TECHNISCHE
UNIVERSITÄT
DARMSTADT

Model-Based Runtime Adaptation of Resource Constrained Devices

VOM FACHBEREICH 18
ELEKTRO- UND INFORMATIONSTECHNIK
ZUR ERLANGUNG DER WÜRDE
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

MSC. KARSTEN SALLER

GEBOREN AM

30. MAI 1981 IN DARMSTADT

REFERENT: PROF. DR. RER. NAT. A. SCHÜRR
KORREFERENT: PROF. DR.-ING. INA SCHAEFER

TAG DER EINREICHUNG: 14. AUGUST 2014
TAG DER DISPUTATION: 12. DEZEMBER 2014

D17
DARMSTADT 2015

The work of Karsten Saller was supported by the German Research Foundation (DFG) in the Collaborative Research Center (SFB) 1053 “MAKI - Multi-Mechanism-Adaptation for the Future Internet” at the Technische Universität Darmstadt.

Please cite this document as:

URN: urn:nbn:de:tuda-tuprints-43224
URL: http://tuprints.ulb.tu-darmstadt.de/id/eprint/4322

This document was provided by tuprints,
TU Darmstadt E-Publishing-Service
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

DECLARATION OF AUTHORSHIP

I warrant that the thesis presented here, is my original work and that I have not received outside assistance. All references and other sources used by me have been appropriately acknowledged in the work. I further declare that the work has not been submitted for the purpose of academic examination, either in its original or similar form, anywhere else.

I hereby grant the Real-Time Systems Lab the right to publish, reproduce and distribute my work.

Darmstadt, 14. August 2014

Karsten Saller

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [Knu74]

DANKSAGUNG

Mein besonderer Dank gilt Dr. Malte Lochau, der mich maßgeblich bei der Umsetzung meiner Ideen und Konzepte unterstützt hat und mich unermüdlich beim Schreibprozess dieser Arbeit beigestanden hat. Es freut mich zu wissen, dass unsere gemeinsamen Arbeiten auch zukünftig in den Sonderforschungsbereich MAKI einfließen werden. Ebenso wichtig für mich war die Unterstützung von meinem Doktorvater Prof. Dr. Andy Schürr. Er hat meine Ideen stets gefördert und mir dabei auch meine Freiheit gelassen, unterschiedliche Themen und Konzepte zu verfolgen. Damit konnte ich meinen eigenen Weg in der Forschung finden, wofür ich Prof. Schürr sehr dankbar bin. Gerne erinnere ich mich an unsere regelmäßigen Treffen, an denen wir meine Ideen besprochen haben. Schließlich möchte ich mich auch bei Prof. Dr. Ina Schaefer bedanken, die mich auf dem FOSD-Treffen 2012 in der Softwareproduktlinien Community willkommen hieß und zudem das Zweitgutachten dieser Arbeit übernommen hat.

Rückblickend möchte ich hier auch meine Kollegen Christian Groß, Max Lehn, Kamill Panitzek und Dominik Stingl erwähnen, die mich seit Beginn meiner Forschungsaktivitäten in der QuaP2P-Gruppe bis zu meinem Ende der Promotion begleiteten. Zusammen haben wir auch an dem Sonderforschungsbereich MAKI gearbeitet, der unsere Forschungsaktivitäten aus QuaP2P die nächsten Jahre fortführen und intensivieren wird.

Zu vielen Teilen der vorliegenden Arbeit haben Studenten im Rahmen von zahlreichen studentischen Arbeiten beigetragen. Hier möchte ich insbesondere die ausgezeichneten Bachelorarbeiten von Stefan Bauregger und Thomas Schnabel hervorheben. Zudem möchte ich Ingo Reimund an dieser Stelle für seine langjährige Unterstützung als wissenschaftliche Hilfskraft danken.

Stellvertretend für die vielen guten Freunde, die mich auf meinem Lebensweg begleitet haben, möchte ich hier meine langjährigen Kommilitonen Daniel Ackermann, Matthias Bernges und Johannes Kohlmann nennen.

Rückblickend auf meine Zeit am Fachgebiet Echtzeitsysteme möchte ich zudem Anthony Anjorin und Dr. Sebastian Oster danken, die immer gerne zu einem Gedankenaustausch bereit waren. Schließlich möchte ich meinen Eltern dafür danken, dass sie mir diesen Weg ermöglicht und mich stets unterstützt haben.

Darmstadt, im August 2014

ABSTRACT

Dynamic Software Product Line (DSPL) engineering represents a promising approach for planning and applying runtime reconfiguration scenarios to self-adaptive software systems. Reconfigurations at runtime allow those systems to continuously adapt themselves to ever changing contextual requirements. With a systematic engineering approach such as DSPLs, a self-adaptive software system becomes more reliable and predictable. However, applying DSPLs in the vital domain of highly context-aware systems, e.g., mobile devices such as smartphones or tablets, is obstructed by the inherently limited resources. Therefore, mobile devices are not capable to handle large, constrained (re-)configuration spaces of complex self-adaptive software systems.

The reconfiguration behavior of a DSPL is specified via so called feature models. However, the derivation of a reconfiguration based on a feature model (i) induces computational costs and (ii) utilizes the available memory. To tackle these drawbacks, I propose a model-based approach for designing DSPLs in a way that allows for a trade-off between pre-computation of reconfiguration scenarios at development time and on-demand evolution at runtime. In this regard, I intend to shift computational complexity from runtime to development time. Therefore, I propose the following three techniques for

(1) enriching feature models with context information to reason about potential contextual changes,

(2) reducing a DSPL specification w.r.t. the individual characteristics of a mobile device, and

(3) specifying a context-aware reconfiguration process on the basis of a scalable transition system incorporating state space abstractions and incremental refinements at runtime.

In addition to these optimization steps executed prior to runtime, I introduce a concept for

(4) reducing the operational costs utilized by a reconfiguration at runtime on a long-term basis w.r.t. the DSPL transition system deployed on the device.

To realize this concept, the DSPL transition system is enriched with non-functional properties, e.g., costs of a reconfiguration, and behavioral properties, e.g., the probability of a change within the contextual situation of a device. This provides the possibility to determine reconfigurations with minimum costs w.r.t. estimated long-term changes in the context of a device.

The concepts and techniques contributed in this thesis are illustrated by means of a mobile device case study. Further, implementation strategies are presented and evaluated considering different trade-off metrics to provide detailed insights into benefits and drawbacks.

ZUSAMMENFASSUNG

Dynamische Software Produktlinien (DSPLs) stellen einen vielversprechenden Ansatz für die Planung und Ausführung von Rekonfigurationen selbst-adaptiver Softwaresystemen dar. Laufzeitrekonfigurationen erlauben es einem System sich kontinuierlich den sich ständig ändernden kontextuellen Anforderungen anzupassen. Jedoch wird die Anwendbarkeit von DSPLs auf eine kontextsensible Domäne, wie die der mobilen Endgeräte (Smartphones, Tablets, etc.), durch eine inhärente Limitierung von Ressourcen eingeschränkt. Daher sind solche mobilen Endgeräte nicht in der Lage, mit den (Re-)Konfigurationsräumen komplexer adaptiver Systeme umzugehen.

Das Rekonfigurationsverhalten einer DSPL wird mittels so genannter Feature-Modelle spezifiziert. Die Ableitung einer Rekonfiguration auf der Basis eines Feature-Modells verbraucht jedoch (i) Berechnungsressourcen und (ii) Speicher. Um diese Nachteile in den Griff zu bekommen stelle ich einen modellbasierten Ansatz für die Spezifikation einer DSPL vor, der darauf ausgelegt ist, einen Ausgleich zwischen zur Entwicklungszeit vorberechneten Rekonfigurationsszenarien und bedarfsgetriebenen Rekonfigurationen zur Laufzeit zu bieten. Daher verschiebe ich den Berechnungsaufwand während der Laufzeit eines Endgerätes und zur Entwicklungszeit einer DSPL. Hierfür schlage ich die folgenden Techniken vor:

(1) Anreichern von Feature-Modellen mit kontextuellen Informationen, um mögliche Änderungen im Gerätekontext zu erkennen,

(2) Reduktion der DSPL-Spezifikation in Bezug auf die individuellen Charakteristiken eines mobilen Endgerätes sowie

(3) Erstellung eines kontextsensiblen Rekonfigurationsprozesses auf Basis eines Zustandsraumes, der zur Laufzeit inkrementell verfeinert wird.

Zusätzlich, zu diesen Optimierungsschritten, die zur Entwicklungszeit ausgeführt werden, stelle ich ein Konzept vor, welches angedacht ist, um zur Laufzeit ausgeführt zu werden:

(4) Lang-Zeit Reduktion der Ausführungskosten einer Rekonfiguration zur Laufzeit auf Basis eines DSPL-Transitionssystems.

Um dies umzusetzen, wird ein DSPL-Transitionssystem mit nicht-funktionalen Eigenschaften angereichert, wie z.B. Kosten einer Rekonfiguration. Darüber hinaus wird das Kontextverhalten mittels Wahrscheinlichkeiten erfasst, um zu beschreiben, wann ein Kontext betreten bzw. verlassen wird. Hiermit können Rekonfigurationen ausgewählt werden, die, in Bezug auf die zu erwartenden Kontextänderungen, auf eine lange Sicht hin minimale Kosten verursachen.

Die in dieser Arbeit erarbeiteten Techniken und Konzepte werden anhand eines Fallbeispiels über mobile Endgeräte vorgestellt. Zusätzlich werden Implementierungsstrategien diskutiert und evaluiert in Bezug auf mögliche Vor- und Nachteile der Ansätze.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications.

Book Chapters

M. Lehn, T. Triebel, C. Gross, D. Stingl, K. Saller, W. Effelsberg, A. Kovacevic, and R. Steinmetz. *Designing Benchmarks for P2P Systems*, pages 209–229. From Active Data Management to Event-Based Systems and More. Springer, 2010.

K. Saller, K. Panitzek, and M. Lehn. *Benchmarking Methodology*, volume 7847 of *LNCS*, pages 19–67. Springer, 2013. Published in "Benchmarking Peer-to-Peer Systems - Understanding the Quality of Service in Large-Scale Distributed Systems".

D. Stingl, C. Gross, and K. Saller. *Decentralized Monitoring in Peer-to-Peer Systems*, volume 7847 of *LNCS*, pages 81–111. Springer, 2013. Published in "Benchmarking Peer-to-Peer Systems - Understanding the Quality of Service in Large-Scale Distributed Systems".

Journal Article

A. Anjorin, K. Saller, I. Reimund, S. Oster, I. Zorcic, and A. Schürr. Model-driven rapid prototyping with programmed graph transformations. *Journal of Visual Languages & Computing*, 24(6), pages 441–462, 2013. Special Issue on Graph Transformation and Visual Modeling Techniques II.

Refereed Conference And Workshop Papers

A. Anjorin, K. Saller, M. Lochau, and A. Schürr. Modularizing Triple Graph Grammars using Rule Refinement. In S. Gnesi and A. Rensink, editors, *Proceedings of Fundamental Approaches to Software Engineering*, LNCS. Springer, pages 340–354, 2014.

A. Anjorin, K. Saller, S. Rose, and A. Schürr. A Framework for Bidirectional Model-to-Platform Transformations. In K. Czarnecki and G. Hedin, editors, *Proceedings of the 5th International Conference on Software Language Engineering (SLE)*, LNCS. Springer, pages 124–143, 2012.

P. Mukherjee, K. Saller, A. Kovacevic, K. Graffi, A. Schürr, and R. Steinmetz. Traceability Link Evolution with Version Control. In *Evolutionäre Software- und Systementwicklung - Methoden und Erfahrungen (ESoSyM-2011), Workshop im Rahmen der Konferenz SE*. Bonner Köllen Verlag, pages 151–161, 2011.

D. Stingl, C. Gross, K. Saller, S. Kaune, and R. Steinmetz. Benchmarking Decentralized Monitoring Mechanisms in Peer-to-Peer Systems. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*. ACM, pages 193–204, April 2012.

K. Saller, M. Lochau, and I. Reimund. Context-Aware DSPLs: Model-Based Runtime Adaptation for Resource-Constrained Systems. In *Proceedings of the 17th International Software Product Line Conference co-located workshops*, pages 106 – 113. ACM Press, ACM, August 2013.

K. Saller, S. Oster, A. Schürr, J. Schroeter, and M. Lochau. Reducing Feature Models to Improve Runtime Adaptivity on Resource Limited Devices. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, volume 2 of *ACM*, pages 135 – 142. ACM, 2012.
K. Saller, D. Stingl, and A. Schürr. D4M, a Self-Adapting Decentralized Derived Data Collection and Monitoring Framework. In *Workshops der wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2011 (WowKiVS 2011)*, volume 37 of *Electronic Communications of the EASST*, pages 1–12, March 2011.

Technical Report

M. Riecker, W. Müller, M. Hollick, and K. Saller. A Secure Monitoring and Control System for Wireless Sensor Networks. Technical Report TR-RI-11-313, Universität Paderborn, Paderborn, 2011.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AP	Access Point
BDD	Binary Decision Diagram
BGP	Border Gateway Protocol
cfm	context-feature model representation as a propositional formula
context-CTCR	Context-Cross-Tree Constraint Ration
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CTCR	Cross-Tree-Constraint Ratio
DSPL	Dynamic Software Product Line
fm	feature model representation as a propositional formula
FTP	File Transfer Protocol
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HSDPA	High Speed Downlink Packet Access
IP	Internet Protocol
KS	Kripke Structure
LAR	Location Aided Routing
LTE	Long Term Evolution Communication Protocol
MAPE-K	Monitor-Analyze-Plan-Execute-Knowledge
PA	Probabilistic Automaton
PC	Personal Computer
PKS	Partial Kripke Structure
PWA	Probabilistic-Weighted Automaton
QMC	Quine and Mc-Cluskey algorithm
RC	Research Challenge
SAS	Self-Adaptive Software System

SCP	Secure Copy Protocol
SPL	Software Product Line
VoIP	Voice over IP
WA	Weighted Automaton
WLAN	Wireless Local Area Network

Part I

INTRODUCTION & BACKGROUND

1 INTRODUCTION

Software systems have to deal with a constant growth of information they have to process. Such software systems are present in the daily life of a person, be it a smartphone a person carries or a data-center that delivers the results for a web-search query. Every smartphone is equipped with a software system providing numerous functionalities to the user. The growth in the smartphone market also increases the demand to satisfy the requirements of each potential user for such software systems. For example, the plot in Figure 1.1 illustrates the importance and the growth of the mobile device market. In 2007 a total amount of 122.32 million smartphones were sold worldwide. Until 2013 the smartphone market increased by 800% to 967.78 million sold devices [Gar14].

The simultaneous increase of information and the integration of technology in our surroundings require new approaches for designing, implementing, executing, and managing upcoming software systems for mobile devices. Such software systems have to be

- *adaptable* in order to react on changes in their context at runtime,
- *configurable* in order to satisfy the needs of a stakeholder at design time,
- *responsive* in order to access certain functionality in real-time, and
- *energy-efficient* in order to provide their functionality as long as possible.

Highly dynamic systems increasingly exceed a level of complexity that is manageable by a person. This implies that the human effort to handle such systems, i.e., tasks such as setup, running, maintaining the system, increases with the complexity of the software system. For example, the market for android applications consists of 1.2 million applications[1] that constitute a multitude of external services, e.g., an integration of *Facebook*, *Skype*, or several email accounts, for an optimal user experience. However, the more applications and services are being integrated into a mobile device the higher is the demand for a simple usage [BE01, CE10, FSSA06].

Self-adaptation is emerging as a necessary technique to manage the complexity in software systems [CLG+09]. Self-adaptive software systems are software systems that are able to adjust their behavior in response to their ever changing requirements imposed by their contextual situation, e.g., their physical environment or by the software system itself. This is especially the case for highly dynamic systems such as context-aware or ubiquitous systems [Bro10].

[1] http://www.appbrain.com/stats/number-of-android-apps

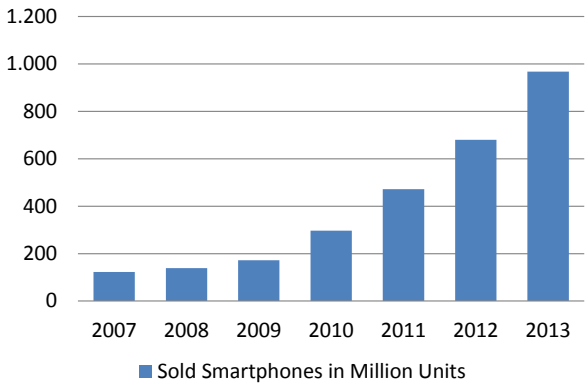


Figure 1.1: Amount of smartphones sold to customers worldwide from 2007 to 2013 according to *Gartner* [Gar14].

To tackle the issue of complexity in self-adaptive software systems, techniques from the domain of *autonomous computing* may be applied. Autonomous computing [Hor01] describes systems that adapt without the need of human interaction. Thus, an autonomous system configures, adapts, and maintains itself at runtime.

AUTONOMOUS SYSTEMS. Biological systems, such as the human nervous system, provided the guidelines for the concept of an autonomous system. The human nervous system acts autonomously to external or internal stimuli. For example, the size of our pupil adapts itself according to the current light intensity. If the human nervous system would not react autonomously a human would have to continuously concentrate on adapting the body to the environment.

Autonomous software systems require appropriate abstractions and models for understanding, controlling, and designing the autonomous behavior, which is executed at runtime. According to [CE10, CLG+09, Hor01], the issue of designing autonomous behavior constitutes an important research challenge. Understanding the problem and selecting a suitable solution requires precise models for representing important aspects of the self-adaptive system, its users, and its context. Furthermore, suitable software engineering methods are required to model important aspects of a self-adaptive software system, such as controlling the adaptivity of an autonomous system, the user behavior, and the requirements imposed by a contextual situation. Current software engineering approaches specify a system at design time in a static manner according to identified requirements as agreed with some stakeholders. Thus, the design of the system is derived prior to runtime and does not reflect the dynamic aspects, which occur at runtime. Since an adaptive system has to adjust its operational state at runtime, techniques to model autonomous behavior occurring at runtime are also considered to be an important research challenge [CE10, CLG+09].

Consequently, to develop autonomous self-adaptive software systems a software engineering approach is required that

- abstracts from the complexity of an adaptive system and
- is applicable to represent the dynamic aspects of an ever-changing context.

Among numerous techniques to tackle the challenges imposed by autonomous systems, Model Driven Development is an extensively investigated domain and has proven itself as a viable concept to abstract from the complex aspects of an (autonomous) self-adaptive software system.

MODEL DRIVEN DEVELOPMENT. Model driven development [AK03, BG01, Voe11] is a software development methodology that intends to capture important aspects of a system through appropriate models. In this regard, model driven development focuses on modeling concepts rather than on computing or algorithmic concepts. The specified models are usable to automatically analyze the system. Therefore, model driven development techniques support key aspects for an autonomous adaptation. To execute an adaptation autonomously, the system requires knowledge as a semantic base, e.g., when an adaptation is to be executed and which system properties have to be adapted [IBM06]. The knowledge, which is required to derive an adaptation decision autonomously at runtime, is specifiable via model driven development concepts [BBF09].

SOFTWARE PRODUCT LINES. Software Product Line Engineering is a domain that uses models to abstract from complexity imposed by large-scale software systems. Software Product Lines (SPLs) are a popular approach for the systematic reuse of software artifacts in development of a family of similar software systems instead of a single software system. Prominent SPLs are the Microsoft Office Suite or the Android operating system for mobile devices from Google. For example, the plot depicted in Figure 1.2 illustrates the vendor specific customization of the different operating systems for mobile devices, i.e., *Android* from Google and *iOS* from Apple. Android, as an SPL, is customized by different vendors. For example, 34% of the Android operating system deployed on mobile devices are customized by Samsung, 27% are customized by HTC, 21% are customized by Motorola, and 18% are customized by other remaining vendors [Nie14]. In contrast to that, iOS is *not* customized by any other vendor than the owning company Apple.

SPLs offer a systematic reuse of software artifacts within a range of products sharing a common set of features, i.e., units of functionality. For example, connectivity features, e.g., Long Term Evolution Communication Protocol (LTE), Global System for Mobile Communications (GSM), Wireless Local Area Network (WLAN), in the Android operating system are re-usable by vendors that develop a customized Android variant.

Particular features may be considered for the entire SPL as not necessarily being part of each software product. In this regard, a feature constitutes

- a product characteristic, i.e., a system property relevant for some stakeholder as identified during domain engineering, and
- a product configuration parameter for deriving stakeholder-specific product variants [CW07].

In order to derive a specific software product of an SPL, features are selected to be either part of the product or they are deselected from the product [PBv05]. Such variability is specified via so called *feature models*.

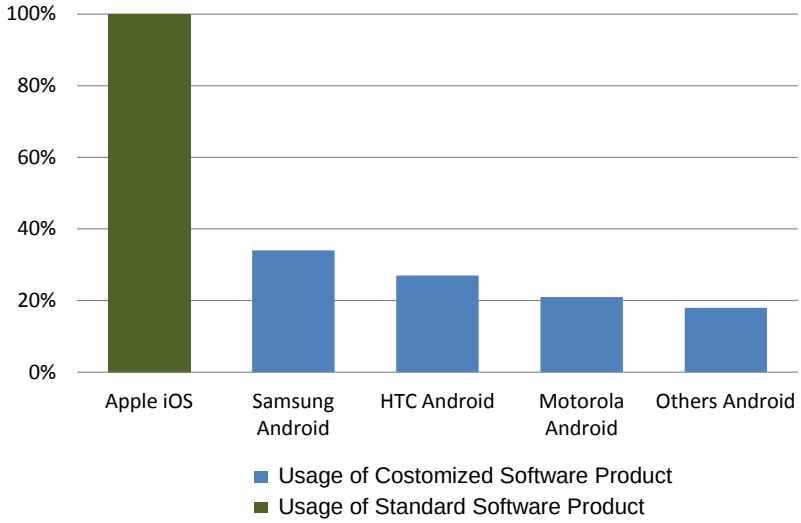


Figure 1.2: Percentage of smartphone operating systems and vendor specific customization in the U.S. in the second quarter of 2012 according to *Nielsen* [Nie14].

