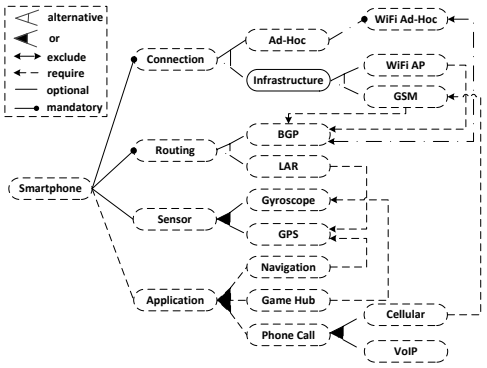
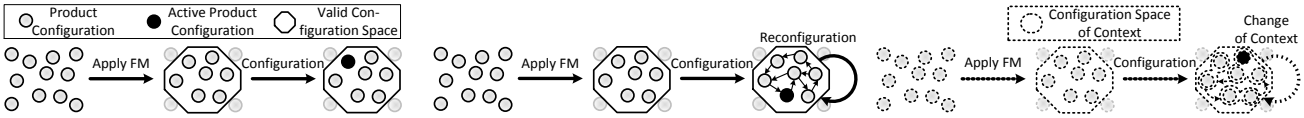


Figure 1: Feature Model for Smartphone SPL

model organizes the supported features in a tree-like hierarchy. For instance, the `Application` features of the smartphone SPL comprises `Navigation`, `Game Hub` and `Phone Call` as direct sub features. Depending on its modality, a single child feature is either *mandatory* for its parent feature, or it is *optional*. For example, the basic `Connection` feature constitute mandatory core functionality to be part of every smartphone variant, whereas the `Sensor` feature is optional. In addition, sets of child features may be collected in groups, where *or*-groups require at least one feature from that group to be present if its parent feature is present, whereas *alternative* groups require exactly one feature to be present. Finally, *cross-tree edges* denote feature dependencies that crosscut hierarchies, e.g., `Navigation` requires `GPS`, `WiFi Ad-Hoc` connections are incompatible with `BGP` etc.

As depicted in Fig. 2(a), a particular *product configuration* of an SPL is obtained by binding all variability, i.e., by either selecting, or deselecting every provided feature according to



(a) SPL Configuration (b) DSPL (Re-)Configuration (c) Context-DSPL Change of Context

Figure 2: (Re-)Configuration in an SPL, DSPL, and Context-Aware DSPL

customer-specific product requirements. The constraints on feature combinations as imposed by a feature model restrict the *configuration space* of an SPL to a subset of *valid* configurations. For instance, the smartphone SPL comprises 19 features thus potentially allowing 2^{19} feature combinations in an unconstrained feature model. For the ease of understanding, we use a feature model, depicted in Fig. 1, that restricts the configuration space to only 66 valid product configurations.

As depicted in Fig. 2(b), a *dynamic* software product line (DSPL) enhances the SPL approach by allowing a product to be not only *(pre-)configured* once during application engineering, but rather by supporting flexible *reconfigurations* at runtime [4]. This enables a product implementation to dynamically evolve to meet continuously changing requirements [3]. A promising field of application for DSPLs constitutes the vital domain of highly-adaptable mobile devices [11] such as our sample smartphone SPL. For instance, if a flash-crowd emerges as a context at runtime, a switch from a current configuration with `Infrastructure`-based communication to a new configuration with `WiFi Ad-Hoc`-based communication might become necessary [13]. Due to the exclude-edge, this may further cause a switch to `LAR`-based routing if `BGP`-based routing is currently active, thus also requiring `GPS` etc. Hence, computing appropriate reconfigurations as reactions to continuously changing runtime *contexts* is challenging due to (1) the various unforeseen and often concurrently emerging environmental influences, each imposing context-specific and even contradicting requirements to be satisfied and (2) the high computational complexity of the resulting constraint solving tasks. Both factors complicate the implementation of adaptation strategies for mobile devices on the basis of DSPLs. In addition, reconfigurations are to be computed and performed in a way (1) that causes no influences on unaffected functionality and, at the same time, (2) respects the inherent resource limitations of mobile devices, as well as runtime efficiency and real-time capabilities. Summarizing, a successful application of DSPLs in the domain of mobile devices is faced with the following research challenges.

1. Perform seamless adaptations to consecutively satisfy complex (re-)configuration requirements of interfering and ever-changing environmental *contexts* [17].

2. Cope with resource constraints that drastically restrict computational runtime capabilities for complex reconfiguration planing and executions tasks [11].

Recent research on DSPLs, therefore, proposes model-based approaches for defining preplanned reconfiguration scenarios [10,20]. Those reconfigurations are then deployed as part of the DSPL implementation by means of a transition system as shown in Fig. 2(b). A state represents the active configuration and transitions specify reconfiguration options.

However, those approaches fail to handle context-aware adaptations of resource-constrained devices, as it is neither possible to deploy the complete configuration space of a complex system onto the device due to limited memory, nor to dynamically explore the configuration space on-demand at runtime due to limited processing capabilities. To overcome these deficiencies, we propose a model-based framework for DSPL design for adaptive, resource-constrained devices incorporating

1. Context-aware reconfiguration planing based on a feature model enriched with context information.

2. Techniques for configuration space reductions based on a transition system specification.

As depicted in Fig. 2(c), the approach considers context requirements to tailor reconfigurations for potentially interfering, compatible contexts emerging at runtime. This approach is based on the observation that users of mobile devices usually move in certain well-known patterns [19]. The design of the transition system for controlling reconfigurations allows for tailoring *incomplete* configuration spaces on the basis of context-aware metrics constituting a trade-off between comprehensively precomputed preconfigurations and on-demand evolutions of the configuration space at runtime. In addition, applying partial transition systems [6] enables the introduction of *partial* states to subsume equivalent configurations thus avoiding unnecessary reconfigurations.

The remainder of this paper is organized as follows. The context-aware DSPL concept is described in detail in Sect. 2 using the smartphone DSPL as a running example. In Sect. 3 we present a sample implementation of the approach. In Sect. 4 we provide an extensive evaluation using different trade-off metrics by means of the smartphone case study. Related work is summarized in Sect. 5 and Sect. 6 concludes our paper.

2. CONTEXT-AWARE DSPLS

To provide autonomous planning and execution of a reconfiguration at runtime, we extend the DSPL variability specification given by a feature model with the requirements imposed by the contextual environment. The enhanced model then builds the basis for a design-time pre-computation of appropriate reconfiguration behavior to be conducted by the implemented DSPL for the corresponding context changes emerging at runtime.

Fig. 3 provides an overview of our approach. The left hand side illustrates the two ingredients of a context-aware DSPL variability specification, i.e., a *feature model* and a *context model*. In Sect. 2.1 we propose how to specify the requirements of runtime contexts via cross-tree constraints between the context model and the feature model. Additionally, we also introduce how we derive a context-specific,

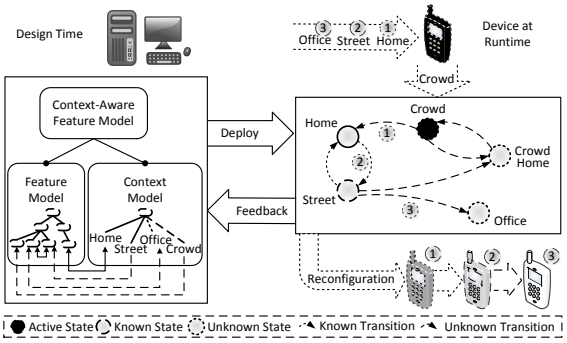


Figure 3: Transition System Configuration Process

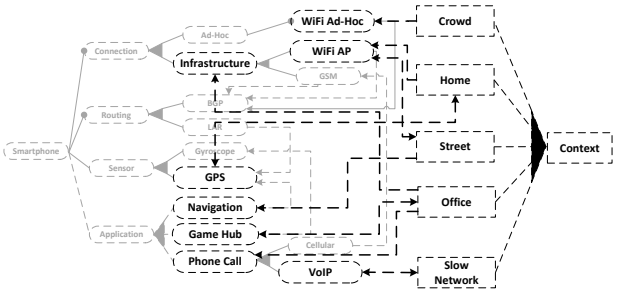


Figure 4: Mapping Contexts to Feature-Selections

tailored transition system, based on those variability constraints, for a pre-planned choice of appropriate reconfigurations at runtime. The right hand side of Fig. 3 depicts sample reconfiguration scenarios based on a precomputed transition system. Therein, every state represents potential configurations which satisfy a context or a set of context for the device and every transition constitutes a reconfiguration. For instance, when starting a state with context `Crowd` being active and the runtime context first changes to `Home` and subsequently to `Street`, the device adapts its configuration by executing the transitions (1) and (2). Besides those pre-planned reconfigurations, the transition system also permits the computation of additional on-demand reconfigurations at runtime for not yet supported context changes. For example, although the context `Office` is not pre-computed and is, therefore, not yet reachable in the transition system, it may be discovered ad-hoc at runtime (3).

2.1 DSPL Specification at Design Time

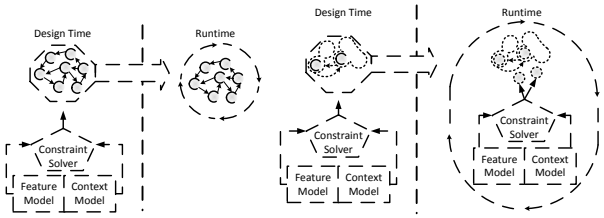
The variability of a DSPL is specified at design time. The set of features under consideration together with the constraints between them are usually represented in a feature model (*FM*) [14]. To provide a (pre-)planning and autonomous execution of a product reconfiguration at runtime, we extend those variability specifications with contextual requirements identified for the DSPL.