Feature models provide a comprehensive formalism for specifying commonality and variability among the different members of a family of similar software products organized in an SPL [KCH+90]. A feature model corresponds to a specification of feature constraints, i.e., dependency and incompatibility relations between features. For example, a routing protocol feature requires an Internet connection and a GSM-based connection may not be active in combination with a WLAN-based connection.

For instance, vendors such as Samsung and HTC configure the Android operating system as a customized product variant for their devices, e.g., the user interface on Samsung devices differs from the user interface of HTC devices. Hence, SPL techniques are used to abstract from complex aspects of a system, i.e., the variability of a product line, in a systematic manner. Therefore, these techniques seem to be a suitable approach to manage the runtime behavior of self-adaptive software systems. However, an SPL is used for systems that remain static in its composition of features at runtime. Prior to runtime, a software product is derived and once the product is derived it may not be further adjusted. In contrast to that, a self-adaptive software system needs to be continuously adjusted at *runtime*.

Dynamic Software Product Lines. The engineering of Dynamic Software Product Lines (DSPLs) enhances SPL engineering by allowing a product to be not only *configured* once at design time. Instead, a DSPL supports flexible *reconfigurations* at runtime [BSBG08]. This enables a product implementation to dynamically evolve and meet continuously changing requirements of the context [BHS12]. A promising field of application for DSPLs constitutes the vital domain of adaptable mobile devices [HPS12] such as (Android) smartphones. For instance, if a flash-crowd of people emerges as a context at runtime, e.g., on a concert, a transition from a configuration relying on an *infrastructure*-based communication to a new configuration, which uses an *ad hoc*-based communication might become necessary [ICP+99]. However, depending on the variability

constraints imposed on the system, such an adaptation may involve more than a single adjustment from *infrastructure*-based communication to an *ad hoc*-based communication. For example, both communication paradigms rely on different routing protocols, e.g., an *infrastructure*-based communication requires the Border Gateway Protocol (BGP) [RLH06] and *ad hoc*-based communication requires a Location Aided Routing (LAR) protocol [KV00]. Such interdependencies between features may become very complex and include a multitude of features, e.g., location aided routing requires the Global Positioning System (GPS) to be active. These constraints are specified in a feature model and are used as adaptation knowledge to derive a system reconfiguration that satisfies the ever changing requirements imposed by a contextual situation at runtime.

In this regard, DSPL techniques are applicable to specify the runtime variability of a self-adaptive software system. However, to achieve an autonomous adaptation, a transition system may be used to specify when a reconfiguration is to be executed.

TRANSITION SYSTEMS. Another kind of model, which may be used to specify the adaptation behavior of an autonomous system, are transition systems. Recent research on DSPLs proposes model-based approaches for pre-planning reconfiguration scenarios at design time. Those reconfiguration scenarios are specified as a transition system [Hel12, DPS12, WDSB09], which is deployed as adaptation knowledge on the device. Thereby, a state represents a device configuration and transitions specify reconfiguration options. Thus, a transition system models the reconfiguration behavior of a DSPL at runtime and further provides the means to analyze such reconfiguration behavior, e.g., identify deadlocks or configurations that may never be active at runtime. However, a complete transition system of a DSPL may become very complex since for every possible configuration the transition system contains the corresponding configuration state. For example, under the assumption that there are no constraints between features, a DSPL that consists of 20 features has $2^{20} = 1,048,576$ different possibilities to configure the DSPL.

1.1 PROBLEM STATEMENT

I chose the domain of mobile devices as a candidate to validate my research. This domain is suited for variability modelling techniques due to the high degree of similarities among different systems. For example, a different product configuration of the Android operating system is required for each of the devices of the Goggle Nexus product line, e.g., the Nexus 4 smartphone, the Nexus 7 tablet etc. Further, mobile devices benefit from autonomous self-adaptive software systems since autonomy eases the usage of a device and improves the user experience. Note that, although I chose a particular application domain for my research, my research is not limited to the domain of mobile devices.

To derive an autonomous self-adaptive software system on the basis of DSPLs, I identified the following four problem statements as important research issues.

(1) Context-Awareness of a DSPL. Existing DSPLs approaches do not yet support awareness of the contextual situation of the system. Without awareness the DSPL is not capable to autonomously reconfigure itself according to the ever changing contextual situations. To establish an autonomous self-adaptive software system based on a DSPL, such an awareness has to be established and seamlessly integrated into the modelling techniques of a DSPL [OGC12].

(2) Resource Consumption of a Reconfiguration at Runtime. Mobile devices have to cope with resource constraints that drastically restrict computational runtime capabilities for complex reconfiguration planning and execution tasks [BHS12]. Existing approaches fail to handle context-aware adaptations of resource-constrained devices due to two reasons. First, it is not possible to deploy the complete configuration space [Hel12] of a complex system onto the device due to limited memory. Second, it is not possible to dynamically explore the configuration space on-demand [FFF+13] at runtime due to limited processing capabilities. Thus, neither a complete state space nor an on-demand reconfiguration is feasible for mobile devices, and it remains open how both approaches may be combined to leverage the benefits and mitigate the drawbacks of both approaches.

(3) Flexibility at Runtime. Following DSPL techniques, every change in the configuration state of the system requires the derivation of a new product configuration w.r.t. the constraints specified in a feature model. Minor changes that occur at runtime change the operational state of the system. However, such minor changes do not necessarily require the derivation of a new product configuration [HPS12]. For example, the reconfiguration from a GSM-based connection to a WLAN-based connection does not affect the configuration of completely independent features such as the selection of a keyboard layout. Up to now, this issue has not been addressed.

(4) Reduction of Operational Cost. Recent generations of mobile devices are very feature rich and, therefore, permit a multitude of possible configurations. Thereupon, reconfigurations at runtime allow those devices to continuously adapt themselves to ever changing environmental context. For each change in the contextual situation of a device, there may be multiple configurations, which satisfy the requirements of a context. The operational cost of a self-adaptive software system may be reduced by choosing the cheapest configuration. Since self-adaptive software systems are permanently executed, those configurations have to be chosen on a long-term basis to achieve a significant reduction of operational cost. However, to this end, it is unknown how such a configuration has to be chosen to reduce the operational cost at runtime on a long-term basis.

Summarizing these research issues, this thesis aims to satisfy the goals of

- establishing an autonomous, model-based adaptation process (G1) and
- reducing the resource consumption of an adaptation at runtime (G2).

The next Section introduces the contributions provided in this thesis to achieve the two goals G1 and G2.

1.2 CONTRIBUTIONS

My research shows that runtime adaptations are specifiable by leveraging variability models and transition systems at runtime as adaptation knowledge. According to my categorization of problem statements (1) to (4), I provide four fundamental contributions in this thesis.

FORMAL FRAMEWORK FOR CONTEXT-AWARE DSPLS. In order to provide a model-based methodology to specify an autonomous self-adaptive software system I develop a formal framework and define (re-)configuration semantics of a DSPL. Therefore, I specify feature model constraints as propositional logic formula, as suggested in [Bat05, CW07], and extend it to be applicable to the domain of three-valued logics. However, in contrast to existing approaches, e.g., on context modelling [ACG09, LY04, BBH+10], contextual requirements are seamlessly integrated into the specification of a DSPL. In this regard, a context-feature model is introduced to autonomously reason about reconfiguration choices at runtime. This has the crucial benefit that traditional (D)SPL techniques are still applicable with my concept of a context-aware DSPL.

Further, a context-feature model provides the means to automatically derive a transition system. Such a transition system allows to (i) specify the reconfiguration behavior in detail and (ii) analyze the reconfiguration behavior at runtime.

REDUCTION OF A FEATURE MODEL. As previously stated in the problem statements, applying DSPL-based methodologies to devices that are limited in their resources is a difficult task. Specifying constraints using a feature model requires a constraint solver to calculate a configuration that matches both the specified constraints and the requirements. The more features and constraints a feature model contains, the harder the computation of a configuration. Thus, every contextual change utilizes available resources, such as draining the battery.

Therefore, I establish an approach to reduce a feature model specification w.r.t. the capabilities of a device. This improves the resource consumption of a DSPL oriented configuration process at runtime. To achieve such a reduction, I remove features that have to be permanently active at runtime, e.g., the display driver on a smartphone, or are incompatible to the capabilities of a device, e.g., features that rely on a cellular communication on a tablet that only supports WLAN. In contrast to existing approaches [ACLF11, RSA11], my approach focuses on maximizing the amount of features that are reconfigurable at runtime.

CONTEXT-SPECIFIC REDUCTION OF THE CONFIGURATION SPACE. A self--adaptive software system for mobile devices has to be responsive and should consume as less resources as possible. Due to the memory consumption and utilization of computational resources, it is neither possible to deploy the complete configuration space onto the device nor to dynamically explore the configuration

space on-demand at runtime. To overcome the deficiencies of existing approaches, I apply my formal DSPL framework to establish techniques to reduce the configuration space in addition to the reduction of a feature model specification.

The reduction of a configuration state space is based on the observation that users of mobile devices move in certain well-known contextual patterns [JL10, MSRJ12, Ver09]. The design of the transition system for controlling reconfigurations allows for tailoring *incomplete* configuration state spaces on the basis of context-aware reduction criteria. This constitutes a trade-off between comprehensively pre-planned configurations and on-demand evolutions of the configuration space at runtime. In addition, relying on a partial Kripke Structure [BG99] as interpretation of a transition system allows the usage of *partial* states. A partial state is applicable to (i) further reduce the memory consumption by subsuming comparable configuration states as well as (ii) gain flexibility in the configuration states, thereby, avoiding unnecessary reconfigurations.

Prediction of Contextual Changes. The well-established *Planning as Model Checking* paradigm provides a flexible, cost-sensitive approach for choosing among multiple reconfiguration options in a transition system. However, existing approaches [CRT98, GT00, PT01] lack (i) to combine fine-grained planning information comprising explicitly quantified cost models, e.g., for energy consumption and predicted contextual changes, and (ii) to cope with the limited resources of mobile devices. To tackle these drawbacks, a novel planning framework based on model checking techniques is proposed. Therefore, the transition system describing reconfigurations is enriched with the probability for a reconfiguration to occur and cost of a reconfiguration. With such a model it is possible to determine reconfigurations with minimum cost w.r.t. estimated long-term contextual changes emerging at runtime.

1.3 Outline

Figure 1.3 shows the road-map of this thesis. The complete thesis consists of nine chapters. A brief variant highlighting the key concepts and contributions given in this thesis consists of three core chapters. The complete nine chapters provide a story-line as follows.

Chapter 2. This chapter presents the fundamental concepts and state-of-the-art approaches on self-adaptive software systems, SPLs, and DSPLs. It provides the reader with a basis to understand the overall thesis. In addition to that, a running example is introduced, based on a Google Nexus product line.

Chapter 3. The third chapter introduces how a DSPL and self-adaptive software systems are combinable. Further, the chapter provides a detailed overview on the contributions of this thesis.

Chapter 4. This chapter introduces the fundamentals for my formal framework to specify an autonomous, context-aware DSPL. In this regard, the configuration semantics of a traditional SPL is stepwise extended to establish the configuration semantics of a context-aware DSPL.

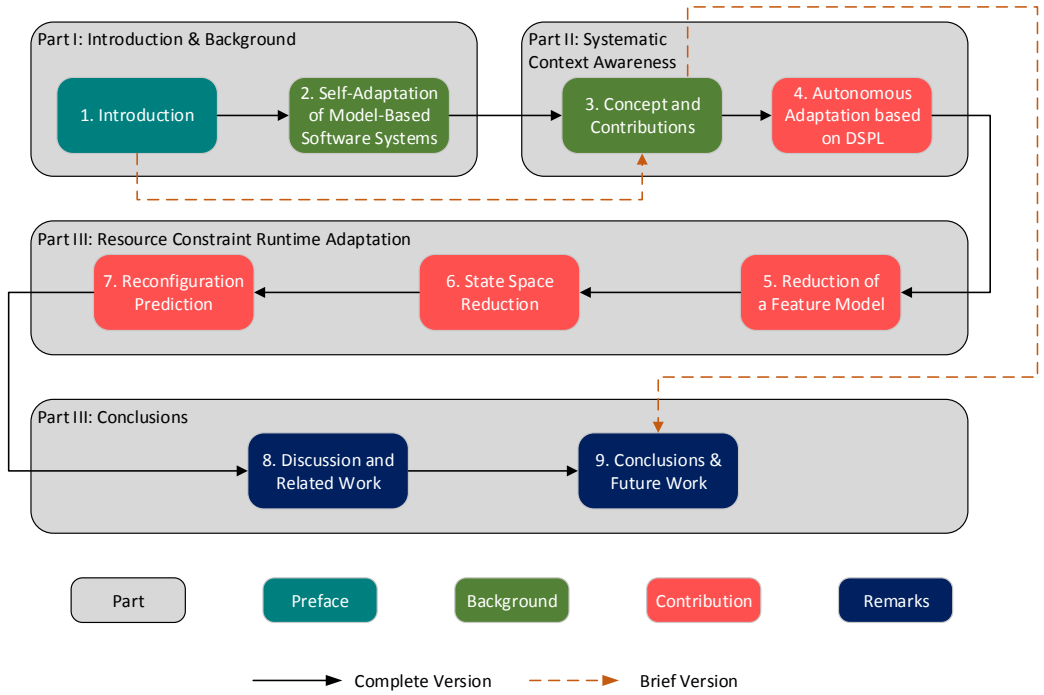


Figure 1.3: Road-Map of this Thesis

Chapter 5. My first approach to reduce the computational efforts of a reconfiguration at runtime is introduced in the fifth chapter. To achieve such an improvement, the variability specification of a DSPL is reduced according to the individual characteristics of a device.

Chapter 6. In the sixth chapter my concept of a DSPL reconfiguration based on a transition system is introduced. Further, techniques to reduce the resource utilization of such a transition system are investigated. This chapter also investigates possibilities to achieve a certain flexibility in the configuration states of a DSPL by using abstraction techniques.

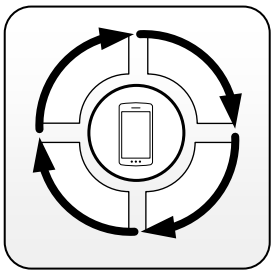
Chapter 7. My concept to reduce the operational cost of a DSPL-based self-adaptive software system is discussed in the seventh chapter. This chapter briefly introduces Model Checking as a tool to predict the reconfiguration cost w.r.t. possible upcoming contextual changes.

Chapter 8. This chapter provides an overview on relevant approaches that have been proposed to support runtime reconfigurations. These approaches are classified according to the concepts and techniques introduced in Chapters 4 to 7.

Chapter 9. The ninth Chapter concludes the thesis by critically discussing the results of the provided contributions. Further, future research topics are proposed to tackle the identified limitations of this thesis.

2

SELF-ADAPTATION OF MODEL-BASED SOFTWARE SYSTEMS



This chapter introduces the background of this thesis. The focus of the first part is on the introduction of the application domain *Self-Adaptive Software Systems (SAS)*. Starting with an overview of self-adaptation at runtime and how an adaptation is processed, related notions are explained and introduced based on a running example. SAS are used in numerous domains such as peer-to-peer or robotics. However, the focus of this thesis is an *autonomous adaptation* of *mobile devices*.

The second part introduces a model-based approach to handle runtime adaptivity, called *Dynamic Software Product Lines (DSPLs)*. The concepts of a DSPL are based on the specification of variability constraints, which may be used to manage runtime-adaptations in a systematic manner. Therefore, an overview of DSPLs is given about their origin, which modeling concepts are commonly used to specify a DSPL, and how a DSPL processes a reconfiguration at runtime. Finally, open research issues in the domain DSPLs are pointed out.

2.1 Self Adaptive Software Systems

The current explosion of information, technological progress, and distribution of software-intensive systems, leads to large-scale systems with an inherent degree of complexity. Especially the continuous growth [TYH+13] and development within the domain of mobile devices demand for innovative approaches for building, running, and managing such software systems. A consequence is the need of a continuous evolution of the respective software. These systems have to become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing. These aspects are achievable by an adaptation to the dynamically changing context, i.e., the environment or system characteristics [CLG+09]. Therefore, the domain of Self-Adaptive Software Systems (SAS) deals with the development of systems, which are able to adjust their behavior in response to an ever changing context.

> "Whenever the system's context changes the system has to decide whether it needs to adapt. [...] we consider context as any information, which is computationally accessible and upon which behavioral variations depend." [CLG+09]

This quotation includes the key aspect of SAS and also provides a definition of a context. SAS have to cope with the inherent dynamics of their context. To satisfy certain specifications and goals, SAS adapt themselves accordingly, whenever the context changes and requirements are violated. In this thesis, a *context* describes information, which are computationally accessible, such as derived information about the environment, e.g., *office-location* or *rainy-day*, about the users, e.g., *listing to music* or *reading mail*, and about the system itself, e.g., *low-battery* or *silent-mode*. To satisfy stated goals the SAS has to be flexible and derive a suitable adaptation from this contextual information.

Self-adaptation is an important research topic in several application domains, such as autonomic computing, dependable computing, embedded systems, mobile ad-hoc networks, mobile and autonomous robots, multi-agent systems, peer-to-peer applications, sensor networks, and service-oriented architecture [Bro10]. In all these domains *flexibility* at runtime is essential. However, little endeavor has been made to establish a systematic software engineering approach to handle the inherent dynamics at runtime [CLG+09].

In the following an overview and classification for self-adaptive software systems is given. Additionally, the paradigm Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) is introduced as a control loop to regulate SAS. Thereupon, research challenges that are addressed in this thesis and limitations of this thesis are discussed.

2.1.1 *Adaptivity in Mobile Devices*

As mobile devices, e.g., smartphones, tablets, or notebooks, become more and more integrated into the everyday living of a person, mobile devices are exposed to continuous changes in their environmental context. This is especially the case for smartphones, because most people have their smartphone with them when they are mobile and change their *location*. Naturally, every *device* is differently constrained in its *capabilities* due to its mobility, production costs, and user-specific requirements, e.g., lightweightness. Thus, it depends on the capabilities of a smartphone how the contextual environment may be exploited, e.g., a WLAN chip has to be integrated in the device to connect to a WLAN access point in the contextual environment. However, the capabilities of a smartphone may change over time, e.g., the battery is continuously drained at runtime or the device may be upgraded. An additional category, which influences the adaptation of a smartphone, are the *activities* the user imposes on the device itself, i.e., the usage of the device. Every contextual category states certain requirements and together they restrict the possibilities in which a device is able to adapt itself.

As depicted in Figure 2.1, if the user sits at *home*, is *reading* an eBook, and the smartphone reaches a *low battery* threshold, the device has to find a suitable configuration within the highlighted plane that satisfies all contexts. Note that the contextual categorization *location*, *device*, *activities* is a matter of granularity. Therefore, additional contextual categories are imaginable, e.g., *network*, *service*, *quality*, or *social* context [PNS+00, RDN06].

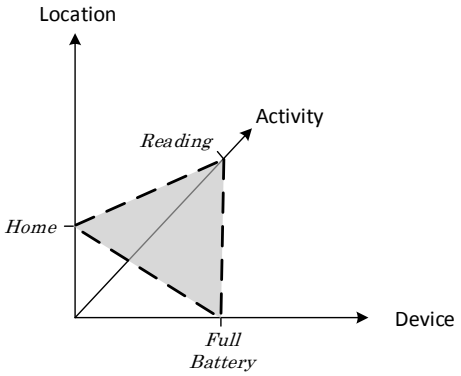


Figure 2.1: Adaptation Dimensions of Mobile Devices

LOCATION. Depending on the contextual location, smartphones provide the means to access a large variety of available ubiquitous services. In recent years, the availability of these ubiquitous and mobile services have significantly increased due to the different form of connectivity provided by mobile devices, e.g., WLAN access points, cellular networks, or GPS navigation [Yan12]. To provide an optimum of service functionality smartphones have to exploit their surrounding environment, e.g., using a WLAN connection if an access point is available. Thus, a smartphone has to cope with the mobility of its user. Figure 2.2 depicts this problem. A user expects different functionality or services, i.e., *goals*, depending on the contextual situation of the smartphone.

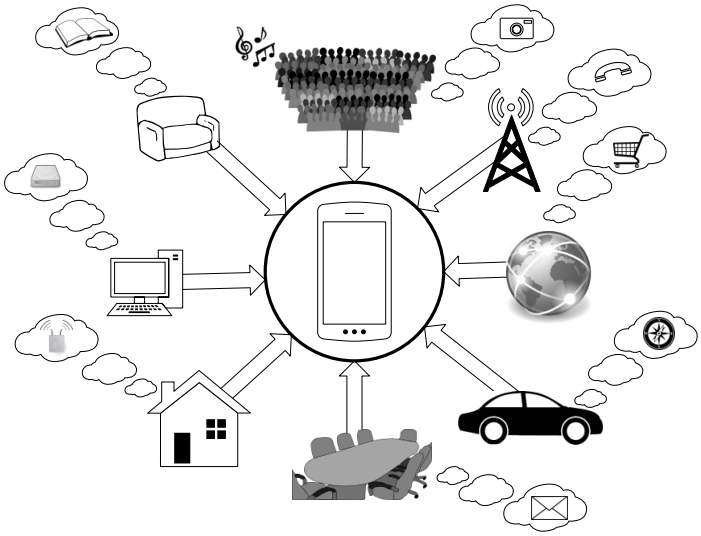


Figure 2.2: Contextual Situations of a Mobile Devices

Example 2.1 (*Impact of Mobility on a Smartphone*).