For example, the context `Crowd` requires the feature `Navigation` to provide routing functionality within a crowd of people and the context `Office` excludes `Game Hub` to avoid distractions during working hours. Thus, each particular context c is associated with some logical requirements, e.g., a require-dependency $c \rightarrow f$, or a conflict $c \rightarrow \overline{f}$ to feature f. It is further possible for multiple contexts to be active simultaneously in arbitrary combinations, thus imposing interfering requirements to features. Hence, we organize contexts in a similar way as features are arranged in a feature model and w.r.t. context modeling techniques, such as context-goal models, spacial models, or ontology models [2,5]. Therefore, we extend the given *FM* of a DSPL with additional context information, again represented as a feature model over contexts, i.e., a context model. Fig. 4 depicts the extension of our DSPL case study by enriching its original *FM* (left hand side) with a context model (right hand side) thus resulting in a context-aware feature model (*CFM*). Here, all contexts are collected in an *or-group* which allows an arbitrary combination of contexts, always requiring at least one context to be active at runtime. Thereupon, further constraints among contexts may be specified which is, however, not further considered at this point. The feature requirements of contexts are specified by *require* and *exclude* cross-tree constraints leading from the context model to the *FM*. The integration of the variability specifications and contextual requirements in a *CFM* allows us to comprehensively reason about changes in the context of a device in a consistent way as variability specifications and contextual requirements are solvable in a single step. Based on a *CFM*, a transition system is derived that defines context-aware reconfiguration behavior. Every state represents a valid product configuration of the DSPL together with a context combination for which the requirements are satisfied by this configuration as depicted on the right hand side of Fig. 3. For example, in our case study a valid configuration state uses an `Ad-Hoc` connection with a `LAR` (Location Aided Routing [15])-protocol. `LAR` is a routing protocol for wireless ad-hoc networks that uses the geographic location of the device, which makes it suitable for decentralized environments, e.g., a `Crowd` of people. Whenever the device leaves a crowded area and enters an alternative context, e.g., `Home`, an `Ad-Hoc`-based communication is not feasible and the device has to switch to an `Infrastructure`-based communication. Such a reconfiguration is specified by transition (1) in Fig. 3.

Our transition system provides the information necessary to execute a reconfiguration at runtime. However, for such adaptations to be performed autonomously, an appropriate interface for recognizing context (de-)activations is to be provided. Here we abstract from technical details, but rather assume corresponding signaling events to be triggered by changes in the contextual environment, e.g., emitted by the built-in sensors of a device [3]. For example, the context may be influenced by the available network capacity, the connectivity, the geographic position, available services, and the current time [1, 18].

Summarizing, a context-aware DSPL is built upon the following assumptions.

- Contexts represent distinct states of the environment and context changes and , therefore, have an external cause. We assume a single context c to become active instantaneously after being recognized by the system interface (denoted by the event $\langle +c \rangle$) and to become inactive (denoted by the event $\langle -c \rangle$) when being left.
- Multiple contexts may be active in combination.
- Contexts require and/or exclude selected features.
- A reconfiguration is triggered by changes imposed by the currently active contexts.



(a) Complete (b) Incomplete

Figure 5: Strategies for Reconfiguration at Runtime

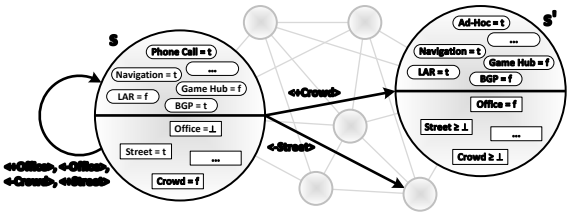


Figure 6: Extract from the Transition System

This concept of a context-aware DSPL has the potential to preplan and optimize efficient reconfigurations at runtime by focusing on the aspects of an autonomous adaptation resulting from changes of the contextual environment.

2.2 DSPL Reconfiguration at Runtime

In existing proposals, all potential reconfigurations of an DSPL are completely determined by a pre-computed *transition system* containing valid configurations and transitions among them [10] as depicted in Fig. 5(a). Alternatively, a not yet computed reconfiguration appropriate for a particular context combination emerging during execution may be also computed on-demand, e.g., by invoking a constraint solver at runtime [9]. In the second case, the precomputed state space is *incomplete* as not all reconfigurations are fully explored, i.e., some valid configurations are not reachable by corresponding transitions.

We propose a combination of selecting reconfigurations precomputed at design time and on-demand computations at runtime to handle reconfigurations due to changes in the context as illustrated in Fig. 5(b). Our approach supports *incomplete* state spaces that are extendable at runtime, as well as *partial* states as an abstraction to reduce the necessity of on-demand solver calls and redundant reconfigurations at runtime.

Incomplete State Space

On the basis of a *CFM*, appropriate sets of configurations are derivable that satisfy (1) one specific context, (2) a valid combination of contexts, as well as (3) no valid combination. Thus, as a first step, the limitation of the state space to *valid* product configurations w.r.t. the feature model leads to a reduction from 2^{19} to 66 valid states for our case study. Thereupon, our concept of contexts allows for further reductions thus resulting in an *incomplete state space* covering particular contextual patterns.

For instance, the requirements of a single context are potentially satisfied by multiple configurations each sufficiently satisfying the contextual requirements at runtime such that the others are redundant and may be, therefore, excluded from the state space. For example, in our case study 21 configurations support the context `Street`. Similarly, configurations that do not satisfy the requirements of any context should be excluded from the configuration process.

Such a coverage of every *single* context offers an extensive reduction of the state space. Besides this one-wise context reasoning, arbitrary combinations of k contexts may arise at runtime. Hence, appropriate configurations satisfying the conjunctions imposed by any valid k-wise combinations are to be provided. Such a k-wise combinatorial context coverage criterion may restrict the computation of a sufficient (sub-)set of configurations at design time, whereas configurations for uncovered combinations beyond k-wise may, again, be computed on-demand as depicted in Fig. 5(b).

Since the usage of a solver is expensive at runtime, those cases should be minimized for resource constrained devices. Hence, a suitable trade-off between memory consumption for a large state space and runtime solving is to be found by means of an appropriate combinatorial parameter k. For example, our case study allows a combination of up to three contexts: `Crowd`, `Slow` `Network`, and `Street` may be active at the same time, although only one configuration supports this combination.

Partial States

A further reduction of the state space and reconfiguration efforts potentially arises in case of feature selections being independent of certain context requirements and vice versa. This allows for abstractions from their concrete value in so-called *partial states*. A partial state, therefore, subsumes a set of concrete states being equivalent w.r.t. to (1) feature selections/deselections and/or (2) context entering/leaving. A particular context c is not relevant for a state if the feature configuration in that state supports both the activation and deactivation of c. Hence, the value of c can be left open (*unbound*) in that state, i.e., a context change affecting c does not require a state change and no reconfiguration is to be performed. Conversely, setting a feature f to unbound in a state denotes that the actual configuration decision for f does not affect the context requirements.

Similar to the combinatorial coverage criteria for active contexts, the pre-computation of valid partial states can be guided by a combinatorial coverage of l features/context simultaneously set to unbound in a state. The computation of l-wise partial states is very expensive as it requires analyzing whether any possible valuations of the l unbound features/contexts are valid for the given *CFM*. However, once found, l-wise partial states reduce (1) the state space by subsuming 2^l states into one and (2) runtime efforts by avoiding unnecessary reconfigurations.

States with unbound features are interpreted as abstract states subsuming sets of equivalent states. Thus, $s(f) = \perp$ denotes a state whose active contexts do not depend on whether f is selected or not and $s(c) = \perp$ denotes states whose feature selections are not affected when entering/leaving context c.

An illustration of this concept is depicted in the excerpt of the transition system from the smartphone case study in Fig. 6. In state s the context `Office` is set to unbound (value $\perp$) as all necessary requirements are met, i.e., `Phone Call` is selected (value `t`), `Game Hub` is deselected (value `f`), and the communication is infrastructure-based via the `BGP`-

protocol. Let us assume, the context `Street` is currently active and, thus, set to selected. Then `LAR` is not included in the currently active configuration and, therefore, the context `Crowd` is deselected. A transition to a subsequent configuration state s' is unnecessary for the activation of the context `Office`, as well as for the deactivation of the contexts `Office` and `Crowd`. The activation/deactivation of `Office` does not obstruct the feature configuration of s and a reconfiguration is not necessary. However, if we assume that the context `Crowd` is entered, a reconfiguration of the system is necessary. As in the currently active state s, the feature `LAR` is deselected, a configuration is required in which `LAR` is selected. As a consequence, `Office` becomes unavailable. However, after this reconfiguration the context `Street` becomes unbound since it is still supported in the target state s', although it has not to be explicitly active.

Representing the reconfiguration semantics of a DSPL by means of a transition system provides various advantages that are summarized in the following.

2.3 Potentials of a Transition System

Our partial transition system is intended to handle the reconfiguration behavior at runtime. However, using such a transition system offers several new application possibilities in addition to planning and executing a reconfiguration.

- **Optimization.** Transition annotations with weights support the quantification of the impact of reconfigurations, e.g., costs metrics. This allows for an optimization of the overall reconfiguration process at design time regarding specific goals, such as energy-efficiency.
- **Verification.** The partial transition system provides the foundation for a formal verification of reconfiguration processes. By using a model checker in combination with a temporal logics, it is possible to identify anomalies, e.g., dead configurations (configurations that may never be reached) or core contexts that are always active.
- **Evolution.** By applying a profiling mechanism it is possible to establish a feedback loop between design time and runtime artifacts as depicted in Fig. 3. The collected runtime behavior of all participating DSPL devices can be used to improve the variability specification within the context model and feature model.

To provide a detailed understanding of our approach, we present a construction algorithm for such a transition system and an adaptation-algorithm in the next section.