Figure 2.2 depicts several contextual situations and in each situation the user expects different functionality or a different behavior. If the smartphone is connected to a *desktop Personal Computer (PC)* the smartphones becomes an external hard drive. An example for a very location sensitive situation is driving in a *car*, where the smartphone becomes a navigation device. Thus, the *goal* of a

smartphone may change in different contextual situations. An example for a cost-optimization is the change of situation where the user is within the area of a WLAN access point, e.g., *at home*, or all communication is processed via a *cellular network*. Another requirement emerges if the user is in his *office*, where he has to be able to receive and send emails.

A *mobility pattern* of the user may be as follows on a working day: *"home → car → office → car → home → couch → desktop PC"*. However, such a mobility pattern on a weekend day will most likely be very different and include different contextual situations, e.g. going to a *concert*.

This example shows that mobile devices, and especially smartphones, are exposed to continuous changes of contextual situations. These situations occur based on certain *mobility patterns* of the individual user.

Device Capabilities. A smartphone is only able to access numerous ubiquitous services, which are provided by a certain environment according to its own capabilities. These capabilities are either static or change over the lifetime of the device. The Central Processing Unit (CPU) of a smartphone is an example for a static capability, whereas the battery load and available storage space are continuously changing at runtime.

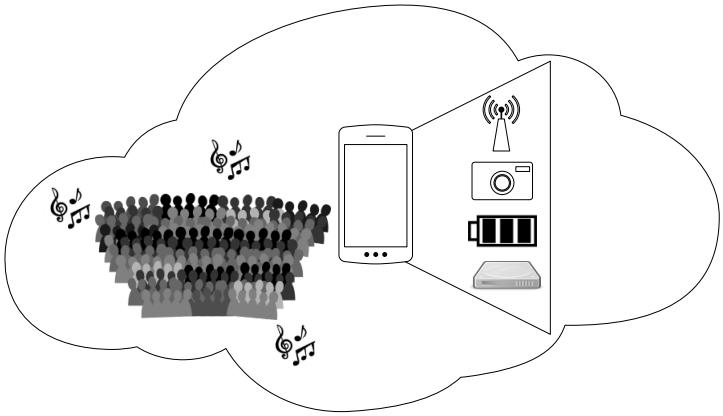


Figure 2.3: Required Capabilities of a Smartphone at a Concert

Example 2.2 (*Capabilities of a Smartphone*).

According to Figure 2.2 a user wants to take pictures and probably publish them via a social network if he is on a *concert*. However, this is only possible if the smartphone is able to satisfy the requirements (i) the smartphone has a camera integrated to take a picture, (ii) the smartphone supports the available communication standard, e.g., WLAN, GSM, or LTE, to share the pictures, (iii) there is enough storage space available to store the pictures, and (iv) the smartphone has sufficient energy resources available to execute the tasks.

User Activity. Independent of the external contextual environment of a smartphone the user may request certain functionality or services on-demand. In this

case, the smartphone has to respond accordingly and provide the functionality or service according to its capabilities and external ubiquitous services.

Example 2.3 (*Usage Impact on a Smartphone*).

If the user wants to do some on-line shopping, a connection to the Internet has to be established. However, depending on the contextual environment and local capabilities, a connection is establishable in different ways, e.g., via LTE or WLAN. Depending on the user preferences, the smartphone has to choose between the available connection types, e.g., if the user prefers a cheaper connection, the smartphone would choose WLAN instead of LTE.

Summarizing, independent from the mobility patterns of a user, the device is only able to adapt according to its capabilities. The expectations of a user, i.e. the *requirements*, depend on the environment and ubiquitous services or interfaces, which are provided. If there is a change in the context of the smartphone, the intentional goal of the smartphone may also change, e.g., the smartphone becomes an external hard drive if it is connected to a PC and it becomes a camera if the user is at a concert. Thus, to optimize the convenience of a user, smartphones have to change, i.e. *adapt*, themselves automatically to satisfy user-specific and context-specific requirements.

This overview shows that mobile devices, such as smartphones, are self-adaptive software systems. The next section provides detailed insights into the overall adaptation process of SAS by describing key technical aspects and characteristics of these systems.

2.1.2 *Classification of Self-Adaptive Software Systems*

The application of a self-adaptation depends on various aspects, e.g. the changing needs of a user, environmental characteristics, and system properties. Understanding the problem and selecting suitable solutions require precise models to represent environment, users, and the system [CLG+09]. This section introduces a classification scheme taken from [CLG+09, ALMW09] to describe the most important facets of SAS. The classification is divided into the four groups (i) *goal* of an adaptation, (ii) cause of a *change*, (iii) adaptation *mechanism*, and (iv) *effects* of an adaptation.

GOALS. Goals are objectives the system has to achieve or constraints that have to hold at runtime [LLYM06]. Such goals may be functional, e.g., *"activate GPS during a navigation process"*, or non-functional, e.g., *"response time for a GPS signal has to be below 2 seconds"*. Existing approaches, e.g., goal models [LY04], propose to use quantifiers and hierarchical decomposition into sub-goals to model complex goal structures for SAS. An example goal for mobile devices may be *"always provide a connection to the Internet"* and a sub-goal may be *"prefer WLAN-connection to a cellular-connection"*. Important aspects for goals of an SAS are

- **Evolution.** Depending on the context of a device the goal specification of a system may change dynamically at runtime. However, the specification of a

goal may also change in a static manner at design time if the SAS is changed, e.g., goals are exchanged during a system update.

- **Flexibility.** Goals do not necessarily specify a rigid system but may be also flexible. The more rigged a goal is the less system configurations are available, which satisfy the requirements of that goal. In that manner, a goal may be rigid, constrained, or unconstrained. A flexible goal is able to handle contextual situations, which are not specified explicitly at runtime.

- **Dependency.** If a system has to meet multiple goals, these potential goals may be related to each other. Thus, goals may be complementary or conflicting to each other. Alternatively, they may also be independent from other goals, i.e., they do not affect each other.

Example 2.4 (*Goals of SAS*).

The goal *"activate flash at night when camera is active"* is the static initial specification of a goal at design time. At runtime, this goal becomes active whenever the user activates the camera during a concert and, therefore, photos are taken with flash. With an update a new filter is deployed on the SAS and with the filter also the goal changes to *"activate dark-filter at night when camera is active"*.

A *dynamic evolution* of a goal occurs at runtime. For example, a goal states *"use GPS during navigation"*. Whenever the smartphone realizes that a GPS signal is not accurate and localization is more precise with an additional cellular triangulation, the goal is updated dynamically at runtime to *"use GPS and cellular triangulation during navigation"*.

The goal *"activate dark-filter at night when camera is active"* is flexible because it still supports dynamic adaptations. For example, there are no restrictions regarding using *flash* or *GPS* while taking a photo. The less requirements a goal specifies the more flexibility is provided to handle additional or even unknown contexts, emerging at runtime.

The goals *"use GPS during navigation"* and *"activate flash at night when camera is active"* are independent of each other. However, the goal *"deactivate all communication signals at night"* contradicts the goal *"use GPS during navigation"*, which implies that a smartphone may not be used to navigate at night.

Change. An adaptation is triggered by changes in the context of a device. If the context changes the requirements for the device also change. Thus, the system has to adapt itself accordingly to re-satisfy the new requirements. To plan a suitable adaptation, information on location, type, and frequency of a change, is important to anticipate the occurrence of a change.

- **Source.** A change may occur external to the device, i.e., within the environment, or internal within the device, i.e., within the system. Internal changes are investigateable in more detail, whereas the detection of external changes relies on the information of sensors.

- **Type.** The type of the requirement change is either functional, i.e., a functionality is required or prohibited, or non-functional, i.e., a the system has to satisfy a certain level of quality.
- **Frequency.** The frequency of a change determines the number of changes over a certain period. The scale may range from once a week over several times per minute to numerous times per second.
- **Anticipation.** Adaptations are predictable. Such predictions may be used to pro-actively prepare an adaptation. However, different adaptation strategies are necessary, depending on the degree of anticipation. If an adaptation is foreseeable, the adaptation is still uncertain, though likely to occur. Therefore, the system has to plan and reason about such an adaptation. In contrast to that, an adaptation may also be unforeseeable, i.e., unknown or spontaneously occurring, in which case the system is not able to plan an according adaption [JL10].

Example 2.5 (*Change in SAS*).

An example for an *external* trigger to adapt the system is the movement of the user. If the user leaves the context of his home and enters the context of his car, the smartphone adapts itself by deactivating WLAN and activating GPS and the navigation system. An example for an *internal* trigger to adapt the system is the remaining amount of energy left in the battery. If the remaining energy is low, all media related applications are closed and are prevented to be opened.

A *functional* goal refers to provided services or functionality of the system. An example for a functional goal is *"deactivate WLAN connection in a car"*. In contrast to that refers a *non-functional* goal to the qualitative aspects of a system. For example, the goal *"the responsiveness has to be below 1 second"* is a qualitative aspect of a connection.

On a standard working day the user enters and leaves his office once, thus the *frequency* of these changes is once a day.

If the user has a repeating mobility-pattern, an adaptation may be *anticipated*. For example, if the user always follows the pattern *"home → car → office → car → home"* the smartphone is able to predict the according adaptations with a certain probability. In that example, the smartphone prepares the adaptation to the context *car* if the user is still in the context *home*. However, if there is an unforeseeable event and the car of the user breaks down, the mobility pattern changes, e.g., *"home → car → street"*, and the smartphone has to adapt whenever the unforeseen contextual change occurs.

MECHANISM. The implemented mechanisms capture the reaction of a system towards a change. Therefore, the adaptation mechanism represents the core of an adaptation process. Adaptation mechanisms cover different levels of autonomy and there are several different possibilities to control an adaptation. Additionally, the impact of an adaptation mechanism has to be considered.

- **Type.** An adaptation is related to either changing the parameter configuration of the system or to changing the structure of the active system implementation, i.e., exchanging or rewiring its components. Therefore, an adaptation is either structural, parametric, or a combination of both.

- **Autonomy.** Existing terminology states that an adaptation is either executed autonomous, assisted, or autonomic. An autonomous adaptation process is executed without external influences, besides the contextual information. An assisted adaptation is executed with the support of an external user, e.g., if an adaptation is not executable automatically. Autonomic adaptations are also executable without any external influence. In addition to autonomous adaptations, an autonomic adaptation process is able to learn and to derive new adaptation strategies dynamically at runtime [SSH06].

- **Organization.** Adaptations are either executed centralized on a single system, or are distributed over several interacting systems, e.g. peer-to-peer systems, in a decentralized manner. A decentralized adaptation is not controlled by a single component or system. Instead, decisions have to be made collaboratively in a network of systems. Similarly, an adaptation is either restricted to local adaptations, i.e., one single system, or to global adaptations, i.e., within a network of systems.

- **Duration.** Depending on the domain, the duration of an adaptation may be a critical aspect to deal with. An adaptation may last from milliseconds up to several hours. However, this characteristic has to be considered relative to the application domain.

Example 2.6 (*Adaptation Mechanisms of SAS*).

If a smartphone is connected to a desktop PC, the smartphone becomes an external harddisk. This is an example for a *structural* adaptation because a specific driver is activated on the smartphone to turn it into an external harddisk. If the smartphone has to adapt to a silent mode, the ringtone and other notification signals become silent. This is an example for a *parametric* adaptation because the volume parameter is adjusted to zero.

An example for an *assisted* adaptation is the manual selection of the user to communicate via WLAN instead of LTE. In this case, the adaptation is manually triggered and requirements are specified by the input of the user. The automatic activation of GPS and starting of the navigation system whenever the user enters a car is an example for an *autonomous* adaptation. In this case, the smartphone adapts itself automatically based on the contextual information. If the smartphone recognizes that a connection via LTE is better, e.g., more responsive, than a connection via GSM and triggers an adaptation for this reason, the smartphone is capable for *autonomic* adaptations. In this case, the smartphone is able to learn and optimize the overall adaptation process by changing adaptation goals or creating new adaptation goals on its own.

Until this point of this thesis, the given examples for an adaptation are intended for a single systems, i.e., *centralized* on a single smartphone. However, if

data has to be exchanged between two smartphones, the smartphones have to reason about the protocol, e.g., Secure Copy Protocol (SCP) or File Transfer Protocol (FTP), and type of communication, e.g. Bluetooth or WLAN. They have to agree on one single solution within the capabilities of both devices. This is an example for a *decentralized* adaptation because the adaptation has to be organized between the participating smartphones without a central smartphone that makes a decision.

Starting the navigation system of a smartphone after arriving in a new country may trigger the download of the required street maps to perform a valid adaptation. In this case the adaptation may last for several minutes or hours, depending on the available connection and the amount of data. Although this is not safety critical, e.g., it does not lead to a traffic accident, it may lead to inconvenience if the user is impatient.

EFFECTS. Every adaptation has an impact on the system it adapts. While the *mechanism* category deals with the adaptation itself, the listed effects are associated to the overall SAS, for which the adaptation is performed.

- **Criticality.** The criticality of adaptations describes the impact of erroneous adaptations. This ranges from harmless to goal-critical or safety-critical.
- **Predictability.** Another important aspect are effects of an adaptation on the system and whether they are foreseeable. An adaptation may perform as expected, in which case guarantees may be specified. However, an adaptation may further perform differently each time it is executed, e.g, if the adaptation is *flexible* regarding the result. In that case, only limited or none guarantees are specifiable.
- **Overhead.** The overhead deals with the additional negative impacts on the overall system performance during an adaptation. An adaptation constantly has to be non-intrusive to the system and, therefore, should always consume a minimal amount of memory and computational power [MRD08]. In the worst case, an intrusive adaptation leads to thrashing and, thereby, a system ceases to provide its services [BMSG+09].

Example 2.7 (*Adaptation Effects on SAS*).

The adaptation *"activate the navigation system"* fails if the smartphone has not enough battery available to support GPS and the navigation. In that case the adaptation fails to complete. The failing of this adaptation is non-critical if the user is not yet driving. However, if such an adaptation fails while the user is driving his car and depends on the navigation system, a failure becomes more *critical* because it distracts the user.

The adaptation for the contextual requirement *"deactivate all communication signals at night"* results always in the same system configuration where all communication related aspects of the smartphone are deactivated. The more flexible requirement for the contextual requirement *"activate dark-filter at night when camera is active"* may not always lead to the same system configuration. The

requirement specifies no restrictions regarding the activation of additional filters, such as a black-white filter. Therefore, one adaptation may activate the black-white filter and the next adaptation for that context may not activate that filter. In that case, the effects of the adaptation are not *foreseeable*.

Every adaptation consumes resources. For example, every context has to be identified, e.g., by using the sensors of the smartphone to identify if the user enters a car. Thereupon, the adaptation mechanism has to compute, which changes are to be made and how the requirements are to be satisfied, e.g., GPS has to be activated and the navigation system has to be loaded. Finally, the adaptation has to be executed, e.g., GPS is being activated and the navigation system is being loaded. The more resources such an adaptation consumes, the more *overhead* it generates. If the adaptation consumes most of the available resources, other services, e.g., receiving emails, may be suspended.

The adaptation aspects of *goals*, *change*, *mechanism*, and *effects* are used to describe and to evaluate the capabilities of an SAS. Although this classification provides a holistic overview of the concept behind SAS the matter of a technical implementation is still unaddressed. Therefore, the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) paradigm is introduced in the next section, as a common approach to handle the inherent dynamics of mobile devices at runtime.

2.1.3 *The MAPE-K Feedback Loop*

The decisions *when*, *what*, and *how* to adapt a mobile device have to be made by the SAS at runtime to cope with dynamic behavior of a user. To facilitate systematic control of the adaptation process, the system continuously reasons about its contextual situation. For example, a mobile device collects contextual information, e.g., its geographical location and available connection types, to reason about how to establish a connection to the Internet in this contextual situation.

Feedback loops are a common paradigm to facilitate an adaptation process at runtime in a coordinated, reliable manner [BMSG+09, CLG+09]. The feedback enables a static or dynamic evolution of the system and is used to optimize the adaptation behavior. Positive feedback occurs when an initial change in a system is reinforced. In contrast, negative feedback triggers a response that counteracts an executed decision.

Example 2.8 (*Positive and Negative Feedback*).

According to some non-functional requirement, the connection with the best throughput is to be preferred. Initially starting with a HSDPA connection, the smartphone continuously checks for other connection types. If GSM becomes available, the smartphone tests this connection type as an alternative. Since GSM has a lower throughput than HSDPA, the adaptation process emits a negative feedback for that decision. The same happens if LTE becomes available. In contrast to GSM, the adaptation process emits a positive feedback because the throughput via LTE is higher than via HSDPA. Therefore, the system is going to prefer LTE over HSDPA automatically in upcoming adaptation decisions.

The basic principle of such feedback loops is a refinement of the sense-plan-act approach used in the AI community [Nil80]. With a feedback loop the adaptation process may become completely autonomous.

The first concrete architecture of a feedback loop for autonomous SAS was introduced in a blueprint of IBM [IBM06]. This feedback loop is depicted in Figure 2.4. The loop facilitates a controlled autonomous adaptation by executing the four tasks (i) *monitor*, (ii) *analyze*, (iii) *plan*, and (iv) *execute*. These tasks share a common *knowledge* basis. This approach is commonly known as the *MAPE-K loop*.

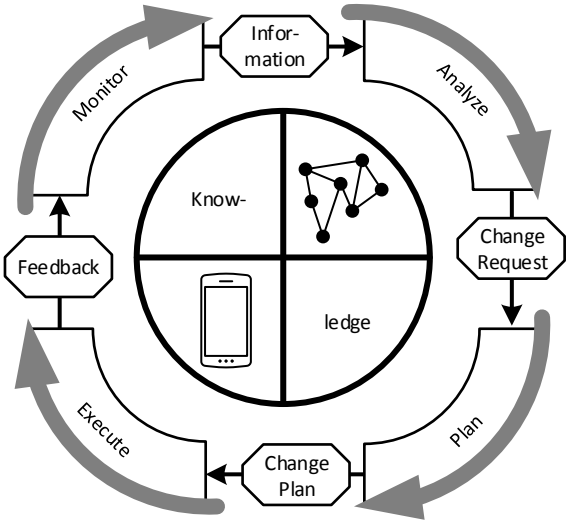


Figure 2.4: MAPE-K Feedback Loop [IBM06]

MONITORING. The monitoring task is responsible for capturing basic information of the contextual situation, e.g., via sensor data. This information is correlated to specific aspects, e.g., the current location and available connection types. Such information includes network topology information, property settings, status of resources, offered capacities, and throughput. The information is either static or the information is highly dynamic and continuously changes over runtime. A monitor aggregates this information, e.g., for a better information scalability [SGS13] in a large decentralized network of smartphones. Thereafter the information is filtered to the most relevant aspects for an adaptation before passing them to the analyze task. For example, assume that in a contextual situation only the location and throughput are of relevance. In that case, the information about responsiveness and lighting conditions are discarded.

ANALYZE. The analysis of the monitoring information provides the means to recognize contextual situations and to derive higher order information. Based on the analysis of that data a decision is met whether an adaptation is necessary or not. The gathered monitoring information about the current contextual situation is correlated to the requirements a context imposes on the system. For example, if the context *office* becomes active, the analysis task checks if the requirement *"able to send and receive email"* is already satisfied with the current system configura-

tion or if an adaptation is necessary. Additionally, the adaptation behavior may be analyzed to employ prediction techniques to plan and prepare for upcoming adaptations. Therefore, the device has to be able to learn about contextual situations and the adaptation behavior executed on the device. If the requirements change due to a change in the contextual situation, the analyze task passes an according change request to the plan task.

Plan. The plan task derives a procedure to execute a desired adaptation of the system to satisfy the imposed requirements. An adaptation procedure may range from a single command, e.g., *"activate WLAN"*, to a complex adaptation sequence, e.g., *"activate WLAN"*, *"deactivate GPS"*, and *"set ringtone volume to 0"*. Furthermore, the planning is driven by specific goals, i.e., functional or non-functional requirements, when an adaptation plan is derived. This ranges from a problem solution, i.e., satisfying all requirements and goals, to a complex optimization process to derive the best system configuration for a contextual situation. The derived change plan is passed to the execute task to apply all changes and reconfigure the system at runtime transparently.

Execute. In the execution task, suitable methods and mechanisms are scheduled to perform the changes in the current system configuration. These changes may range from parameter changes, e.g., ringtone volume, to the exchange of components or code fragments, e.g., discarding email notification component or changing the keyboard component from English to the German. The execute task is responsible for carrying out these changes within a running system in an ordered sequence of non-conflicting tasks.

Knowledge Basis. The four tasks *monitor*, *analyze*, *plan*, and *execute* share a common knowledge model. This knowledge includes specifications or profiled information required by the adaptation process. For example, the knowledge model includes contextual information, adaptation behavior profiles, monitoring metrics, adaptation policies, as well as system capabilities. The specified knowledge is either *explicitly* specified, i.e., everything that may happen is covered within the model, or *implicitly* specified, i.e., uncertain or unknown behavior may occur at runtime, which is not covered within the model. The knowledge model is implemented as a registry, dictionary, database, or any other repository that provides access to knowledge according to the interfaces prescribed by the architecture.

The knowledge is obtainable in three ways [IBM06].

1. Knowledge is a-priori specified at *design time* and deployed on the system locally before runtime,

2. the knowledge is continuously retrieved from an external knowledge source, e.g., an external database in the Internet, or

3. the adaptation process itself generates knowledge, e.g., leveraging monitoring information, planning decisions, and profiles of contextual changes. This information is either used to continuously update or extend existing knowledge at *runtime*.

The knowledge is representable via three different types [IBM06].

- **Solution Topology Knowledge.** Captured knowledge about components and their architecture, e.g., dependencies between components such as navigation system requires GPS.
- **Policy Knowledge.** A policy is knowledge that is used to decide whether an adaptation is necessary and which changes have to be applied. Thus, a policy corresponds to a set of constraints or preferences that influence the analyze task and planning task.
- **Problem Determination Knowledge.** Problem determination knowledge includes monitored data, symptoms and decision trees. This knowledge is used to further derive or update existing knowledge at runtime, i.e., via learning algorithms or profiling.