3. IMPLEMENTATION

In this section, we describe DSPL design and implementation strategies based on context-aware model-based reconfiguration planning using an extendable partial transition system (PTS) as introduced in the previous section. Based on a feature model *CFM* enriched with context information as defined in the previous section, an appropriate preparation of the PTS state-transition graph is guided by criteria for

1. precomputing a set S of configuration states as a subset of all valid configurations CFG_{CFM} suitably covering reconfiguration scenarios potentially emerging at runtime and

Algorithm 1 PTS Precomputation

```
1: Input: CFM, k,l ∈ ℕ, 1 ≤ k ≤ |C|, 0 ≤ l ≤ |F ∪ C|
2: Output: PTS = (P, S, ⟶)
3: Init: P := (F ∪ C), S := ∅, ⟶ := ∅;
4: // Incomplete State Space Computation
5: for all C ∈ C^k do
6:     if ∄ s_i ∈ S : s_i covers C then
7:         compute s ∈ CFG_CFM where ∀c ∈ C : c is active in s;
8:         S := S ∪ {s};
9:     end if
10: end for
11: // Partial State Abstraction
12: for all s_i ∈ S do
13:     compute CFG^(i,l);
14:     S := S ∪ CFG^(i,l);
15: end for
16: CFG = {s ∈ S | s is subsumed by some partial state in S};
17: S := S ∖ CFG;
18: // Complete Transition Relation
19: for all s, s' ∈ S, s' ≠ s do
20:     ⟶ := ⟶ ∪ {s ⟶ s'};
21: end for
22: return PTS
```

2. identifying equivalent states which can be subsumed into partial states to reduce unnecessary reconfigurations at runtime.

For (1), the precomputation of a PTS state-transition graph, an exhaustive strategy generates the complete state space, i.e., $S_{max} = CFG_{s_0}$, whereas a minimal approach solely precomputes an initial configuration, i.e., $S_{min} = \{cfg_0\}$. The first strategy avoids any expensive computations of reconfigurations at runtime, but leads to high memory consumption, whereas the contrary holds for the second strategy.

Reconfigurations are enforced by context changes at runtime in order to satisfy the configuration requirements of a subset of concurrently active contexts. Thus, appropriate context-aware metrics can be used for tailoring incomplete state spaces that suitably cover reconfiguration options. For example, a strategy would be to select subsets $\mathcal{C}^k \subseteq \mathcal{P}(\mathcal{C}_{CFM})$ of *context combinations* covering any valid k-wise context combination such that for each combination $\mathcal{C} \in \mathcal{C}^k$ a corresponding configuration state $s_i \in S$ is provided.

For (2), parameter $0 \leq l \leq |\mathcal{F} \cup \mathcal{C}|$ denotes the maximum number of features $\mathcal{F}$ and/or contexts $\mathcal{C}$ concurrently set to unbound ($\perp$) during the identification of partial states. Therefore, for each complete configuration cfg_i that corresponds to a state $s_i \in S$ obtained in step (1), a set of configurations $CFG^{(i,l)}$ is computed containing exactly those partial configurations with (1) at most l parameters set to $\perp$ and (2) any parameter that is set to $\perp$ does not contradict CFM. Hence, each of the 2^l possible parameter binding combinations of the l unbound parameters in cfg yields a valid configuration. Therefore, all states corresponding to configurations cfg' being refinement of some partial state in the set $CFG^{(i,l)}$ can be removed from the PTS.

Algorithm 1 summarizes the precomputation of a PTS for a given *CFM* and parameters k and l. The algorithm results in a PTS state-transition graph, defined by a set of predicates P, a set of configuration states S and a set of transitions $\longrightarrow$.

We require $k > 0$, i.e., at least for every single context $c \in \mathcal{C}$ a corresponding configuration is precomputed that satisfies the requirements imposed by c in *CFM*. In lines 5–10, for the set $\mathcal{C}^k$ of all k-wise combinations of contexts, corresponding configurations are computed, e.g., by applying a constraint

Algorithm 2 Runtime Reconfiguration and PTS Evolution

1: **Input:** CFM, PTS
2: **Init:** $s := s_0$, $cfg := cfg_0$
3: **loop**
4: **await** $\langle \chi c \rangle$
5: **if** $(\chi = + \;\&\&\; s(c) < \bot) \parallel (\chi = - \;\&\&\; s(c) > \bot)$ **then**
6: **if** $\exists s' \in S : s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$ **then**
7: **choose** $s' \in S$ **where** $s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$
8: **else**
9: **compute** $s' := cfg' \in CFG_{CFM}$ **where**
10: $cfg'(c) = \texttt{true}$ if $\chi = +$ **and** $cfg'(c) = \texttt{false}$ if $\chi = -$
11: **and** $\forall c' \in C, c' \neq c : cfg'(c') \geq s(c')$
12: **if** $cfg' = \epsilon$ **then**
13: $S := S \cup s_\epsilon$, $\longrightarrow := \longrightarrow \cup \{s \longrightarrow s_\epsilon\}$
14: **else**
15: $S := S \cup s'$, $\longrightarrow := \longrightarrow \cup \{s \longrightarrow s'\}$
16: **end if**
17: **end if**
18: $s := s'$
19: **for all** $f \in \mathcal{F}$ **do**
20: $cfg(f) = \begin{cases} cfg(f) & \text{if } s'(f) = \bot \\ s'(f) & \text{else} \end{cases}$
21: **end for**
22: **end if**
23: **end loop**

solver on a preconfigured context-aware feature model *CFM*. If a configuration is thus found, the requirements of active contexts are compatible and the configuration is added to the set of states. Note that we permit $l = 0$, i.e., no partial states are to be computed. In case of $l > 0$, the set $CFG^{(l,i)}$ of subsuming partial configurations is identified for every state previously computed as described above and added to the set of states S (line 12–15). Thereupon, the set $\overline{CFG}$ of redundant configuration states, i.e., being subsumed by some newly added partial states, are removed from the set of states (line 16-17). Finally, transitions $\longrightarrow$ are added between any pair of remaining states thus building a fully connected state-transition graph.

The implementation of a precomputed PTS on an adaptive device is done by deploying the subset of explored states S together with an encoding of the transition relation to control the reconfiguration life cycle. Algorithm 2 outlines the resulting context-aware reconfiguration process of the adaptive device at runtime on the basis of a PTS specification. Since states s denote configurations $cfg \in CFG$ of a *CFM*, we use both symbols as synonyms. Starting with an initial configuration $cfg_0 \in CFG_{CFM}$ and a start state $s_0 \in S$ such that cfg_0 is directly associated to state s_0, the current configuration cfg is continuously adapted corresponding to the active state $s_i \in S$ of the PTS.

Reconfiguration transitions $\Rightarrow$ are potentially released in the control loop (line 3–23) due to the occurrence of context switches $\langle \chi\, c \rangle$, $\chi \in \{+, -\}$, where $\langle +c \rangle$ denotes context $c \in \mathcal{C}$ to become active and $\langle -c \rangle$ denotes context $c \in \mathcal{C}$ to be left (line 4). A reconfiguration becomes necessary for a context change $\langle \chi c \rangle$ if the current state does not support the requirements of the resulting context combination (line 5), where two cases arise.

- The PTS contains at least one appropriate reconfiguration transition $s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$ (line 6–7).
- The PTS contains no appropriate reconfiguration thus requiring the on-demand computation of an appropriate configuration cfg' (line 9–16).

In the first case, if more than one reconfiguration is available, selection strategies may suggest to choose (1) the one with less change impact w.r.t. the current configuration, or (2) the one with the target state with most unbound parameters.

In case of an on-demand computation of an unsupported configuration cfg' not yet explored in the incomplete state-transition graph, either (1) fails, denoted $cfg' = \epsilon$ due to incompatible context constraints (line 13–14), or (2) it succeeds yielding a fresh state s' and a corresponding transition to reach s' from the current state s (line 15–16). To keep track of erroneous context combinations s_ϵ, such as conflicting context combinations, we add a transition $s \longrightarrow s_\epsilon$ to a special error state $s_\epsilon \notin CFG_{CFM}$. If an appropriate reconfiguration $s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$ is identified, its target state s' becomes the new active state (line 19). The adaptations of the current feature configuration cfg is potentially imposed by that new state applied in line 20-21.

Note that for a reconfiguration $s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$ leading from a state s to a state s' both being precomputed according to Algorithm 1 either (1) $s = s'$ if s already supports $\langle \chi c \rangle$, or (2) $s \longrightarrow s'$ holds due to the fully connected transition graph. In contrast, reconfigurations involving states computed on-demand potentially require multiple reconfiguration steps due to the unplanned on-demand incrementation of the state-transitions at runtime.

Summarizing, for sequences of potential context changes supported by a context-aware feature model *CFM*, we construct a corresponding PTS that provides an appropriate sequence of suitable reconfigurations. Hence, either a reconfiguration already exists in the deployed PTS, or in case of an incompletely explored state space, an appropriate reconfiguration definitely exists and, therefore, may be added to the PTS at runtime. At runtime we use such an PTS to execute reconfigurations according to the contextual changes.

To provide insights into our implementation we present an evaluation incorporating trade-offs between a prearranged reconfigurations and runtime evolutions of the deployed PTS in the next section.

4. EVALUATION

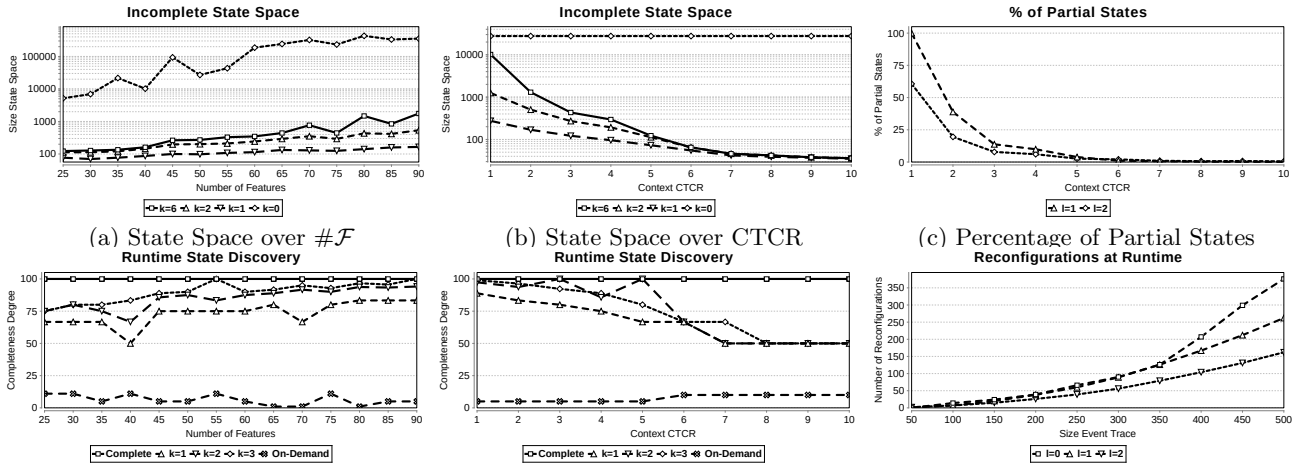
We evaluate the proposed precomputation metrics, i.e., k-wise context combinations and l-wise unbounds, by investigating their impacts and resulting trade-offs on resource demands. For this, we measure (1) the estimated memory required for deploying a PKS state space precomputed during the planning phase, and (2) the estimated processing efforts for performing on-demand reconfigurations caused by simulated context patterns [19] emerging at runtime.