The main part of this thesis is focused on the *plan* task based on a *knowledge* model. Relation to the other tasks of *monitoring*, *analyze*, and *execute* are pointed out, but it is assumed that these tasks are provided as a black-box. The next section points out research challenges that are addressed in this thesis.

2.1.4 *Research Challenges in SAS*

The domain of SAS provides numerous challenges still to be faced in research. Therefore, this section lists specific challenges for SAS in the domain of *mobile devices*, which are addressed in the remainder of this thesis. The scope of this thesis is restricted by the introduction of certain limitations and assumptions. Therefore, the *Research Challenges (RCs)* of (i) a *systematic engineering*, (ii) *flexibility at runtime*, (iii) *autonomy*, and (iv) *intrusiveness* are introduced in the following.

RC1: SYSTEMATIC ENGINEERING.

> "A major challenge [...in the development of adaptation mechanisms ...] is to accommodate a systematic engineering approach that integrates control-loop approaches with decentralized agent inspired approaches." [CLG+09]

This quotation highlights the importance of a systematic engineering approach for SAS. This implies a consistent development methodology from requirement engineering, e.g., goal modelling, over an a-priori specification of the common knowledge basis, e.g., policies to describe the adaptation behavior, to the continuous update and extension of this knowledge at runtime. Since the usage of a feedback loop, i.e., MAPE-K, is state of the art to control SAS, a systematic engineering approach has to be integratable into such a loop. Although this quotation highlights the importance of a decentralized applicability, this issue is not addressed further in the remainder of this thesis.

RC2: Flexibility at Runtime. To cope with the dynamically changing contexts, an adaptation is specified explicitly or implicitly. In an explicit specification all adaptation possibilities are assumed to be known and integrated into the knowledge model. For example, an explicit specification covers all contextual situations depicted in Figure 2.2. Therefore, a mobile device is capable to adapt to all these contextual situations. However, if a *new* contextual situation emerges at runtime, e.g., entering an *airplane* or combining two existing contexts such as *home* and *couch*, the mobile device is not able to adapt to this context on its own. In contrast, an implicit specification is an incomplete specification and intends to cover a basic set of contextual situations with the ability to *extend* and *update* the knowledge at runtime. For example, the contextual situations *home* and *couch* are specified individually. If the specification is implicit, the mobile device is able to adapt to a combination of *both* situations either completely autonomous or with the assistance of the user. Hence, such a specification is more flexible and open to new contextual situations, emerging at runtime. However, the result of such an adaptation that is derived on-demand at runtime is not always predictable. An important research issue is to provide certain flexibility within the adaptation process to cope with unknown contextual situations [BMSG+09]. At the same time an adaptation has to be reliable and foreseeable [ALMW09], i.e., such that an adaptation exposes always the same behavior.

RC3: Autonomous Adaptation. An autonomous adaptation implies a process without any interaction with external actors, e.g., a user or developer. Therefore, the adaptation process has to evaluate the context and derive necessary adaptation strategies on its own [SSH06]. Especially for mobile devices the user experience is of importance. Hence, an adaptation of a mobile device should not bother the user. The adaptation has to be executed automatically and should be transparent to the user [Har06].

RC4: Non-intrusive Adaptation.

> "In highly dynamic systems, e.g., mobile systems, where the environmental parameters change frequently, the overhead of adaptation due to frequent changes in the system could be so high that the system ends up in thrashing. [However, ...] responsiveness is a crucial property in real-time software systems [...]" [CLG+09]

This quotation refers to the effects an adaptation process imposes on the system. The more resources are utilized within any of the MAPE-K tasks, the more likely the system starts thrashing up to the point of a complete system failure. Although these effects are mitigated by scaling the hardware resources of the system, e.g., more memory, bigger CPU, etc., such a solution is either costly or infeasible. Especially mobile devices are constrained in their resources, i.e., the battery will be drained eventually. Thus, SAS for mobile devices have to operate resource-efficient to cope with the high frequency of dynamic contextual changes.

RC5: Predictability of Adaptations. A prediction of contextual changes may be used to optimize the adaptation process, e.g, by pre-computing an adap-

tation or by harmonizing a sequence of adaptations. However, a prediction of an adaptation or a sequence of adaptations for a mobile device is complicated due to the broad spectrum of contextual situations and the spontaneous, unforeseeable behavior of a user [MSRJ12].

LIMITATIONS AND ASSUMPTIONS. In the remainder of this thesis I address the research challenges *RC1* to *RC5* with a specific focus on the *plan* task based on a common *knowledge* model. Note that the approach is designed to operate on resource constrained devices, i.e., mobile devices. In this regard, I neglect details about the *monitoring*, *analyze*, and *execute* tasks of MAPE-K but assume them as a provided black-box, which is accessible to retrieve certain information or execute certain tasks. Hence, I also do not explicitly address intersecting topics, e.g., *context reasoning*, *dynamic class loading*, *feedback evaluation*, and *information correlation*.

Mobile devices are heterogeneous in their characteristics, e.g., one device supports LTE whereas another device does not. To handle such heterogeneity in a systematic manner, software product lines are used. The next section introduces the domain of software product line engineering. Furthermore, an extension of software product line engineering is introduced that focuses on the systematic management of runtime adaptations.

2.2 Software Product Line Engineering

In traditional software engineering a software system was typically not regarded as being variable. Either each software system was developed for the needs of the customer individually or the customer bought a software system with all possible features. Since the implemented code of a software system is easy and cheap to copy, transport, and replace, a structured development process for similar software systems seemed not to be necessary. However, in recent years the situation in software engineering has changed. The complexity of software systems are more and more exceeding the limits of what is feasible to handle with traditional software engineering approaches [PBv05].

In a highly competitive and segmented market, such as the mobile device industry, mass production is not enough anymore and mass customization of the respective software systems is due to become a must for market success [BSRC10, Par76]. An example for this is the vendor specific Android operating system. Mass customization aims to meet as many individual needs of a customer as possible. In addition to that, the mass production has to be as efficient as possible. To maintain efficiency during the production, practitioners propose to build products from existing assets that share more commonalities than singularities. These assets, i.e., features of a product, have to be flexible in the common sense of variable to be reused in other software products in the same product line. This means that, for a structured engineering process of software product lines, features have to be identified and to which extent they differ, fulfill the same requirements, or share a similar underlying architecture.

Clement elevated Software Product Line (SPL) engineering to a dominant development paradigm [Cle99]. The goal of an SPL engineering process is to maximize

the reuse of software artifacts within the range of products sharing a common set of software characteristics [Bos01, CN01, PBv05]. SPLs are intended for the development of domain specific software products, which are individually tailored for the individual needs or problem statement of a customer based on the variability of a software system. The SPL concept is based on the idea to combine the advantages of custom tailored software and commercial off-the-shelf software. Custom developed software are intended to solve a specific problem. Usually, developing such software is expensive since the software is intended for a broad market. In contrast to customized software, off-the-shelf software is intended for a wide area of domains used by a broad spectrum of users.

Example 2.9 (*Software Product Lines*).

Commercial examples for software product lines are the Android Operating System for mobile devices or the Office suite from Microsoft Windows.

SPLs intend to develop software products for a certain domain. Thereby, SPLs address the needs and requirements of various users as well as provide the means for products that address specific problems or requirements from a user. Clements and Northrop define SPLs as follows

> "A software product line is a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [CN01].

This quotation provides the most basic information about SPLs:

- SPL shifts the focus from the development of single software products to a family of similar software products.
- Features are software related entities that in their individual combination define the final software product.
- SPLs are created for a specific domain, based on expert knowledge or related functionality.

Thus, SPLs are able to support developers to efficiently derive customized software products from a common repository instead of developing every product from the scratch.

This section introduces the fundamental concepts of SPL engineering. Thereafter, feature models are introduced and how they are used to specify the variable characteristics of an SPL. Finally, dynamic software product lines are introduced as an conceptual extension of SPLs, to shift the process of a product configuration from design time to runtime.

2.2.1 *Fundamentals of SPL Engineering*

The SPL engineering process consists of two main activities: (i) domain engineering and (ii) application engineering [CN01, vSR07, PBv05, WL99].

- **Domain Engineering.** This process is responsible for creating a repository of reusable software assets, i.e., software *features*, and identifying commonalities and variability of an SPL. The domain engineering process, depicted in the upper part of Figure 2.5, consists of five activities: product management, domain requirement engineering, domain design, domain realization, and domain testing. This engineering process results in a software repository that contains parts of the system in the form of reusable assets, such as requirements, design artifacts, implemented components, tests, etc.
- **Application Engineering.** The application engineering process, depicted in the lower part of Figure 2.5, deals with the *configuration* of each individual software product. The process consists of four activities: application requirement engineering, application design, application realization, and application testing. The application engineering results in a software product that is built by reusing assets developed during the domain engineering.

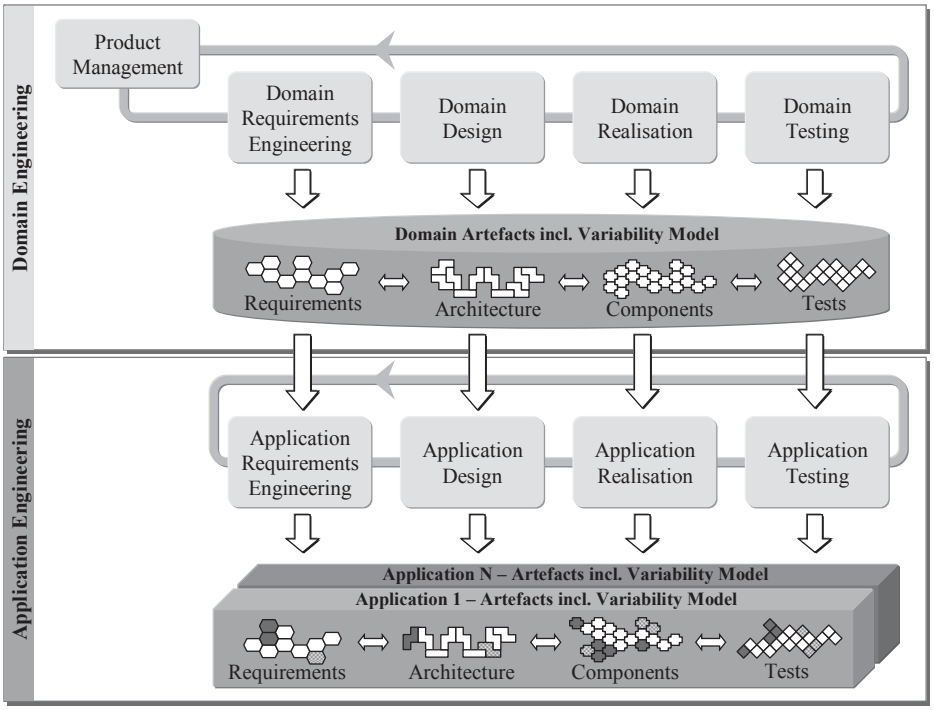


Figure 2.5: Software Product Line Engineering Approach [PBv05]

The expected positive effects of such an approach are an optimization of the maintenance of software, the overall software quality, development costs and time-to-market of individual products of the SPL [Bos01, vSR07]. For a detailed introduction into the engineering process of SPLs, I refer to [PBv05].

Example 2.10 (*Smartphone Products*).

Figure 2.6 shows two different products of the android operating system as a software product line. The products share common elements, which are aggregated in the building block `Stock Android`. However, the products differ

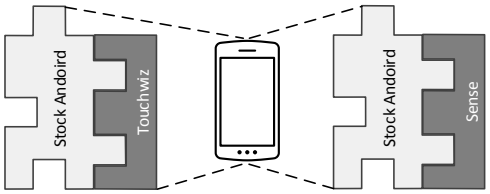


Figure 2.6: Two Product Variations of a Smartphone

in their user interface. The software product on the left-hand side uses `Touchwiz` from *Samsung*, whereas the product on the right-hand side uses the `HTC Sense` as a graphical user interface.

This example illustrates the main concept of a product line: the identification of commonality and variability between different products of a software product line. These common and variable characteristics of an SPL are represented as *features*.

> "[. . . a *feature* is] a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems." [KCH+90].

Thus, features provide the means to customize, i.e. *configure*, a specific software product at design time to the needs and requirements of a stakeholder.

Example 2.11 (*Software Features of a Mobile Device*).

Considering mobile devices, *software features* are the driver for the integrated WLAN chip, a protocol, such as the border gateway protocol, or an application, such as the navigation system *Google Maps*.

To derive a final software product, the SPL has to be configured. The process of configuration results in a product configuration that is tailored to the custom needs of a stakeholder.

> "A product configuration is a collection of parameterized feature assignments specifying a customized product of an SPL." [Loc12]

Thus, a configured product is a customized instance of an SPL. The configuration of a resulting software product consists of features that are selected according to certain *constraints* imposed by the SPL [CE00]. Therefore, every feature is either bound to be included into the resulting product, i.e., *selected*, or it is bound to be excluded from the resulting product, i.e., *deselected*. Once the configuration process of a software product is finished and the product is deployed, the resulting product is static and fixed regarding the selection of features.

Figure 2.7 shows an abstract illustration of a configuration of a smartphone SPL. The model on the left side represents the variability specification. During the configuration process, features of the SPL are explicitly selected (features with a check mark) and deselected (feature with a cross mark) to derive the final smartphone product configuration.

A key artifact to configure a product of an SPL is the domain variability model. It defines the *variability* of the SPL and introduces dependency relations between

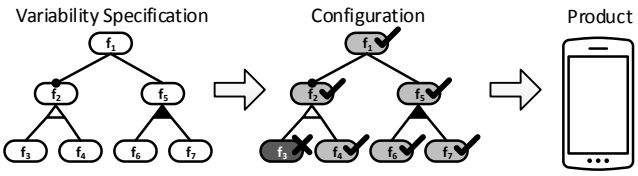


Figure 2.7: Configuration of a Product Line

different features. Thus, variability denotes the ability of a software system to be adapted, customized, or configured for a specific application scenario or to the custom needs of a stakeholder.

Example 2.12 (*Product Configuration of a Mobile Device SPL*).

The Android operating system from Google supports various heterogeneous devices. The mobile devices Google Nexus 4[1] and Google Nexus 7[2] require two different product configurations because of their individual hardware characteristics. The Nexus 4 is a smartphone with

- a 4.7″ display,
- a front-facing camera,
- a rear camera,
- GPS,
- a light-sensor,
- and wireless loading.

In contrast to that, the Nexus 7 tablet has the following characteristics

- a 7″ display,
- a front-facing camera,
- and GPS.

Although both devices use Android 4.2 as operating system, they use different product variants of the operating system. Both devices require the features for a front-facing camera and GPS, i.e., the necessary drivers have to be integrated into the system kernel. Therefore, every constraint that is related to these features, e.g., navigation system requires GPS, is satisfiable and applications such as Google Maps are supported by both devices.

Both devices differ in their capabilities. The Nexus 7 is missing a rear camera, light-sensors, and wireless loading. Therefore, the according software features are not included into the product variation that is specific for the Nexus 7. This restricts the Nexus 7 in its support for applications. Light-sensitive applications,

[1] http://www.google.de/intl/ALL/nexus/4/specs/
[2] http://www.google.de/intl/ALL/nexus/7/specs/

e.g., a night-stand clock, or applications that are executed when the device in a charging tray, e.g., overview of the loading process, are not compatible with the Nexus 7 but it are compatible with the Nexus 4.

The variability constraints imposed by an SPL are specified by using features models. In a feature model, features and their relations are modeled during the domain engineering phase. The next section introduces feature models.

2.2.2 *Feature Models*

A set of features describes an increment in product functionality constituting a distinctive characteristic of a software product and, therefore, provide a notion for specifying and distinguishing the members of a product line [AK09]. In general, not all combinations of features lead to a meaningful or usable product. Therefore, the commonalities and variability associations between features are specified via explicit constraints. Every product configuration of an SPL consists of a distinct combination of features, which has to be *valid* w.r.t. the constraint specification.

Feature models are used to describe the *problem space*, i.e. the variability within an SPL. The *configuration space*, i.e., every valid product configuration, is directly defined by the problem space. The size of the problem space depends on the amount of features and constraints defined in a feature model. For instance, if the feature model does not contain any constraints to restrict the configuration space, the size of the configuration space corresponds to $2^{|\text{number of features}|}$ configurations. In a configuration, a feature is bound to one of the two different states

- **selected** – the feature is bound to be included into the resulting product, or
- **deselected** – the feature is bound to be excluded from the resulting product.

Example 2.13 (*Software Features of a Mobile Device*).

For instance, considering a smartphone, a *software* feature may be the driver for the integrated WLAN chip, a protocol, e.g., the border gateway protocol, or an application, e.g., Skype for Voice over IP (VoIP) calls. To make a VoIP call, an Internet connection has to be active via the border gateway protocol via a WLAN access point. These feature constraints are specified in a feature model. Thus, if a VoIP call is taken, the features `VoIP`, `border gateway protocol` and `WLAN driver` have to be selected to be part of the product configuration.

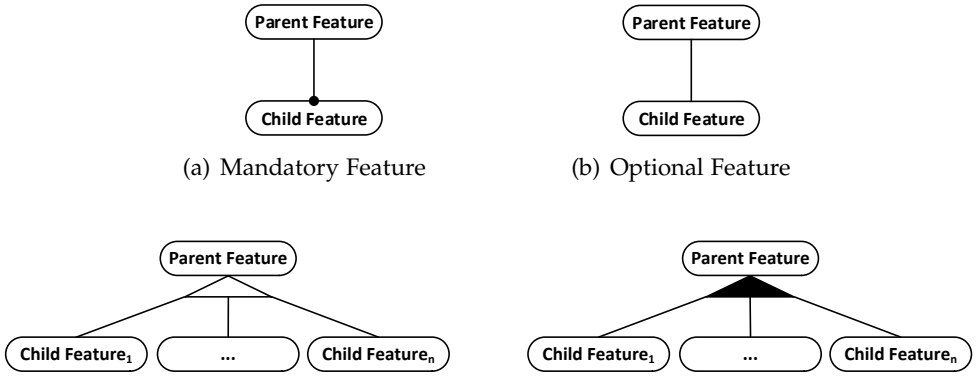
Feature model diagrams are a common graphical notation to specify such characteristics of a feature and variability constraints amongst different features.

Feature Model Diagrams. Feature model diagrams are used to describe the variable and common parts within an SPL. This graphical representation of variability depicts a structured, hierarchical overview of dependencies, restriction, and constraints between features [HST+08]. For that purpose graphical notations have been defined such that a broader audience of stakeholders, e.g., developers,

managers, and customers, are able to interpret and understand the variability within an SPL.

Feature model diagrams are introduced by Kang et al. in [KCH+90] as a part of the feature-oriented domain analysis. They are usually represented as a hierarchical tree to specify dependencies and constraints between features. In feature model diagrams features not only represent user-visible distinctive functional aspects of an SPL, but may furthermore be annotated with *non-functional properties*, e.g., cost or reliability. The features are organized as tree-nodes and tree-edges to specify the dependencies and constraints between the features. The hierarchical ordering of a tree structure allows for a grouping of sibling features as a parent-child-group to constitute the different kinds of variability constraints.

A multitude of competitive graphical representations of feature model diagrams exists within the SPL community. In the remainder of this thesis, a graphical notation according to [CE00] is adopted. Figure 2.8 provides an overview of the graphical representation for the different kinds of tree edges and node groups used in this thesis. Edges connect parent nodes and child nodes, thus repre-



(a) Mandatory Feature (b) Optional Feature

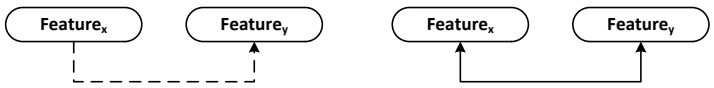
(c) Alternative Group (d) Or-Group

Figure 2.8: Syntactical Constructs of Feature Model Diagrams

senting hierarchical decomposition relations between the corresponding parent feature and its child feature(s). This hierarchical decomposition of features into sub features is used to impose constraints between the involved features on the validity of product configurations. Hence, the selection of a child feature into a product configuration implies the presence of the parent feature within the same configuration [Bat05].

A *structurally well-formed* feature model has to follow several modalities for decomposing features into groups of sub features to impose further constraints on a valid configuration. The four kinds of feature decompositions as depicted in Figure 2.8 are to be interpreted as follows.

- A *mandatory* feature is always part of the product configuration if its parent feature is selected. The presence of the parent feature implies the selection of all of its mandatory child features. Figure 2.8(a) depicts the graphical notation of a mandatory feature.



(a) Binary Require Constraint (b) Binary Exclude Constraint

Figure 2.9: Binary Cross-Tree Constraints between Features

- An *optional* feature may or may not be selected into product configurations if its parent feature is selected. The presence of the parent feature offers the possibility to select or not select any of its optional child features. Figure 2.8(b) depicts the graphical notation of an optional feature.
- From the features contained in an *alternative*-group exactly one feature must be selected into a product configuration if the parent of the group is selected. The presence of the parent feature, therefore, implies the selection of *exactly one* of its child features. Figure 2.8(c) depicts the graphical notation of an alternative-group.
- From the features contained in an *or*-group at least one feature must be selected into a product configuration if the parent of the group is selected. The presence of the parent feature implies the selection of *at least one* of its or-related child features. Figure 2.8(d) depicts the graphical notation of an or-group.

The feature model diagram constructs introduced above are not yet expressive enough to specify the inherent variability of SPLs. Thus, further constructs to express additional constraints among features have to be provided [SHT06]. Those constraints are often represented as additional propositional formulas over features arbitrarily crosscutting the feature tree [BSRC10]. Schobbens et al. [SHT06] discussed that a directed acyclic graph structure for implication dependencies among feature nodes is at least to be supported by a feature model language in order to be conceptually complete for an SPL. Therefore, the introduced tree-structure has to be extended with binary require and exclude cross-tree constraints as shown in Figure 2.9. The meaning of those edges is given as follows.

- If a feature *requires* another feature then the required feature must also be selected into the configuration if the requiring feature is selected.
- If a feature *excludes* another feature then both features may never be part of the same configuration.