4.1 DSPL Precomputation at Design Time

The configuration space and, therefore, the PKS state space depends on the corresponding *CFM*. The potential state space reduction by an incomplete state space design covering k-wise context combinations depends on the number of features and contexts and the complexity, i.e., the constraint density, of the *CFM*, as well as the choice of k.

Fig. 7(a) shows the state space reduction for feature models scaling over the number of features. The feature models are generated using the the BeTTy-Framework[1] with a relative high cross-tree constraint ratio of 50%. These fea-

[1]http://www.isa.us.es/betty/betty-online



(a) State Space over $\#\mathcal{F}$ (b) State Space over CTCR (c) Percentage of Partial States

(d) State Coverage over $\#\mathcal{F}$ (e) State Coverage over CTCR (f) Reconfiguration Ratio

Figure 7: Evaluation Plots Precomputation and Runtime

ture models are extended to *CFM* with a context-to-feature ratio of $\frac{1}{4}$ and a fixed context-*Cross-Tree Constraint Ratio* (CTCR) of 2. In the following, we define context-CTCR as the ratio of the number of contexts in the cross-tree constraints and the overall number of contexts. Obviously, the state space generated for context combination $k = 1$ (covering single contexts) is the smallest and grows with increasing k. The ratio in which the size of the state space grows decreases with a higher k. In Fig. 7(b) we range over the context-CTCR with a fixed number of features. In our case study the context-CTCR is 2 ($\frac{10}{5}$). Fig. 7(b) shows that the number of possible configurations decreases with increasing context-CTCR.

We further investigate the potential memory savings by state subsumptions using partial states. Fig. 7(c) shows the the percentage of partial states within an incomplete state space for $k = 2$. For $l = 1$ in 100% of the states a single feature/context can be set to unbound up to a context CTCR of 1. When increasing the combination of unbounds in a state, this ratio decreases as shown, e.g., for $l = 2$. For a context CTCR above 5 the percentage of partial states reaches 0% for $l = 1$ and $l = 2$. We want to point out, that with a standard CTCR of 50% and a context CTCR above 3 the restrictions in a CFM are very high, which in turn limits the possibilities of unbound features/contexts. Hence, even if we assume unlimited computational resources at design time, at some point, no more gains w.r.t. to memory savings and the overall constraint ratio arise by further increasing k and l.

4.2 DSPL Reconfiguration at Runtime

An appropriate choice of precomputation metrics for k and l depends on the actual runtime behavior, i.e., the context changes and the corresponding constraints specified in *CFM*. Therefore, we relate the number of incomplete states precomputed for a k-wise context combination and on-demand state computations required at runtime depending on the number of features and context CTCR. We first evaluate for $k = 1$, $k = 2$, and $k = 3$ the frequency of on-demand configuration computations requiring solver calls at runtime. To have a baseline for comparison, we also consider completely precomputed as well as initially empty state spaces only containing an initial state. Fig. 7(d) shows the result for ranging over the number of features. Trivially, no runtime computations arise for completely precomputed state spaces. In case of initially empty state spaces, the coverage achieved by the initial state is always below 20% and, therefore, 80% of the states are to be discovered at runtime. With increasing k, the initial state space coverage increases, accordingly. For instance, for $k = 3$, only 10% of the states are to be discovered at runtime on average. Fig. 7(e) shows the same scenario but ranging over the context CTCR with a fixed number of 50 features. The plot illustrates that with a higher density of contextual constraints the completeness of a precomputed state space decreases significantly.

Besides memory savings, a further benefit of introducing partial states is given by the potential reduction of reconfigurations by means of equivalent configurations. Fig. 7(f) illustrates the number of reconfigurations released by context change events, ranging from 50 to 500 events. The plot illustrates that with increasing l less reconfigurations emerge. Here, only 150 of 500 context changes caused a reconfiguration in case of $l = 2$, whereas for $l = 1$, 250 of them triggered an reconfiguration. If no partial states are considered, up to 360 context events triggered a reconfiguration.