The constraint relations introduced above are sufficient to model the variability usually apparent in mobile devices. A feature model is *structurally well-formed* if it is a composition of the introduced group-constraints *mandatory*, *optional*, *or*, and *alternative* as well as the cross-tree constraints *require* and *exclude*.

Example 2.14 (*Mobile device SPL - Specification and Configuration*).

Figure 2.10 depicts an excerpt of the Google Nexus SPL[3] for mobile devices. The feature model organizes the supported features in a tree-like hierarchy. For

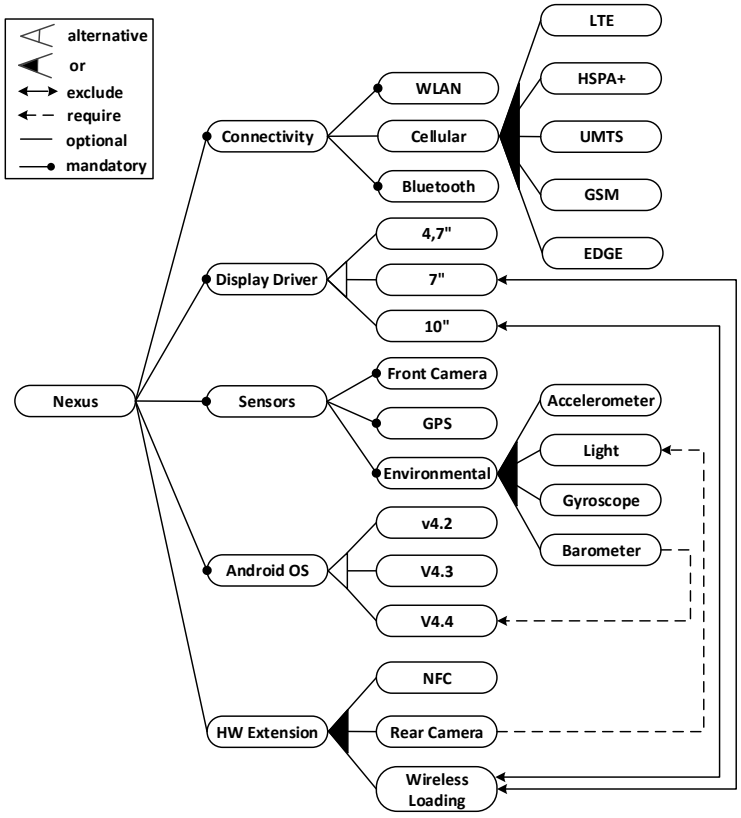


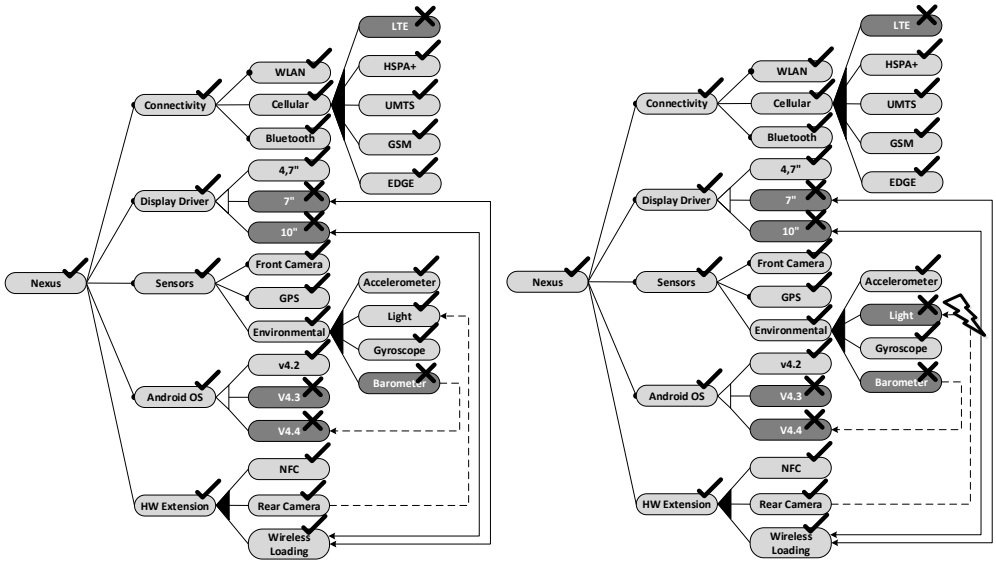
Figure 2.10: Google Nexus SPL

instance, the `Connectivity` feature of the SPL comprises `WLAN`, `Cellular` and `Bluetooth` as direct child features. Depending on its modality, a single child feature is either *mandatory* for its parent feature, or it is *optional*. For example, the `Connectivity` feature constitute mandatory core functionality to be part of every variant of a mobile device, whereas the `HW Extension` feature with its child features is optional, e.g., not all products of the Nexus SPL necessarily include the features `NFC`, `Rear Camera`, and `Wireless Loading`. However, if a child-feature is selected for a product then the parent feature must also be a part of the feature selection, e.g., if `NFC` is selected then `HW Extension` has to be selected additionally.

A Nexus product is able to support a multitude of different connection types. Every product supports the *mandatory* features `WLAN` and `Bluetooth`, whereas the child features of the *optional* feature `Cellular` constitute an *or-group*. Therefore, whenever a `Cellular` connection is a required feature for the product configuration *at least one* and up to *every* of the features within an *or-group* have to be selected.

The software of every mobile device has to support a display, thus `Display` is a *mandatory* feature. In addition to that, features are collectable in an *alternative-group*. For a display, it is sufficient to support one category of displays

[3] http://www.google.de/intl/ALL/nexus



(a) Valid Configuration (b) Invalid Configuration

Figure 2.11: Product Configuration Google Nexus 4

exclusively, e.g., either a `4.7"`, a `7"`, or a `10"` display. Therefore, the display features are collected within an *alternative-group*, which implies the selection of *exactly one* of those group features.

Finally, *cross-tree edges* denote feature dependencies that crosscut hierarchies, e.g., `Rear Camera` *requires* a `Light` sensor for an automatic filter correction and `Barometer` *requires* the most recent Android operating system `v4.4`. In addition to *require* edges, features may be incompatible, e.g., due to space restrictions. For example, `Wireless Loading` is never used in a device with large displays such as `7"` and `10"`.

Figure 2.11 depicts two variants of a product configuration for the stakeholder selection of the features `4.7"`, `EDGE`, `GSM`, `UMTS`, `HSPA+`, `v4.2`, `NFC`, and `Rear Camera`. Features that are selected for the final product are highlighted in light-grey and have a check-mark, whereas deselected features are highlighted in a dark-grey and have a cross-mark. A valid product configuration is depicted in Figure 2.11(a), in which the selection of the features is completed w.r.t. the constraints imposed by the feature model. For example, since `Rear Camera` is required by the stakeholder selection, the `Light` sensor has to be selected to satisfy the variability constraints of the Nexus SPL. In contrast to that depicts Figure 2.11(b) an invalid configuration of the Nexus SPL. In this example the stakeholder additionally requires the `Light` sensor to be excluded from the final product configuration, e.g., to reduce production costs. However, this requirement contradicts the selection `Rear Camera` of the stakeholder, which requires the `Light` sensor. Therefore, it is not possible to derive a valid product configuration according to the requirements of the stakeholder.

Feature models may not only be used to specify the static variability of an SPL but are also capable to specify variability of an adaptive system. The concept of

dynamic software product lines is introduced in the next section as an approach to manage variability dynamically at runtime.

2.2.3 *Dynamic Software Product Lines*

Self-Adaptive Software Systems (SAS) are required to operate under the influence of a continuous changing context without interruption. This is especially true for highly mobile resource constraint systems, e.g., smartphones, tablets or notebooks. Contextual changes usually imply changes on the requirements imposed on the running system. Such scenarios motivate the adaptation of systems that react to external variations by adapting themselves dynamically at runtime in order to uphold certain functionality and service qualities.

Dynamic Software Product Lines (DSPLs) represent one possibility to handle the inherent changes emerging at runtime of a system [HSSF06]. DSPLs exploit the concepts of static SPL engineering. A traditional SPL product implements a static behavior and is not reconfigurable at runtime. However, DSPLs rely on the same concepts as SPLs but are intended to handle variability at runtime dynamically. The DSPL development mainly intends to produce configurable products [LK06] whose autonomy allows to reconfigure themselves and benefit from constant updating.

> "Dynamic software product lines extend existing product line engineering approaches by moving their capabilities to runtime, helping to ensure that system adaptations lead to desirable properties." [HPS12]

Thus, a DSPL supports feature (de-)selection dynamically at runtime, depending on the requirements imposed by a contextual situation.

STATIC AND DYNAMIC ASPECTS. The dynamic character of DSPLs as well as the alignment to classic SPLs make a DSPL based adaptation a suitable candidate to manage dynamic systems such as mobile devices. On the one hand, mobile devices are representative examples for SPLs as shown in the case of the previously introduced *Nexus SPL* for mobile devices. On the other hand, a user-visible aspect, quality or characteristic of a software system, i.e., the definition of a *feature* according to [KCH+90], represents exactly those properties of a mobile device, which are important to a user. However, the scope of a feature differs slightly in an SPL and a DSPL.

Features in an SPL refer to *static* characteristics of the software or system, i.e., the feature selection for a software product does not change once the configuration process is finished. Static features are selected during a configuration process at design time. The selection of static features depends on the requirements stated by a stakeholder, e.g., the user requires a navigation system, or on the restrictions imposed by the target domain, e.g., the hardware capabilities of a mobile device. Therefore, the general definition of a feature has to be refined to refer to the *static features* of an SPL.

Definition 2.1 (*Static Feature*)**.** A *feature* is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system [KCH+90]. A *static* feature is selected or deselected from a software product at design time.

In contrast to the static features of an SPL, at least features have to be *dynamic* in a DSPL, due to the continuous reconfigurations at runtime. Dynamic features have to be selected and deselected from a runtime configuration of the DSPL continuously, depending on the requirements stated by the context of a mobile device. Thus, dynamic features are selected during a reconfiguration process at runtime and not at design time. Therefore, the general definition of a feature has to be refined to refer to the *dynamic features* of a DSPL.

Definition 2.2 (*Dynamic Feature*)**.** *Dynamic features* satisfy requirements stated by a contextual situation and are continuously reconfigurable at runtime.

Mobile devices are composed of static features as well as dynamic features. In a first step, the mobile device has to be configured statically at design time like a traditional SPL, due to the stakeholder requirements and capabilities of the target device. After the software product configuration is deployed on a device it becomes a DSPL, due to the requirements of a constantly changing context. This implies two aspects

- the configured SPL product still has to contain open variability, and
- the variability specification of SPLs and DSPLs are different.

If the configured software product does not contain any variability, e.g., features that may *not* be selected or deselected at runtime, the product is static. However, this contradicts the key aspect of DSPLs of a dynamically reconfigurable system at runtime.

The different scope of an SPL and DSPL results in different views on the variability aspects [SvGB05]. An SPL is specified according to the static constraints imposed by software or hardware aspects, whereas a DSPL is specified according to dynamic aspects of the software. An SPL has to deal with issues such as the compatibility of features with the hardware capabilities of the target platform, e.g., a WLAN driver must not be deployed on a device without a WLAN hardware chip. In contrast to that, a DSPL has to deal with the dynamic changes in a software configuration, i.e., selection and deselection of features at runtime of the software. Thus, every feature in a DSPL has to be supported by the device, on which the DSPL is deployed. Additionally, the variability constraints are different for a static and a dynamic specification. For example, WLAN is a mandatory feature in the Nexus SPL, whereas WLAN may be activated or deactivated at runtime. Thus, WLAN is an optional feature in the respective DSPL.

Based on the previously introduced Nexus SPL the following example introduces an excerpt of a respective DSPL.

Example 2.15 (*Nexus DSPL - Runtime Variability Specification*).

Figure 2.12 depicts a case study of a DSPL for a Google Nexus SPL [SOS+12, SLR13]. The DSPL is used as a running example throughout the remainder of this thesis. In contrast to the Nexus *SPL* example depicted in Figure 2.10 the

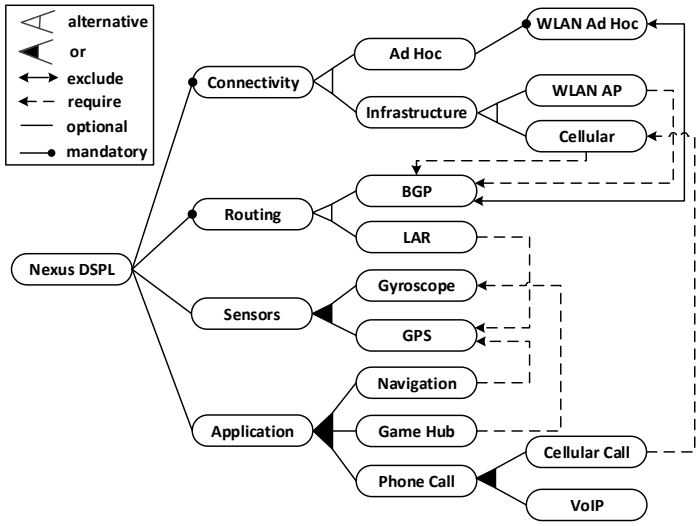


Figure 2.12: Feature Model Nexus DSPL

DSPL focuses on the variability at runtime. Although there are some features in both specifications, these features have different constraint relations due to their *static* and *dynamic* scope. For example, `GPS` as a static feature in Figure 2.10 is a mandatory feature and therefore deployed on every device of the Nexus SPL. In contrast to that the dynamic `GPS` feature in Figure 2.12 is reconfigurable since it is in an *or-group* with the sibling feature `Gyroscope`. Another example for a static and dynamic feature is `WLAN`. Within the Nexus SPL, `WLAN` is a mandatory feature. However, there are two dynamic features in the DSPL specification, which depend on the `WLAN` driver, e.g., `WLAN Ad Hoc` uses the `WLAN` driver in an ad hoc mode and `WLAN AP` uses the `WLAN` driver to connect to an Access Point (AP).

The remaining variability at runtime is specified as follows. The `Application` feature of the Nexus DSPL comprises `Navigation`, `Game Hub` and `Phone Call` as direct sub features. A phone call is executable in two possible ways

- via `VoIP`. A Voice over IP (VoIP) call is always executable. Since `Connectivity` is a mandatory feature, the device provides a connection that is usable by `VoIP`.
- via `Cellular` network. A standard phone call is taken via a cellular network provider. Therefore, this feature depends on an `Infrastructure` based connection to a `Cellular` tower.

Thus, technically it is possible that several phone calls may be executed via both features in parallel, e.g., putting a call on hold.

The basic `Connectivity` feature constitutes a mandatory core functionality and has to be part of every smartphone variant, whereas the `Sensors` feature is optional and may be deactivated at runtime. Depending on the environment, different `Routing` protocols have to be used to provide an efficient connection. In the mobile device case study, there is an alternative-group of two different routing protocols

- `BGP` (Border Gateway Protocol) [RLH06] is an Internet Protocol (IP) based routing protocol, which is commonly used in infrastructure based networks like the Internet. Hence, it is not compatible with ad hoc connections.
- `LAR` (Location Aided Routing) [KV00] is a routing protocol for wireless ad hoc networks that uses the geographic location of the devices, provided by a `GPS` sensor. Instead of flooding the message to all devices in the neighborhood, the message is directed in a geographic direction. `LAR` cannot operate without an active `GPS` sensor.

Thus, both routing protocols depend on a different connectivity configuration and may never be active at the same time.

Reconfiguration at Runtime. A reconfiguration adapts a running software system dynamically from one product configuration to another product configuration. As a result, a different product configuration becomes active and the behavior of the software system changes according to the new specifications. The new product configuration may be adapted into yet another product configuration and so forth [MCP13]. In the remainder of this thesis, I refer to the active product configuration that is about to be adapted as the *current configuration* and the product obtained after the reconfiguration as the *target configuration*.

Example 2.16 (*Continuous Reconfiguration*).

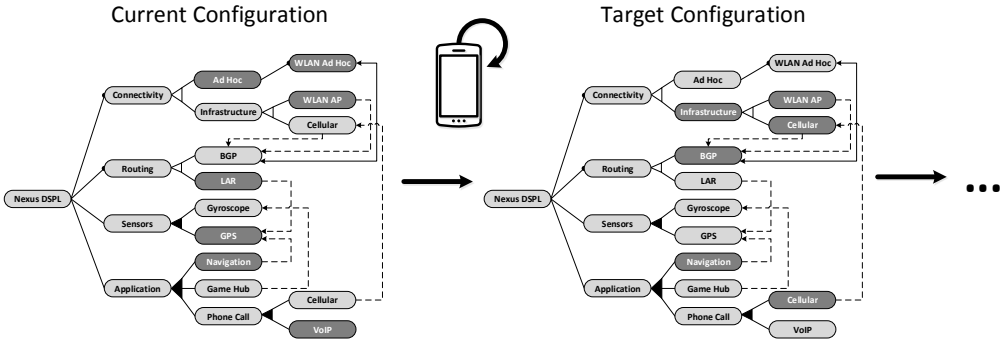


Figure 2.13: DSPL Reconfiguration Sequence

Figure 2.13 depicts an excerpt of a reconfiguration sequence for a mobile device. The two depicted configurations differ slightly in their set of selected and deselected features. The left-hand side configuration provides the functionality to execute a phone call via a `Cellular` network connection. After the reconfiguration, the device executes phone calls via `VoIP` by using an `Ad Hoc` connection.

The left-hand side configuration in Figure 2.13 depicts the current configuration of the device. Due to a change in the context of the device, e.g., the cellular network broke down, the device has to be reconfigured to the target configuration on the right-hand side configuration in Figure 2.13.

For every performed reconfiguration the variability specification, i.e., the feature model, has to be analyzed. The contextual requirements and the specified feature constraints have to be *solved* to derive a new target configuration [BHS12, MBJ09]. Although this process is automatable by so called *solvers*, the process of deriving a target configuration is a difficult computational problem [Coo71].

The degree of difficulty for deriving a target configuration is directly related to the number of steps required by a solver to perform a satisfiability check of all imposed constraints and requirements. Therefore, the complexity of a variability specification grows with the number of features and constraints [MWC09]. The more complex the specification, the greater the number of required steps. For all known solving algorithms, the amount of executed steps grows exponential in the worst case. Especially on a mobile device, such a worst case reconfiguration behavior has to be reduced to a minimum w.r.t. its processing time and its resource consumption [MWC09].

2.2.4 *Research Challenges in DSPLs*

As an evolving field, the DSPL community still has to face numerous research challenges. In the following, state-of-the-art research challenges are introduced that are addressed in the remainder of this thesis. Similar to the previously introduced research challenges in SAS, DSPLs have to deal with the problem of (i) an *autonomous reconfiguration* and (ii) non-intrusive *resource consumption*.

RC5: AUTONOMOUS RECONFIGURATION.

> "While the core variability model focuses on modeling the reconfiguration options, ongoing DSPL research attempts to enlarge these approaches to capture context description and decision making. [...] an additional concern is how to best extend variability modeling so that developers can use it as a basis for context interpretation and thus support the fully autonomous case." [HPS12]

This quote summarized the main issue for an autonomic reconfiguration at runtime. The variability specification of a DSPL has to be aware of the contextual changes and resulting changes in the requirements imposed on the system. To derive a reconfiguration from a current configuration to a target configuration, a mapping is necessary, which links all contextual requirements to the variability specification of a DSPL.

RC6: RESOURCE CONSUMPTION.

> "The quest to solve such [constraint satisfiability] problems pushes computers to their limits and requires advanced methods..." [HMS10]

Although Hölldobler et al. do not address constraint solving on mobile devices specifically, this quote is generally applicable for all computing platforms. The resource utilization to compute a suitable target configuration grows with the complexity of the variability specification. Thus, the energy consumption and processing time of a reconfiguration grow with the constraint density of a DSPL specification. However, especially for mobile devices, this effect has to be mitigated due to the responsiveness of a device and the limited resources.

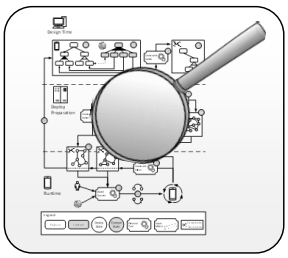
After the introduction of SAS and DSPLs as fundamental concepts of this thesis and discussion of research challenges in the respective domains, an overview of related concepts and frameworks are provided in the next section.

Part II

SYSTEMATIC CONTEXT-AWARENESS

3

CONCEPT AND CONTRIBUTIONS



A common approach for a Self-Adaptive Software System (SAS) to plan and execute an adaptation in a regulated and controlled manner is the Monitor-Analyze-Plan-Execute feedback loop. This feedback loop, in short a MAPE-K feedback loop, is based on adaptation Knowledge, which is stored on the device as previously introduced. Thereby, an adaptation is divided into the four tasks of Monitoring, Analyze, Plan, and Execute based on a Knowledge Model. The MAPE-K is a paradigm to control an adaptation. A concrete realization of each step remains to the developers, i.e., which techniques or methodologies are used to implement a MAPE-K loop in an SAS remains open. In this thesis, I propose to apply SAS to specify the knowledge basis of an SAS and to plan an adaptation.

The concept of a Dynamic Software Product Line (DSPL) based adaptation has two fundamental advantages. Firstly, the SPL provides the means to support a product line of heterogeneous devices. The running example of a Nexus (D)SPL illustrates the need for such a support of heterogeneity, e.g., Nexus tablets as well as smartphones rely on the same fundamental specification. Most vendors provide a product line of mobile devices, such as the Galaxy product line by Samsung, the Desire product line by HTC , or the Lumia product line by Microsoft.

Secondly, the systematic characteristics of a DSPL lead to a higher reliability and predictability of an adaptation. With a DSPL, every adaptation is executed w.r.t. the constraints that are imposed on the system, thereby avoiding inconsistent or erroneous adaptations. For example, if a Nexus 4 is to be used for navigation purposes in a car, certain constraints have to be considered, such as the activation of GPS. Those constraints are specifiable in a feature model, which is a fundamental part of the DSPL adaptation knowledge.

The two domains SAS and DSPL have the same purpose, i.e., to change the behavior of a software system at runtime. However, they use a different terminology to describe such a change in the system, i.e., *adaptation* and *reconfiguration*, respectively. Note that the terms adaptation and reconfiguration have a different meaning in their respective domain. An adaptation describes a change in the system w.r.t. the MAPE-K loop paradigm. Thus, for an adaptation, a change in the context of the device has to be detected, a corresponding adaptation plan has to be derived, i.e, what features have to be how *reconfigured*, before this adaptation plan is executable.

A DSPL extends the concept of a static configuration from the SPL domain to describe dynamic reconfigurations at runtime. A *reconfiguration*, as a fundamental concept in a DSPL, refers to the change in the configuration of a single or multiple features [Lee06, HSSF06]. In this case, the system switches from a source product configuration to a target product configuration. Thus, a reconfiguration is a part of the adaptation process of an SAS and is triggered by a change in the contextual situation of the device.

However, due to the limited resources of mobile devices the development of a DSPL based adaption is challenging. Research challenges (RC) that still remain unsolved for SAS have been identified in the fundamentals of this thesis as follows

RC1. Systematic Engineering Approach,

RC2. Flexibility at Runtime,

RC3. Autonomous Adaptation, and

RC4. Non-Intrusive Adaptation,

whereas the following two RCs have been identified for DSPLs

RC5. Autonomous Reconfiguration and

RC6. Improvement of Resource Consumption.

Note that the RC3 and RC5 as well as RC4 and RC6 deal with similar problems in different domains (SAS and DSPL, respectively). Although, I address all of the listed research challenges above, I focus on these similar research challenges and aim to satisfy the following goals

- to provide an autonomous, model-based adaptation process (G1) and
- to minimize the resource consumption of an adaptation at runtime (G2).

This chapter introduces how a model based DSPL engineering process is applied to the MAPE-K loop of an SAS. Further, an overview of the techniques and approaches that are discussed in this thesis is given. These techniques are categorized w.r.t. the life-cycle stages of a mobile software system, i.e., *design time*, *deployment*, and *runtime*.

3.1 DSPL BASED MAPE-K FEEDBACK LOOP

Figure 3.1 depicts a DSPL engineering approach integrated into the MAPE-K paradigm based on a concept introduced by Bencomo et al. [BHS12]. Starting with an initial feature model specification at design time on the left-hand side of Figure 3.1, the SAS is derived and deployed on the device. For example, at design time system specific constraints are specified in a feature model, such as dependency relations or conflict relations between features. For example, a navigation system requires GPS or an ad hoc connection contradicts an infrastructure connection. Further, contextual constraints are specified in a context model, e.g.,

the context *car* imposes a requirement such as *"navigation system has to be active for routing purpose"*.

At runtime, the MAPE-K adaptation loop is continuously executed, as depicted on the right-hand side of Figure 3.1. The four MAPE-K tasks monitoring, analyze, plan, and execute are processed in order to adapt the device. These tasks rely on the knowledge that is residing on the device, such as the feature model and the context model. Every adaptation is profiled, e.g., what and why features have been reconfigured. This information is collected and passed as feedback back to the developers at design time.

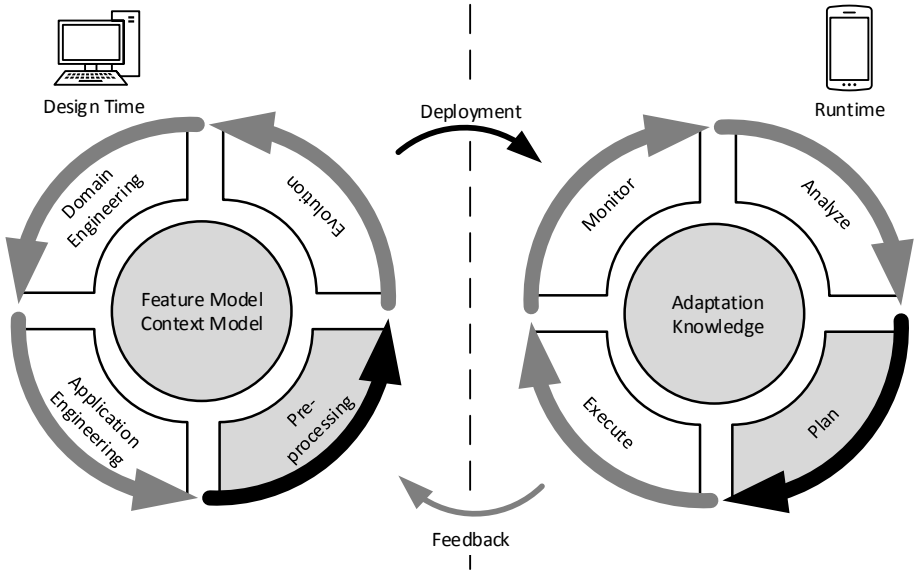


Figure 3.1: MAPE-K Feedback Loop for DSPLs (adopted acc. [BHS12])

3.1.1 *Evolution at Design Time*

The left-hand side depicts the engineering process at design time. To provide the means for an offline-evolution of the DSPL, I added the task of *pre-processing* and *evolution* to the traditional SPL tasks *domain engineering* and *application engineering*.

Contributions discussed in this thesis are highlighted with a grey background or with a black solid arrow. In this regard, I focus on the pre-processing task which is discussed in the Chapter 5.

The tasks executed at design time are summarizeable as follows.

DOMAIN ENGINEERING. Key artifact of the domain engineering is the variability specification of the system based on a feature model [PBv05]. In addition to that, the contextual situations are specified, i.e., the requirements of a context and the mapping to the feature model.

APPLICATION ENGINEERING. During the application engineering, a software product is derived, based on the feature model specification [PBv05]. In contrast to a traditional SPL product where every feature is either selected or deselected, the DSPL software product contains yet unconfigured features prior to runtime that are reconfigurable at runtime.

PRE-PROCESSING. In a pre-processing task, adaptation knowledge is derived based on the properties of the target device. In contrast to traditional SPL engineering (c.f. Figure 2.5) the introduced DSPL engineering process adds a context model to the traditional feature model. Based on these models, an extensive reasoning is applicable to automatically pre-plan adaptation knowledge for each individual device. For example, the adaptation behavior at runtime may differ between a Nexus 4 smartphone and a Nexus 7 tablet.

OFFLINE EVOLUTION. The data that is collected at runtime of the device is exploitable to optimize the specification of the DSPL. The monitored contextual changes, why the device had to be adapted, if an adaptation was executed erroneous, and what features had to be reconfigured, are aggregated as feedback and passed to the developers. Based on this feedback the developers are able to optimize the specifications constantly. I refer to such an update of the DSPL specification at design time in the following as *offline evolution*.

Example 3.1 (*Offline Evolution of a DSPL*).

Based on the feedback that is gathered at runtime of a DSPL, constraints in the feature model and contextual situations in the context model may be added, removed, and changed. Consider the Nexus DSPL depicted in Figure 2.12. If at runtime requirements emerge that demand that the features `Cellular` and `WLAN AP` are active at the same time, a corresponding adaptation is not executable. According to the feature model specification, `Cellular` and `WLAN AP` exclude one another. When such unsatisfiable requirements emerge, they are profiled and sent to the developers as feedback. At design time the developers have to evaluate if the demand for such an adaptation outweighs the cost for changing the specification.

3.1.2 *Adaptivity at Runtime*

The right-hand side depicts the MAKE-K feedback loop as introduced in the previous chapter. The combination of a DSPL with the MAPE-K paradigm is mainly tied to the *knowledge* and the *plan* part of MAPE-K. The adaptation knowledge consists of the feature model and context model specified at design time. Further, all valid configurations of a DSPL are collected in a configuration state space in which every state corresponds to a configuration of the DSPL. This configuration state space is extended with transitions to denote the reconfiguration possibilities of a DSPL. The state space is used to coordinate and profile the adaptation behavior at runtime. For example, the feature model specifies system specific constraints, e.g., `Navigation` requires `GPS`, the context model specifies requirements

that are imposed by contextual situations, e.g., at *Home* it is required that *"WLAN is active"*, and the state space describes possible adaptations of the system, e.g., the switch from a system configuration that uses an `Infrastructure` based communication to a system configuration that uses an `Ad Hoc`-based communication.

ANALYZE AND PLAN TASKS. The analyze task identifies a change within the active contextual situation, e.g., the context *Home* is left or the context *Office* is being entered. The associated requirements for the currently active contextual situation are passed to the planning task. During the planning task, the feature model and the context model are used to derive a configuration of the device that satisfies the contextual requirements. The feature model specification ensures that conflicting feature configurations, e.g., different connectivity features are active at the same time, are avoided and dependency relations among features, e.g., the feature `Navigation` requires the feature `GPS` to be active, are satisfied in every configuration. The context model ensures that the requirements of a contextual situation are mapped accordingly to the features of the system and that those requirements are satisfied if the respective contextual situation becomes active.

ONLINE EVOLUTION OF A DSPL. In contrast to an offline evolution of a DSPL, the information that is collected at runtime is also exploitable to extend and update the knowledge stored on the device. At runtime new knowledge may be gathered, such as new contextual situations and the requirements they impose or new configurations of the device that are derived on-demand. Such knowledge is added to the available adaptation knowledge, i.e., to the feature model specification, to the context model, or to the configuration state space residing on the device. I refer to such an update of the DSPL knowledge at runtime in the following as *online evolution*.

Example 3.2 (*Online Evolution of a DSPL*).

Consider the following example. The developers of the Nexus DSPL specified the contextual situation of *Home* and *Office*. However, they did not explicitly consider the possibility of a combination of both contexts, e.g., the user is working at home. Thus, both contexts have to be combined to satisfy the requirements of *Home* as well as *Office*. If such a contextual situation is satisfiable and is yet unknown to the device, the adaptation knowledge residing on the device is extended with the information discovered at runtime. The new contextual situation *{Home, Office}* is added to the context model and the corresponding device configuration which satisfies this context is added to the configuration state space.

SPL engineering approaches are applied at design time, prior to runtime. Although DSPLs are an extension of SPL concepts and executed at runtime, they are not intended to operate in a resource efficient manner. Thus, the concept of a DSPL, i.e., derivation of a valid configuration and the reconfiguration of features at runtime, is not intended for resource constraint systems such as mobile devices.

According to *RC4*, the adaptation process has to be non-intrusive for an SAS. It has to be shown that DSPLs are an appropriate solution to execute an adaptation

of resource constrained devices. The next section introduces new techniques and extensions of existing concepts to establish a DSPL engineering process, specifically designed to enhance the resource utilization of adaptive mobile devices.

3.2 Optimization of a DSPL Engineering Process

At *design time* the developers specify and update the feature model. In addition to that, a reconfigurable product configuration is derived w.r.t. the characteristics of the target device. At this point of time the configured software product is ready for *deployment* on the intended mobile device. In this thesis, I extended the deployment process with a pre-processing step to reduce the impact of a DSPL-based adaptation at runtime. Since this step is computationally expensive, I assume that the pre-processing is executed on a computer with sufficient computational power.

At *runtime*, the reconfiguration behavior of the device is profiled to improve the reconfiguration process on a long-term basis, e.g., reducing the overall reconfiguration costs. Additionally, the collected information is leveraged to provide the means to improve the SPL specification at design time.

Figure 3.2 provides a conceptual overview of the contributions given in this thesis. The presented DSPL engineering methodology is divided into the *design time*, *deployment preparation*, and *runtime* stages of a software application for mobile device. In the following, an introduction of these stages and the respective techniques which are applied in these stages is given.

3.2.1 *Optimization at Design Time*

At design time the developers specify a *feature model* for a mobile device DSPL, e.g. the Nexus SPL. The specification comprises the traditional variability characteristics of a product line as well as the DSPL runtime variability characteristics. In addition to the feature model, a *context model* is specified containing contextual scenarios, e.g., *Office*, and the requirements they impose on the system, e.g., *"WLAN has to be active"*. To minimize the computational efforts at runtime, the feature model and context model specification of a DSPL is reduced according to the individual characteristics of a device.

In the following the artifacts which are developed during the design time of a DSPL are introduced as depicted in Figure 3.2, (01) the feature model, (02) the context model, and (03) a device specific reduction of the DSPL specification.

(01) **Feature Model.** The feature model specifies the variability of a product line, e.g., the Nexus SPL depicted in Figure 2.10, as well as the variability occurring at runtime, e.g., the Nexus DSPL feature model depicted in Figure 2.12.

(02) **Context Model.** A traditional feature model is used to describe the properties of a software system. However, contexts represent characteristics of the environment of a device, e.g., the device is currently residing in the context *Office*. Further, every context imposes requirements on the system which have to be mapped to the features of a DSPL, e.g., the context *Office*

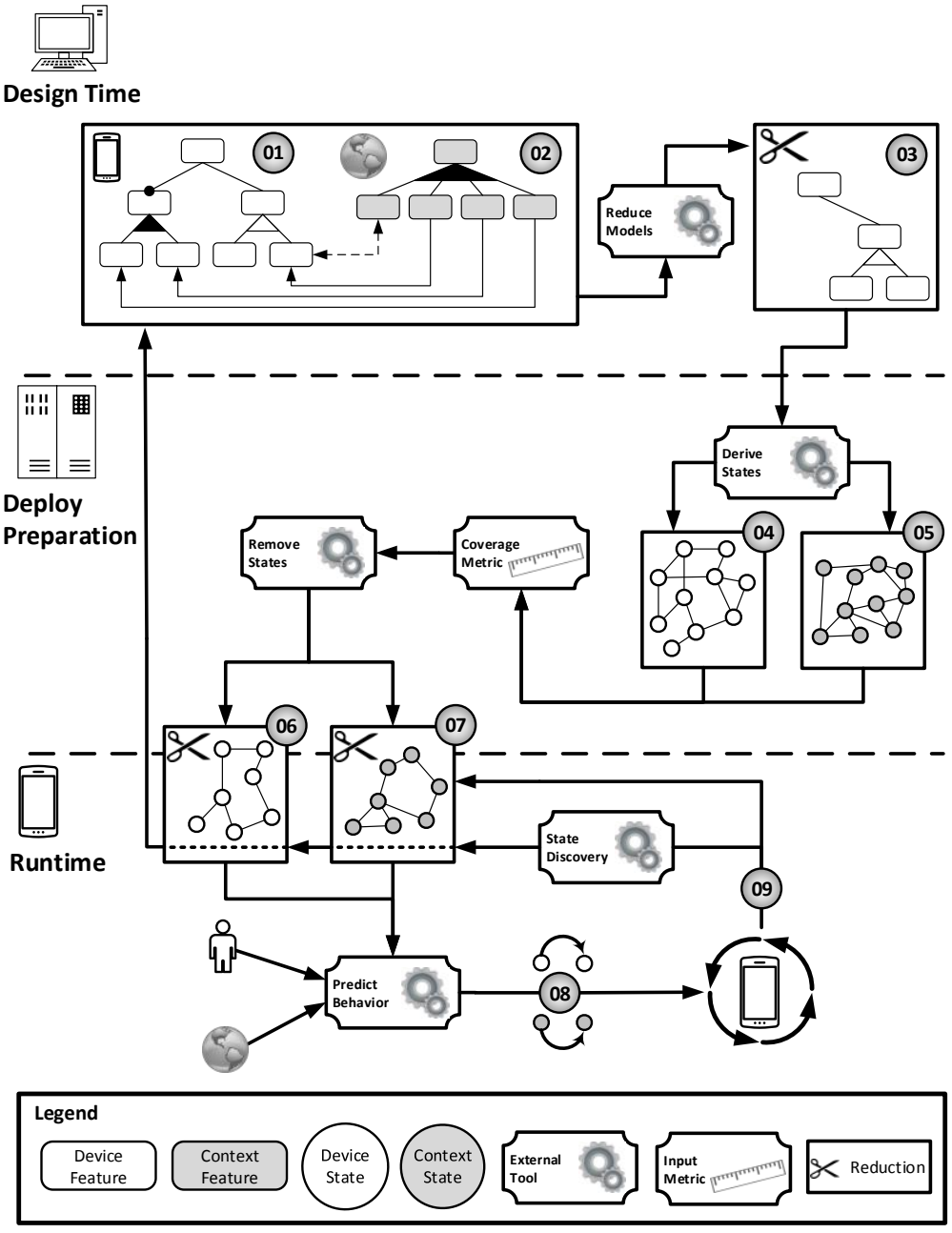


Figure 3.2: Overview of a resource friendly DSPL adaptation methodology. The main intention of this thesis is to establish an autonomous adaptation process based on a DSPL and to improve the resource consumption of a DSPL at runtime. The 9 techniques to realize these propositions are divided into three distinct stages in the life cycle of a DSPL, i.e., (i) Design Time, (ii) Deploy Preparation, and (iii) Runtime.

demands *"WLAN has to be active"* which is mappable to a constraint such as that the feature `WLAN AP` has to be selected when *Office* is active. In this regard, every context specifies requirements on the feature configuration of a DSPL. Thus, a traditional feature model provides the means to specify the variability of a software system. It is not intended for a specification of contexts and how they relate to features. However, the absence of a context specification prevents a DSPL to autonomously reconfigure itself if the context changes. Therefore, the concept of a traditional feature model has to be extended to support the requirements imposed by a context.

The combination of feature model and context model enables a DSPL to be aware of the dynamic changes that occur in the context of the device. Therefore, a context-aware DSPL is capable to autonomously reconfigure a set of features according to the requirements of a context. The DSPL is aware how features have to be reconfigured and why, e.g., the context specifies the feature requirements and the device denotes a (de-)activation of a context. In this regard, an extended context-aware DSPL is capable to process an *adaptation* autonomously.

03 **Product Specific DSPL.** The feature model and the context model of a DSPL are applicable for the whole product line, e.g., the Nexus DSPL depicted in Figure 2.12 is applicable for runtime reconfigurations of the Nexus 4, 7, and 10. However, not every device may support the variability of the whole product line. For example, the Nexus 4 supports connectivity features such as `EDGE`, `GSM`, and `UMTS`, whereas a basic Nexus 10 tablet does not support any of these features. Thus, a Nexus 4 is variable w.r.t. the connectivity features. These features are dynamically reconfigurable at runtime, whereas the Nexus 10 is not able to active them. The device characteristics also affect the contexts that are supported by a device, e.g., every context that depends on connectivity features such as `EDGE`, `GSM`, and `UMTS` is not supported by a Nexus 10 tablet.

To support such heterogeneity in the characteristics of a set of devices, the original feature model and context model specification of a DSPL may be pre-processed according to the characteristics of an individual target device. In order to reduce the computational efforts at runtime, features and contexts that are either incompatible with the target device or have always to be active at runtime, i.e., they are not reconfigurable on a device, are removed from the specification during a *reduction process*. Less features and less constraints implies less computational efforts whenever a configuration is derived at runtime.

The reduction of a feature model and context model is executed at design time and requires the usage of a constraint solver to avoid an erroneous reduction of the specification. The feature model constraints, e.g., or-groups, alternative-groups, require edges, and exclude edges, have to be considered to ensure a correct reduction of the specification.

Traditional SPL pre-configuration approaches [CHE05] are not applicable for DSPLs because they intend to fully configure a product at design time that

is not intended for a reconfiguration at runtime. Current approaches that also reduce the size of a feature model [RSA11, ACLF11] have a similar goal as pursued in this thesis, i.e., to derive a reduced feature model according to the individual characteristics of a device. However, they do not consider the reconfigurability of a DSPL at runtime. Existing reduction approaches do not aim to maximize the device specific reconfiguration possibilities at runtime.

To establish a fundamental basis of this thesis, I provide a formal definition of the configuration semantics of a DSPL in Chapter 4. Additionally, this chapter discusses the extension of a DSPL to a context-aware DSPL based on the combination of a *feature model* and a *context model*. The process of *reducing* a feature model and context model to derive a device specific DSPL is discussed in Chapter 5.

When the configuration process of the DSPL for a certain target device is finished and the specification is reduced w.r.t. the characteristics of the target device, the DSPL is ready for deployment, which is discussed in the next section.

3.2.2 *Optimization and Deployment Preparation*

A feature model restricts the number of valid configuration possibilities of all features to a set of configurations that are valid for the DSPL. For example, the 19 features of the DSPL potentially allow 2^{19} configurations. However, the constraints imposed by the feature model of the Nexus DSPL reduce these configurations to 66 valid product configurations.

The valid configurations of a DSPL form the *state space* of a DSPL in which every state corresponds to a valid configuration of the DSPL. Since a traditional DSPL is extended to a context-aware DSPL, the state space further contains valid contextual configuration states. Thus, the state space of a DSPL consists of two set of states, i.e., the configuration states and the contextual states.

Depending on the initial variability and context specification, it is possible that the state space is over-specified. For example, the requirement *"WLAN has to be active"* is satisfiable by several configuration states of the device, e.g., one configuration where `GPS` is active and another where `GPS` is deactivated. Therefore, a computationally expensive reduction approach is applied during the deploy preparation of the DSPL. The state space is reduced to a representative *incomplete* set of configuration states and contextual states, based on a reduction criterion.

In the following, the artifacts that are derived during the deploy preparation of a DSPL are introduced as depicted in Figure 3.2, ⑭ the *configuration state space*, ⑮ the *contextual state space*, ⑯ the *incomplete configuration state space*, and ⑰ the *incomplete contextual state space*.

⑭ **Configuration State Space.** By using a constraint solver, every valid configuration of a DSPL is computed to form the configuration state space w.r.t. the feature model specification. This state space consists of all valid product configurations. This state space is enriched with transition relations between the states to denote the reconfiguration possibilities of a DSPL.