Summarizing, our experiments show, that the size of the state space can be significantly reduced by covering contextual requirements. The higher the contextual coverage, the less configurations have to be computed at runtime. However, comparing $k = 2$ and $k = 3$, we achieved a similar completeness regarding the minimization of on-demand computations at runtime. We further investigated, a pair-wise combination of unbounds drastically reduces the amount of reconfigurations at runtime. These observations justify the costly precomputation of unbound features and contexts.

5. RELATED WORK

Recent approaches on (re-)configuration of SPLs [20] and DSPLs [10] use transition systems to model the configuration process. Just like in our approach, product configuration corresponds to a state and transitions represent reconfigurations. The transitions maybe labeled and/or weighted to guide reconfiguration choices, e.g., by choosing the less expensive transition. Similarly to our approach, the tool Moskitt4SPL[2] provides the means to model a DSPL and de-

[2]http://www.pros.upv.es/m4spl

rive a state space transition system resembling the reconfiguration possibilities. In contrast to our approach, Moskitt4SPL is not capable to reduce the state space according to certain metrics or to summarize equivalent states in partial states. Despite those similarities, existing transition systems for (D)SPLs are neither intended to be tailored for resource-constraint devices nor are they dynamically adaptable at runtime. Thus, these approaches do not scale with the complexity of a feature model and are, therefore, insufficient for resource constraint devices.

Surveys, such as [3], indicate the usage of transition systems to handle the reconfiguration of a DSPL at runtime. However, to the best of our knowledge, there are no approaches that describe (1) a context-aware state space derivation as a transition system as well as (2) the means to reduce the complexity of a state space without restricting the adaptation capabilities at runtime of a DSPL.

Lee et al. [16] presented a similar approach so derive a product configuration based on contextual information. In contrast to our concept of a context sensitive DSPL, this approach aims to support the application engineers to select features suitable for a context based on some selection criteria. Staged configuration approaches for feature models propose (1) to incrementally configure a product through a sequence of stages, and (2) to apply workflow modeling for scheduling the orderings of configuration decisions [8, 12].

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel approach for modeling context-aware DSPL adaptation scenarios for resource constrained systems. The approach is a based on a feature model enriched with context information and, thereupon, defines a transition system to specify appropriate reconfigurations supporting the requirements of potentially interfering runtime contexts patterns. Based on that approach, we proposed two techniques and corresponding context-oriented coverage metrics to optimize the resource efficiency of a DSPL at runtime, i.e., (1) *incomplete state spaces* and (2) *partial states*. Our evaluation shows promising potentials for a significant state space reduction without causing serious further computational efforts at runtime. As a future work, we plan to enhance our approach to incorporate criteria for choosing appropriate reconfigurations, e.g., using transition weights. Besides an a-priori design and verification of adaptation strategies, our approach is also applicable as part of a feedback loop between runtime and design time to enhance the DSPL (re-)design as indicated in Fig. 3. To evaluate our approach regarding resource consumption and further optimization possibilities, we will integrate our DSPL approach into a smartphone testbed architecture.

7. REFERENCES

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *1st Symposium on HUC*, pages 304–307, 1999.

[2] R. Ali, R. Chitchyan, and P. Giogini. Context for Goal-level Product Line Derivation. *3rd International WS on DSPL*, pages 24 – 28, 2009.

[3] N. Bencomo, S. Hallsteinsen, and E. Santana de Almeida. A View of the Dynamic Software Product Line Landscape. *Computer*, 45(10):36–41, 2012.

[4] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *12th SPLC*, pages 23–32, 2008.

[5] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2), 2010.

[6] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *11th CAV*, pages 274–287, 1999.

[7] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[9] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *18th Symposium on FSE*, pages 7–16, 2010.

[10] M. Helvensteijn. Dynamic delta modeling. In *6th WS on DSPL*, 2012.

[11] M. Hinchey, S. Park, and K. Schmid. Building Dynamic Software Product Lines. *Computer*, 45(10):22–26, 2012.

[12] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *13th SPLC*, 2009.

[13] A. Iwata, C. Chiang, G. Pei, M. Gerla, and T. Chen. Scalable routing strategies for ad hoc wireless networks. *Selected Areas in Communications, IEEE Journal on*, 17(8):1369–1379, 1999.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S. A. Peterson. Feature Oriented Domain Analysis (FODA). Technical report, Carnegie-Mellon University, 1990.

[15] Y.-B. Ko and N. H. Vaidya. Location-aided routing (lar) in mobile ad hoc networks. *Wireless Networks*, 6(4):307 – 321, 2000.

[16] K. Lee and K. Kang. Usage context as key driver for feature selection. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2010.

[17] O. Ortiz, A. García, and R. Capilla. Runtime variability for dynamic reconfiguration in wireless sensor network product lines. In *16th SPLC*, volume 2, pages 143–150, 2012.

[18] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *WS on Advanced Context Modelling, Reasoning and Management*, 2004.

[19] H. Verkasalo. Contextual patterns in mobile service usage. *Personal and Ubiquitous Computing*, 13:331–342, 2009.

[20] J. White, B. Dougherty, D. C. Schmidt, and D. Benavides. Automated reasoning for multi-step feature model configuration problems. In *13th SPLC*, pages 11–20, 2009.