Existing approaches such as [BSBG08, Hel12, DPS12] similarly propose to use such a transition system to describe reconfigurations of a DSPL. However, they neglect the influence of a context on the system and, therefore, are not capable to execute a transition autonomously if the context changes. Further, they assume that every product configuration is part of the transition system and that a reconfiguration may be executed from one configuration to *every* other configuration of the DSPL. This assumption has the disadvantage that it consumes a lot of resources.

05 **Context State Space.** The contextual state space is a collection of contexts which may emerge at runtime. Every state represents a contextual situation, e.g., *Office*, or *Car*, and the requirements a context imposes. Similar to the configuration state space, the contextual state space is derived from the context model by using a constraint solver. Further, the context state space is also enriched with transition relations between the states to denote changes between contextual situations. Both state spaces are associated, e.g., when the state of the context changes a corresponding reconfiguration in the configuration state space may be triggered.

With the combination of a contextual state space and a configuration state space, it is possible to avoid a third disadvantage of existing DSPL transitions systems, i.e., the *inflexibility* of reconfigurations. A traditional transition system is inflexible w.r.t. minor changes in the feature configuration. For every reconfiguration that is triggered by a contextual change, a transition has to be executed. However, it may be possible that certain features are irrelevant for a context situation. For example, *Office* requires *"WLAN has to be active"*. The feature `Gyroscope` is irrelevant for that requirement and may be arbitrarily reconfigured. This thesis provides an abstraction concept for such irrelevant features to avoid the additional execution of reconfiguration transitions within a contextual situation.

06 & 07 **Incomplete State Space.** Instead of using a state space that contains every valid device configuration and every context that may occur at runtime, the state space may be reduced to avoid an unnecessary utilization of resource at runtime, e.g., for searching suitable configuration states for a contextual situation. Both state spaces may be reduced according to a specific goal. The goals for such a reduction are specifiable via a certain criterion. Based on the criterion, certain contextual states or configuration states are identified that are either removed from the state space or remain within the state space. For example, it may be sufficient to combine compatible contexts in a pair-wise manner, e.g., *Home* and *Office*, to cover all contextual situations emerging at runtime. Therefore, contextual states that reflect contextual combinations consisting of more than two contexts are discarded.

Based on the configuration state space, the contextual state space, and a reduction criterion, a constraint solver is used to reduce both state spaces which results in an *incomplete configuration state space* and an *incomplete contextual state space*. This reduction approach minimizes a third disadvantage

of existing transition systems for DSPLs, i.e., the amount of states and transition of a fully specified transition system. The larger the transition system, the more complicated it is to identify an appropriate transition for a certain contextual change.

An extensive introduction of the *configuration state space* of a DSPL, the extension with *transitions* to denote reconfigurations, and an approach to *reduce* the state space is given in Chapter 6. Furthermore, the formal DSPL framework is extended with the reconfiguration semantics of a DSPL.

After the state space is reduced, the device specific feature model, the context model, and the incomplete state space are deployed on the device as the adaptation knowledge for the MAPE-K loop. The next section introduces further techniques to improve the runtime properties of the system, based on the deployed adaptation knowledge.

3.2.3 *Optimization at Runtime*

The pre-processing performed in the previous steps reduces the complexity the device has to deal with during an adaptation, i.e., the feature model and the context model contain less constraints and the state space contains less states. In addition to the pre-processing, I identified strategies to reduce the overall costs of an adaptation executed at runtime.

To improve the reconfiguration behavior on a long-term basis, the state space is used to profile the reconfiguration behavior of the device. This reconfiguration behavior is expressible with probabilistic attributes, e.g., in the context state `Office`, 33% of the contextual changes are executed to the context state `Car` and 77% of the contextual changes are executed to the context state `Home`. In addition to such behavioral information, the state space is annotatable with non-functional properties, e.g., the costs of a reconfiguration.

Based on the behavioral information, predictions about future upcoming reconfigurations are derivable, whereas the non-functional properties are exploitable to minimize the costs of a reconfiguration. Again, the behavioral information and the non-functional information of a DSPL are considered to be part of the adaptation knowledge residing on the device. However, since the reconfiguration behavior may change over the life time of a DSPL, the probability of a contextual change has to be continuously updated. Additionally, new contexts may emerge at runtime, which are not yet part of the incomplete state space. Therefore, the state space is extended if a, yet unknown, contextual situation emerges.

In the following the techniques, which are executed at runtime of a DSPL, are introduced as depicted in Figure 3.2, (08) *Reconfiguration Prediction*, and (09) Feedback for an *Online Evolution*.

(08) **Reconfiguration Prediction.** Existing approaches in the domain of behavior prediction are capable to predict the next contextual situation w.r.t. certain contextual categories such as time or geographic position [MSRJ12, Yan12, JL10]. However, it is yet unclear how a DSPL based adaptation may benefit from such predictions and how such a prediction approach may be inte-

grated into a DSPL based adaptation process. Therefore, I propose a technique to predict reconfigurations which is based on the available adaptation knowledge of a DSPL.

To achieve such a prediction, the frequency of contextual changes is profiled. This behavior is captured in a probability distribution of the contextual changes. The probability distribution provides the means to derive a prediction of future upcoming contextual changes. This prediction may be used to derive an estimation of the future upcoming requirements that are imposed by contextual changes, which, in turn, describes the demand for upcoming reconfigurations.

As previously explained, a context may be satisfiable by multiple configurations, i.e., configuration states in the state space. Together with the prediction of upcoming reconfiguration demands, this choice between configuration states for a contextual situation provides the possibility to reduce the reconfiguration costs on a long-term basis. For example, assume that the context `Office` has just become active. In this context there is a high probability that the next active context will be `Home`. Thus, if the device has to adapt to the context `Office`, a configuration should be chosen, in which the features are configured in such a way that only a minimal set of those features have to be reconfigured when the context changes to `Home`.

09 **Feedback Online Evolution.** The probability distribution of the contextual state space has to be updated with every change in the contextual situation of the device. This update is considered to be a part of the continuous online evolution of the adaptation knowledge.

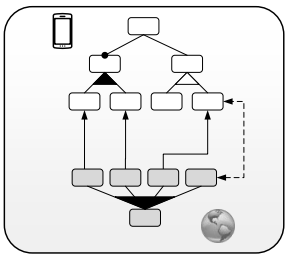
With the reduction of the state space the adaptation knowledge becomes incomplete. Thus, if a context emerges that is not yet part of the state space, a configuration state that satisfies the new context has to be derived on-demand at runtime. Every state that is discovered at runtime is added to the existing state space and the adaptation knowledge is updated respectively, e.g., the probability distribution of contextual changes is adapted to account for the new context.

The *prediction* of upcoming reconfigurations and how this information is utilizable to enhance the reconfiguration process on a long-term basis is discussed in Chapter 7. Along with this concept, the exploitation of the *online feedback* of the reconfiguration behavior and discovered states is introduced.

The next section introduces the first contributions of this thesis, i.e., the formalization of (D)SPL configuration semantics and the extension of a DSPL to an autonomous context-aware DSPL.

4

AUTONOMOUS ADAPTATION BASED ON DSPLS



The first contribution of this thesis is a formal framework to specify a DSPL-based adaptation. With this framework, I tackle the first goal G1 of this thesis, i.e., providing an *autonomous*, *model-based* adaptation process, by introducing appropriate models to describe the structure and behavior of an adaptation. Although literature provides approaches [Hel12, WDSB09], I adopt existing DSPL approaches and derive new concepts as a basis for further contributions given in this thesis.

Existing approaches are not capable to support an autonomous context-aware adaptation based on a DSPL. Traditionally, DSPL techniques are used to *reconfigure* a system [Lee06, HSSF06], i.e, to change the configuration of features. For example, a dynamic activation or deactivation of the `Navigation` feature at runtime corresponds to a reconfiguration of the respective software system. Based on the feature model specification, a DSPL is capable to identify *what* features have to be transitively reconfigured in order to reconfigure the single feature `Navigation`. Thus, transitive effects, such as the activation of the `GPS` feature if `Navigation` is activated, are considered in a DSPL reconfiguration. However, such a reconfiguration is only a part of an autonomous adaptation. An *adaptation* describes *what* has to be reconfigured as well as *how* and *when* the reconfiguration is to be executed.

To achieve an autonomous adaptation, additional rule-based systems may be used to specify *when* a reconfiguration is to be executed [HR85]. Such rule-based systems extend the knowledge base of the system with a specification such as {`<Condition> : <Act>`} rules, e.g., {*If time > 9am && time < 6pm : deactivate GPS*}. However, to establish a model-based adaptation process, I introduce a concept that supports an autonomous adaptation without the need for an *additional* rule-based system. Instead, I propose to specify contexts in a similar way as the variability of a DSPL by using a feature model. In this regard, both specifications may be combined in a single specification, which results in a context-aware DSPL. Thus, context-specific reconfiguration rules are integrated into a DSPL specification. In comparison to a traditional DSPL with an additional rule-based system, my concept has the advantage that

- the same modelling techniques are used,
- existing approaches that are applicable to a standard DSPL are also applicable for a context-aware DSPL, and
- no additional mapping between rules and a feature model is needed.

To realize such an autonomous adaptation, the feature model of a DSPL is enriched with contextual information. A feature model specifies *what* features have to be reconfigured to derive a target configuration. The extension with contextual information provides the means to specify the requirements that are imposed by a context, i.e., *how* features have to be reconfigured to satisfy a context, and *when* a reconfiguration is to be executed. For example, the context `Home` imposes the requirement *"WLAN has to be active"*. Therefore, if the user enters the context `Home` the device has to reconfigure the feature `WLAN AP` to be active.

This chapter proposes a formal framework for an autonomous context-aware adaptation of a DSPL. At first, the configuration semantics of a traditional SPL are introduced. Afterwards, an SPL is extended accordingly to derive the reconfiguration semantics of a DSPL. To provide the means for an autonomous adaptation, the concept of a traditional DSPL is enriched with contextual information to provide a seamless context-aware modelling approach.

4.1 Formal Characteristics of a DSPL

The intention of a DSPL is to describe the reconfiguration of a system at runtime. Therefore, a model is required which is capable to express *what* is to be reconfigured. For example, a reconfiguration of the feature `Cellular` from *selected* to *deselected* results in the reconfiguration of further features, e.g., `WLAN AP` has to be reconfigured from *deselected* to *selected*. Until this point of the thesis it is still unclear why such interdependencies exist and how they are resolved.

This section introduces the configuration semantics of an SPL that is based on existing literature. Afterwards, I explain how a product configuration for an SPL feature model is computable with a constraint solver. Finally, I extend and adapt the established configuration semantics of an SPL accordingly to fit my needs of a DSPL.

4.1.1 *Fundamentals of Propositional Logic*

According to Batory [Bat05], every feature model diagram is alternatively specifiable as a propositional formula. Granting that a feature model diagram is easier to model for a developer and easier to understand for third-party stakeholders, a propositional formula representation is easier to process by an algorithm.

In order to formulate axioms, theorems and proofs based on a propositional feature model, the *language of a propositional logic formula* is introduced. The alphabet consists of a set of variables $v \in \mathcal{V}$, and the constants `1` and `0`. Further, the alphabet consists of logical connective symbols $\neg$, denoting a negation, $\wedge$, denoting a logic and, $\vee$, denoting a logic or, $\veebar$, denoting a logic exclusive or, and $\rightarrow$, denoting an implication. To these parenthesis are added as auxiliary symbols.

Definition 4.1 (*Propositional Logic Alphabet [EFT94]*)**.** The alphabet of a language for propositional logic consists of the following symbols

- a set of Boolean variables $v \in \mathcal{V}$,

- a set of Boolean constants $\mathcal{W}$ with $\mathcal{W} := \{0, 1\}$,
- logical connectives $\neg$ (negation), $\wedge$ (and), $\vee$ (or), $\veebar$ (xor), and $\rightarrow$ (implication),
- parenthesis "(" and ")".

With this definition of a propositional language, a *well-formed propositional formula* over a set of variables $\mathcal{V}$ is defined as follows.

Definition 4.2 (*Well-Formed Propositional Formula [EFT94]*)**.** Let $v \in \mathcal{V}$ be a set of variables. The set $\varphi \in \Phi_{\mathcal{V}}$ of all well-formed propositional formulae over $\mathcal{V}$ is inductively defined as follows

(1) $\mathcal{V} \subseteq \Phi_{\mathcal{V}}$, i.e., every atomic variable is also a well-formed formula,

(2) $\mathcal{W} \subseteq \Phi_{\mathcal{V}}$, i.e., every atomic constant is also a well-formed formula,

(3) if $\varphi \in \Phi_{\mathcal{V}}$ is a formula, then $(\varphi) \in \Phi_{\mathcal{V}}$ is also a formula,

(4) if $\varphi \in \Phi_{\mathcal{V}}$ is a formula, then $\neg\varphi \in \Phi_{\mathcal{V}}$ is also a formula,

(5) if $\varphi_1 \in \Phi_{\mathcal{V}}$ and $\varphi_2 \in \Phi_{\mathcal{V}}$ are formulae then $(\varphi_1 \star \varphi_2) \in \Phi_{\mathcal{V}}$ with $\star \in \{\wedge, \vee, \veebar, \rightarrow\}$ is also a formula.

Formulae derived using (1) and (2) are called *atomic formulae* because they are not formed by combining other formulae $\varphi \in \Phi_{\mathcal{V}}$. The formula in (4) is called a *negation*, and the connectives in (5) $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \veebar \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$, $(\varphi_1 \leftrightarrow \varphi_2)$ are called *conjunction*, *disjunction*, *exclusive-or*, *implication*, and *logical equivalence*, respectively.

Two formulae connected by the abbreviating connective exclusive-or $(\varphi_1 \veebar \varphi_2)$ are equivalent to $((\varphi_1 \vee \varphi_2) \wedge \neg(\varphi_1 \wedge \varphi_2))$ and formulae connected by the abbreviating connective implication $(\varphi_1 \rightarrow \varphi_2)$ are equivalent to $(\neg\varphi_1 \vee \varphi_2)$. Thus, an exclusive-or is also expressible by using the connectives logic-or, negation, and logic-and. Further, an implication is also expressible by using the connectives logic-or and negation.

Definition 4.3 (*Abbreviating Connectives [EFT94]*)**.** The abbreviating connectives *exclusive-or* $(\veebar)$ and *implication* $(\rightarrow)$ between two well-formed formulae φ_1 and φ_2 are also expressible via an equivalent expression as follows.

- $\varphi_1 \veebar \varphi_2$ is equivalent to $(\varphi_1 \vee \varphi_2) \wedge \neg(\varphi_1 \wedge \varphi_2)$
- $\varphi_1 \rightarrow \varphi_2$ is equivalent to $\neg\varphi_1 \vee \varphi_2$

with $\{\varphi_1,\ \varphi_2\} \in \Phi_{\mathcal{V}}$.

In a propositional formula $\varphi \in \Phi_{\mathcal{V}}$ every occurrence of a variable $v \in \mathcal{V}$ may be replaced by a well-formed formula $\varphi \in \Phi_{\mathcal{V}}$ as Defined in 4.2. In this regard, a *variable substitution* σ is defined as follows.

Definition 4.4 (*Variable Substitution*)**.** A *substitution* $\sigma : \mathcal{V} \to \Phi$ is a replacement relation of variables $v \in \mathcal{V}$ in a propositional formula $\varphi \in \Phi_{\mathcal{V}}$, i.e.,

$$\varphi[\sigma] = \varphi'$$

where φ' corresponds to the formula φ in which every occurrence $v \in \mathcal{V}$ is replaced by $\sigma(\varphi)$.

In propositional logic, the variables $\mathcal{V}$ of a well-formed formula $\varphi \in \Phi_{\mathcal{V}}$ are assignable with the Boolean truth values `true`, denoted by `t`, and `false`, denoted by `f`, which corresponds to an *interpretation* of that formula.

Definition 4.5 (*Interpretation [EFT94]*)**.** An interpretation is a function $\mathfrak{I} : \mathcal{V} \to \{\mathtt{f}, \mathtt{t}\} \in \mathbb{B}$ that assigns either `false` (`f`) or `true` (`t`) to every propositional variable in $\mathcal{V}$.

Whether an arbitrary interpretation $\mathfrak{I}$ is a model of a formula $\varphi \in \Phi_{\mathcal{V}}$ or not is defined as follows. If $\mathfrak{I}$ is a model of φ one may also say that $\mathfrak{I}$ *satisfies* φ or that φ *holds* in $\mathfrak{I}$, which is denoted as $\mathfrak{I} \models \varphi$.

Definition 4.6 (*Satisfiability [EFT94]*)**.** The value assignment of an interpretation $\mathfrak{I}$ *either satisfies* a formula $\varphi \in \Phi_{\mathcal{V}}$, denoted as $\mathfrak{I} \models \varphi$, *or* $\mathfrak{I}$ does *not satisfy* φ, denoted as $\mathfrak{I} \not\models \varphi$.

This satisfiability relation is defined by the following inductive rules.

(1)	$\mathfrak{I} \models v$	iff $\mathfrak{I}(v) = \mathtt{t}$ holds
(2)	$\mathfrak{I} \not\models v$	iff $\mathfrak{I}(v) = \mathtt{f}$ holds
(3)	$\mathfrak{I} \models \varphi$	iff $\varphi = \mathtt{1}$,
(4)	$\mathfrak{I} \not\models \varphi$	iff $\varphi = \mathtt{0}$,
(5)	$\mathfrak{I} \models \neg\varphi$	iff not $\mathfrak{I} \models \varphi$ holds
(6)	$\mathfrak{I} \models (\varphi_1 \wedge \varphi_2)$	iff $\mathfrak{I} \models \varphi_1$ and $\mathfrak{I} \models \varphi_2$ hold
(7)	$\mathfrak{I} \models (\varphi_1 \vee \varphi_2)$	iff $\mathfrak{I} \models \varphi_1$ or $\mathfrak{I} \models \varphi_2$ (or both) hold.

An interpretation $\mathfrak{I}$ satisfies a set of propositional formulae $\Phi_{\mathcal{V}}$ iff the interpretation $\mathfrak{I}$ satisfies every single formula $\varphi \in \Phi_{\mathcal{V}}$, i.e.,

$$\mathfrak{I} \models \Phi_{\mathcal{V}} \textit{ iff } \forall \varphi \in \Phi_{\mathcal{V}} : \mathfrak{I} \models \varphi.$$

With the provided definitions, formulate axioms, theorems and proofs based on a propositional feature model may be formulated. The properties of the configuration semantics, which are based on a propositional feature model formula, are discussed in the next section.

4.1.2 *Feature Models as Propositional Formula*

In the following a feature model specification based on such well-defined propositional formulae is introduced. Further, a transformation of a feature model diagram specification into a propositional formula is given.

An SPL defines a finite set $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ of supported features. A propositional formula representation of a feature model *fm* consists of a set of feature variables $\mathcal{F}$. Further, the constraints are specified via well-formed propositional formula $\varphi \in \Phi$ over feature variables $f \in \mathcal{F}$, as defined in Definition 4.2.

Definition 4.7 (*Feature Model*)**.** A *feature model representation as a propositional formula (fm)* is a well-formed propositional formula $\Phi_{\mathcal{F}}$ over a set of feature variables $f \in \mathcal{F}$

$$fm \in \Phi_{\mathcal{F}}.$$

The set $\mathcal{FM}_{\mathcal{F}}$ of all feature model formulae over a set of supported features $\mathcal{F}$ is defined as follows

$$\mathcal{FM}_{\mathcal{F}} \;=\; \Phi_{\mathcal{F}}.$$

A feature model *fm* is *valid* iff there exists an $\mathfrak{I}$ for which holds $\mathfrak{I} \models fm$.

Hence, a feature model such as the Nexus SPL depicted in Figure 2.10 is a $fm \in \mathcal{FM}_{\mathcal{F}}$ with a specific set of constraints specified in well-formed formulae $\varphi \in \Phi_{\mathcal{F}}$ over the set of feature variables $\mathcal{F}$. I assume that the set of feature models $\mathcal{FM}_{\mathcal{F}}$ to contain only *valid* feature models.

Batory [Bat05] introduced a transformation of feature model diagrams specified as a directed graph of feature associations into a propositional logic formula. A direct translation of feature diagrams similar to those introduced in the fundamentals of this thesis into a propositional formula and vice versa is presented in [CW07]. According to [Bat05, CW07] the constraints *root feature*, *parent-child relations*, *mandatory-child relations*, *or-groups*, *alternative-groups*, and *cross-tree edges* of a feature model diagram are translated into a propositional formula by iteratively applying the following transformation rules.

- the *root feature* is mapped to the feature variable f_r, e.g., `Nexus DSPL` depicted in Figure 2.12 is mapped to a corresponding variable *Nexus DSPL*.
- *parent-child relations* are transformed into implications $f' \rightarrow f$, where f' is the child feature and f the parent feature. For example, the optional parent-child relation among `Sensors` and `GPS` is translated into (*GPS* $\rightarrow$ *Sensors*).
- *mandatory-child relations* require an additional implication $f \rightarrow f'$, where f' is a mandatory child feature of f. For example, the mandatory parent-child

relation among `Nexus DSPL` and `Routing` is translated into ((*Routing* → *Nexus DSPL*) ∧ (*Nexus DSPL*→*Routing*)).

- an *or-group* is translated into an implication to a set of disjunctive variables $f \rightarrow \bigvee_{f' \in \mathcal{F}'} f'$ for each or-group $\mathcal{F}'$ with parent feature *f*. For example, the `Sensors` or-group {`Gyroscope`, `GPS`} is translated into ((*Sensor* → (*Gyroscope* ∨ *GPS*)) ∧ (*Gyroscope* → *Sensor*) ∧ (*GPS* → *Sensor*)).
- an *alternative-group* is translated into an implication to a set of exclusive disjunctive variables $f \rightarrow \underline{\bigvee}_{f' \in \mathcal{F}} f'$ for each alternative-group $\mathcal{F}'$ with parent feature *f*, where $\underline{\vee}$ denotes an n-ary xor operator. For example, the `Routing` or-group {`BGP`, `LAR`} is translated into ((`Routing` → (*BGP* $\underline{\vee}$ *LAR*)) ∧ (*BGP* → *Routing*) ∧ (*LAR* → *Routing*)).
- *cross-tree edges* are translated as an implication for require edges and an exclusive disjunction for an exclude-edge. For example, `Cellular` requires `BGP` becomes (*Cellular* → *BGP*) and `WLAN Ad Hoc` excludes `BGP` becomes (*WLAN Ad Hoc* $\underline{\vee}$ *BGP*).

If not stated explicitly, I assume a feature model to be represented as a propositional formula $fm \in \mathcal{FM}_{\mathcal{F}}$ over feature variables $f \in \mathcal{F}$.

During the configuration of a product each feature variable $f_i \in \mathcal{F}$ in *fm* corresponds to a product configuration parameter, i.e., a Boolean variable with either $f_i = \mathtt{t}$ (true, i.e., selected), or $f_i = \mathtt{f}$ (false, i.e., deselected) as possible value bindings. A *product configuration* γ in a set of all possible configurations $\Gamma_{\mathcal{F}}$ for a feature model *fm* corresponds to an interpretation $\mathfrak{I}$ of an Boolean feature variable interpretation w.r.t. a feature model formula *fm*.

Definition 4.8 (*Product Configuration*)**.** A configuration $\gamma \in \Gamma_{\mathcal{F}}$ is an interpretation $\mathfrak{I}$ of a feature model formula *fm*. By $\Gamma_{\mathcal{F}}$ I denote the set of all interpretations of feature variables in $\mathcal{F}$.

Thus, an SPL configuration $\gamma \in \Gamma_{\mathcal{F}}$ interprets a feature $f \in \mathcal{F}$ to be either *selected* or *deselected* in a product configuration.

Notation 4.1 (Selected and Deselected Features)**.** An SPL configuration $\gamma \in \Gamma_{\mathcal{F}}$ interprets a feature $f \in \mathcal{F}$ as follows

- $\gamma(f) = \mathtt{t}$ denotes feature $f \in \mathcal{F}$ to be selected, and
- $\gamma(f) = \mathtt{f}$ denotes feature $f \in \mathcal{F}$ to be deselected.

A configuration $\gamma \in \Gamma_{\mathcal{F}}$ defines an interpretation for the feature variables $f \in \mathcal{F}$. To evaluate a the feature model formula *fm*, the variables $f \in \mathcal{F}$ have to be substituted with propositional constants or variables in $\Phi_{\mathcal{F}}$ according to the interpretation provided by a configuration γ. According to the Definition 4.4, the *substitution of a configuration interpretation* is defined as follows.

Definition 4.9 (*Configuration Substitution*)**.** A configuration substitution $fm[\sigma_\gamma] = fm'$ with $\gamma \in \Gamma_\mathcal{F}$ is a substitution of feature variables $\mathcal{F}$ with propositional constants or variables in $\Phi_\mathcal{F}$ according to the configuration interpretation γ of a feature model formula *fm*.

Thereby, a configuration substitution $[\sigma_\gamma]$ is executed as follows

- $\forall f \in \mathcal{F} : fm[\sigma_\gamma](f) = \mathtt{1}$ iff $\gamma(f) = \mathtt{t}$ and
- $\forall f \in \mathcal{F} : fm[\sigma_\gamma](f) = \mathtt{0}$ iff $\gamma(f) = \mathtt{f}$ otherwise.

A configuration $\gamma \in \Gamma_\mathcal{F}$ represents an arbitrary interpretation of the feature variables $f \in \mathcal{F}$. However, with regard to the propositional formula *fm* only a subset of these configurations represent a valid configuration, denoted by $\gamma \in \hat{\Gamma}_{fm}$. A configuration γ is *valid* w.r.t. *fm* iff the feature variable substitution in $\varphi[\gamma]$ becomes a true statement for every formula $\varphi \in \Phi$.

Definition 4.10 (*Valid Configuration*)**.** A configuration $\gamma \in \Gamma_\mathcal{F}$ is a *valid* configuration for a feature model *fm* iff $\gamma \models fm$.

The subset of all valid configurations is denoted as $\hat{\Gamma}_{fm} \subseteq \Gamma_\mathcal{F}$.

Example 4.1 (*Valid Configuration of a Feature Model Formula*).

A valid configuration $\gamma_1 \in \hat{\Gamma}_{fm}$ for the feature variables $\mathcal{F}$ of the Nexus DSPL is the following interpretation (γ_1*(Nexus DSPL)*=t, γ_1*(Connectivity)*=t, γ_1*(Routing)*=t, γ_1(*Ad Hoc*)=f, γ_1*(Infrastructure)*=t, γ_1*(WLAN Ad Hoc)*=f, γ_1*(WLAN AP)*=f, γ_1*(BGP)*=t, γ_1*(Cellular)*=t, γ_1*(LAR=)*f, γ_1*(GPS)*=t, γ_1*(Game Hub)*=f, γ_1*(Phone Call)*=t, γ_1*(Cellular Call)*=t, γ_1*(Sensors)*=t, γ_1*(Gyroscope)*=f, γ_1*(Application)*=t, γ_1*(Navigation)*=t, γ_1*(VoIP)*=f). Therefore the configuration satisfies the constraints of the feature model, i.e., $\gamma_1 \models fm$.

However, some interpretations lead to a contradiction of the constraints specified in the Nexus DSPL. For example, with the extract of a configuration $\gamma_2 \in \Gamma_\mathcal{F}$ (γ_2*(BGP)*=t, γ_2*(LAR)*=t, ...) the formula φ=(BGP $\veebar$ LAR) corresponds to a false statement. Therefore, $\gamma_2 \not\models fm$.

Summarizing, I introduced the following properties for an SPL

- a feature model specification of an SPL is expressible as a propositional formula *fm*,
- a propositional formula *fm* corresponds to a set of conjunctive propositional formulae $\{\varphi_1 \wedge \ldots \wedge \varphi_n\}$ with $\{\varphi_1, \ldots, \varphi_n\} \in \Phi$ over Boolean feature variables $\mathcal{F}$,
- a configuration γ is a Boolean interpretation of the feature variables, i.e., $\gamma : \mathcal{F} \to \mathbb{B}$, in a feature model *fm*,
- a feature *f* is *selected* iff $\gamma(\mathtt{f}) = \mathtt{t}$,

- a feature *f* is *deselected* iff $\gamma(\mathtt{f}) = \mathtt{f}$,
- a configuration γ is valid, iff the feature-value interpretation satisfies every formula $\varphi \in \Phi$, i.e., $\gamma \models \mathit{fm}$

The formalization of feature models based on propositional formulae provides the necessary means to define a product configuration of an SPL and whether a configuration is valid or not. Further, it provides the basis to automatically derive a configuration by constraint solving techniques. The next section discusses how constraint solving is used to derive a product configuration.

4.1.3 *Derivation of a Configuration with a Constraint Solver*

Satisfiability of a propositional formula *fm* is the problem of finding all configuration interpretations $\gamma \in \Gamma_{\mathcal{F}}$ for a formula *fm* in which $\gamma \models \mathit{fm}$ holds. The solving of a formula is an NP-complete problem [GKSS08] and, therefore, multiple executions of a solver to (continuously) derive a configuration consumes a lot of time and computational effort.

In the following, binary decision diagrams are introduced as a category of satisfiability solvers. A Binary Decision Diagram (BDD) solver relies on a propositional formula which is transformed into an ordered tree-like diagram.

Solvers which rely on a BDD as internal representation of the satisfiability problem have proven to be efficient if the problem is very complex, i.e., if the amount of variables and disjunctive clauses is very high. In contrast to classic satisfiability algorithms that rely on backtracking [ES04, SS96], BDD Solvers compute faster and more efficiently all possible interpretations that satisfy the respective propositional formula [PG96].

A BDD is a tree-based structure which is usable to represent a propositional formula. The nodes within the acyclic graph represent variables and the constraint relations amongst variables are represented via labeled (*true* or *false*) edges. Figure 4.1(a) depicts a BDD representation of the features `Sensors`, `GPS`, and `Gyroscope` from the Nexus DSPL depicted in Figure 2.12.

A BDD has always one root node and several leave nodes that represent the satisfiability of the propositional formula. The root node and every intermediate node have an out-degree of two. One edge represents a `true` interpretation of the node, i.e., the corresponding variable, and the second edge represents a `false` interpretation of the node.

The BDD depicted in Figure 4.1(a) contains redundant information which may lead to unnecessary overhead during the process of solving feature model formula. To enhance the efficiency of the solving process a BDD is transformed to a reduced order BDD (ROBDD) [Ake78]. This transformation exploits the fact that a solver is usually used to compute a valid variable assignment which satisfies the formula.

Thus, the information contained in the left sub-tree of Figure 4.1(a) is completely encoded in the right sub-tree. Removing such redundant information results in a ROBDD without loosing any specification characteristics in the feature model formula *fm*, as depicted in Figure 4.1(b).

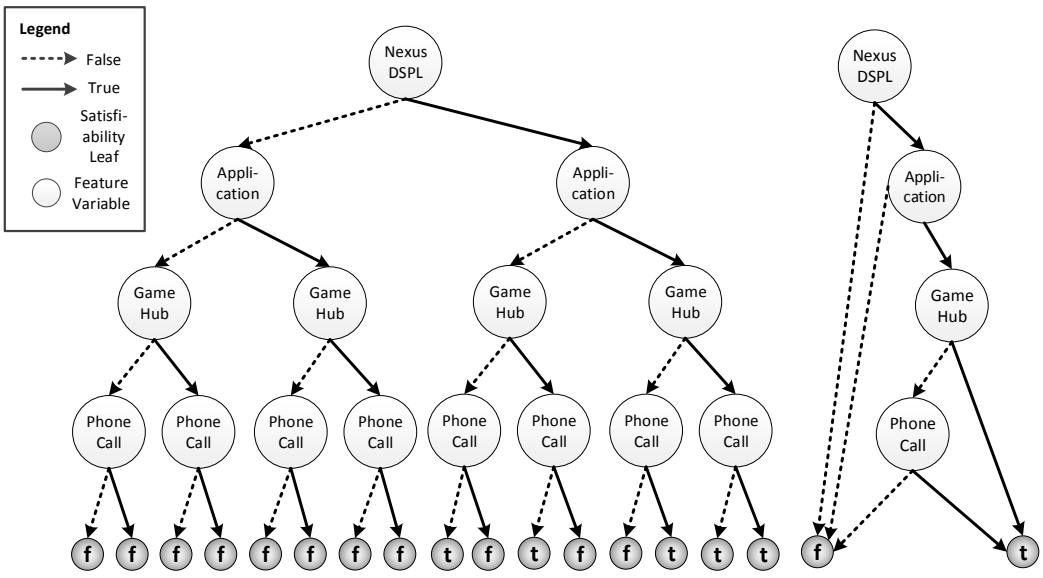


Figure 4.1: Extract of Nexus DSPL as a BDD

Example 4.2 (*Feature Model as BDD*).

Figure 4.1 depicts the BDD representation for the following variability specification as propositional formula.

$$\varphi = (\mathrm{NexusDSPL}) \wedge (\mathrm{Application} \rightarrow \mathrm{NexusDSPL}) \wedge (\mathrm{Application} \rightarrow (\mathrm{Game\ Hub} \vee \mathrm{Phone\ Call}))$$

The root node, `Nexus DSPL`, and the remaining intermediate nodes are depicted in white. A dotted outgoing arrow indicates a feature variable interpretation of *false* and a solid arrow indicates a variable of *true*. The grey leaf nodes indicate whether the variable interpretation of every incoming path satisfies the propositional formula φ (represented by (t)) or not (represented by (f)). Thus, if `Nexus DSPL` is interpreted as deselected (`f`) the formula φ is never satisfiable, whereas the interpretation of `Nexus DSPL`, `Application`, `Game Hub`, and `Phone Call` as selected (`t`) satisfies the formula φ.

However, the concept of deriving a *single configuration* has to be extended. For DSPLs, it is necessary to describe a *continuous reconfiguration*, i.e., invoking a solver call for each reconfiguration. Therefore, I introduce DSPL specific concepts that extend the traditional SPL concepts in the next section.

4.1.4 *DSPL Extension*

Mobile devices have to adapt to the context of their surroundings. To execute such an adaptation a DSPL has to be reconfigurable at runtime. However, this variability may be restricted by the characteristics of the mobile device. For ex-

ample, the Nexus 7 does not have a built-in rear-camera. Therefore, the feature model of the DSPL has to be restricted to the characteristics of the device.

Further, each context imposes certain requirements on the device, as previously introduced. Thus, a DSPL hast to be capable to execute a reconfiguration based on the requirements of the active context. For example, the context `Car` requires the feature `Navigation` to be active. As previously explained, a DSPL may also be customized to the characteristics of a specific device. Both, device specific requirements and contextual requirements, correspond to a *partial configuration*.

To provide the necessary means to express such DSPL specific characteristics, the configuration semantics of an SPL have to be extended. Therefore, the basics of three-valued propositional logics are introduced at first. In a *partial configuration*, features may not only be explicitly bound but left unconfigured, e.g., for a further *refinement* at design time w.r.t. the requirements of a developer or at runtime w.r.t. the requirements imposed by a context. Basis of such a partial configuration and a configuration refinement is a three-valued logic. An description of the necessary DSPL terminology is given in the following.

THREE-VALUED PROPOSITIONAL LOGIC. To describe the dynamic aspects of a DSPL (re-)configuration, the Definition 4.8 of an SPL product configuration needs to be extended. A DSPL product configuration interprets each feature variable $f \in \mathcal{F}$ by means of a (three-valued) Boolean value, where value $\bot$ represents a (yet) undecided interpretation. With the extension of the value domain of a configuration from $\mathbb{B}$ to $\mathbb{B}_\bot$, the Definitions 4.5 and 4.6 of an interpretation and satisfiability have to be adapted accordingly. To establish the fundamentals on three-valued propositional logic, an order of the values $\{\mathtt{f}, \bot, \mathtt{t}\}$ in the three-valued logic domain $\mathbb{B}_\bot$ and the applicability of the standard logic operator *negation* are introduced as follows.

Definition 4.11 (*Properties of Three-Valued Logics*)**.** The domain of three-valued Boolean values is denoted as

$$\mathbb{B}_\bot = \{\mathtt{f}, \bot, \mathtt{t}\}.$$

The sub domain of Boolean values is denoted as $\mathbb{B} = \{\mathtt{f}, \mathtt{t}\} \subset \mathbb{B}_\bot$. Further, a total order

$$\mathtt{f} < \bot < \mathtt{t}$$

is defined on $\mathbb{B}_\bot$. It holds $\neg\mathtt{t} = \mathtt{f}$ and $\neg\mathtt{f} = \mathtt{t}$ as usual as well as $\neg\bot = \bot$.

Thereupon, further logical connectives for three-valued logics such as implication are defined, accordingly. Note that the sub domain $\mathbb{B} = \{\mathtt{f}, \mathtt{t}\} \subset \mathbb{B}_\bot$ denotes Boolean values as in the standard propositional logic of an SPL.

In three-valued propositional logic, the variables $\mathcal{V}$ of a well-formed formula $\varphi \in \Phi_\mathcal{V}$ are assignable with the Boolean truth values `true`, denoted by `t`, `false`, denoted by `f`, and *unknown*, denoted by $\bot$, which corresponds to a three-valued *interpretation* of that formula.

Definition 4.12 (*Three-Valued Interpretation*)**.** An interpretation is a function $\mathfrak{I} : \mathcal{V} \to \{\mathtt{f}, \bot, \mathtt{t}\} \in \mathbb{B}_\bot$ that assigns either `false` (`f`), *unknown* ($\bot$), or `true` (`t`) to every propositional variable in $\mathcal{V}$.

The satisfiability as defined in Definition 4.6 has to be adapted for a three-valued interpretation $\mathfrak{I}$ as follows. The properties of a standard satisfiability still apply for a three-valued satisfiability. However, with this definition it is not possible to evaluate whether $\gamma \models \mathit{fm}$ holds or not if the interpretation of at least one variable $v \in \mathcal{V}$ is still unknown, i.e., $\exists v \in \mathcal{V} : \mathfrak{I}(v) = \bot$. Therefore, a formula $\varphi \in \Phi_\mathcal{V}$ may be both, satisfiable and unsatisfiable, if a variable is unknown $\mathfrak{I}(v) = \bot$. Thereby, an Interpretation of a formula is satisfiable as long as an atomic formula $\varphi \in \Phi_\mathcal{V}$ is not solely unsatisfiable.

Definition 4.13 (*Three-Valued Satisfiability*)**.** The value assignment of a three-valued interpretation $\mathfrak{I}$ satisfies a formula $\varphi \in \Phi_\mathcal{V}$, denoted as $\mathfrak{I} \models \varphi$, or $\mathfrak{I}$ does not satisfy φ, denoted as $\mathfrak{I} \not\models \varphi$.

The definition of a three-valued satisfiability relation is adapted w.r.t. Definition 4.6 according to the following inductive rules.

(1)	$\mathfrak{I} \models v$	iff $\mathfrak{I}(v) \geqslant \bot$ holds
(2)	$\mathfrak{I} \not\models v$	iff $\mathfrak{I}(v) \leqslant \bot$ holds
(3)	$\mathfrak{I} \models \varphi$	iff $\varphi = \mathtt{1}$,
(4)	$\mathfrak{I} \not\models \varphi$	iff $\varphi = \mathtt{0}$,
(5)	$\mathfrak{I} \models \neg\varphi$	iff $\mathfrak{I} \not\models \varphi$ holds
(6)	$\mathfrak{I} \not\models \neg\varphi$	iff $\mathfrak{I} \models \varphi$ holds
(7)	$\mathfrak{I} \models (\varphi_1 \wedge \varphi_2)$	iff $\mathfrak{I} \models \varphi_1$ and $\mathfrak{I} \models \varphi_2$ hold
(8)	$\mathfrak{I} \not\models (\varphi_1 \wedge \varphi_2)$	iff either $\mathfrak{I} \not\models \varphi_1$ or $\mathfrak{I} \not\models \varphi_2$ (or both) hold
(9)	$\mathfrak{I} \models (\varphi_1 \vee \varphi_2)$	iff either $\mathfrak{I} \models \varphi_1$ or $\mathfrak{I} \models \varphi_2$ (or both) hold
(10)	$\mathfrak{I} \not\models (\varphi_1 \vee \varphi_2)$	iff $\mathfrak{I} \not\models \varphi_1$ and $\mathfrak{I} \not\models \varphi_2$ hold.

Thus, a three-valued interpretation $\mathfrak{I}$ satisfies a set of propositional formulae $\Phi_\mathcal{V}$ iff the interpretation $\mathfrak{I}$ satisfies every single formula $\varphi \in \Phi_\mathcal{V}$, i.e.,

$$\mathfrak{I} \models \Phi_\mathcal{V} \textit{ iff } \forall \varphi \in \Phi_\mathcal{V} : \mathfrak{I} \models \varphi.$$

In comparison with Definition 4.6, I adapted the satisfiability of a variable interpretation $v \in \mathcal{V}$ in such a way that the formula may be satisfiable as well as unsatisfiable if the interpretation of a variable is unknown. Further, explicit rules are added if a formula φ is not satisfiable, i.e., rules (6), (8), and (10), to the definition of a three-valued satisfiability.

3-VALUED DSPL CONFIGURATION. For the three-valued configuration semantics, each feature variable $f \in \mathcal{F}$ occurrence in the formulae $\varphi \in \Phi_{\mathcal{F}}$ is substituted by $\gamma(f)$ iff $\gamma(f) \neq \bot$. If $\gamma(f) = \bot$ the feature is *not* substituted in any of the formulae $\varphi \in \Phi$ and is still (re-)configurable. To describe such an interpretation of a DSPL configuration, the three-valued configuration semantics is extended as follows.

Definition 4.14 (*Three-Valued Configuration Semantics*)**.** Let $fm \in \Phi_{\mathcal{F}}$ be a feature model. A three-valued configuration γ corresponds to a three-valued interpretation of feature variables $f \in \mathcal{F}$ with $\gamma : \mathcal{F} \to \mathbb{B}_{\bot}$.

Thus, in three-valued configuration a feature $f \in \mathcal{F}$ is interpretable in three different ways, i.e.,

- $\gamma(f) = \mathtt{f}$,
- $\gamma(f) = \bot$, and
- $\gamma(f) = \mathtt{t}$.

With such a three-valued configuration interpretation $\gamma \in \Gamma_{\mathcal{F}}$, the variables in a propositional feature model formula are substituted as follows. As in standard configuration interpretation, a feature $f \in \mathcal{F}$ is substituted by `1` if $\gamma(f) = \mathtt{t}$ and by `0` if $\gamma(f) = \mathtt{f}$. However, if a feature remains unconfigured $\gamma(f) = \bot$, the feature variable f is not substituted.

Definition 4.15 (*Three-Valued Configuration Substitution*)**.** A configuration substitution $fm[\sigma_\gamma] = fm'$ with $\gamma \in \Gamma_{\mathcal{F}}$ is a substitution σ that replaces the feature variables $\mathcal{F}$ with propositional constants or variables in $\Phi_{\mathcal{F}}$ according to the configuration interpretation γ of a feature model formula fm, i.e.,

$$fm[\sigma_\gamma] = fm' : \mathcal{F} \to \Phi_{\mathcal{F}}.$$

Thereby, a configuration substitution $[\sigma_\gamma]$ is executed as follows

- $\forall f \in \mathcal{F} : fm[\sigma_\gamma](f) = \mathtt{1}$ iff $\gamma(f) = \mathtt{t}$ and
- $\forall f \in \mathcal{F} : fm[\sigma_\gamma](f) = \mathtt{0}$ iff $\gamma(f) = \mathtt{f}$, and
- $\forall f \in \mathcal{F} : fm[\sigma_\gamma](f) = f$ otherwise.

Thus, if a configuration interprets a feature variable as *unconfigured*, the variable is *not* substituted.

RECONFIGURABILITY AT RUNTIME. A configuration that is passed to the execution task of the MAPE-K loop has to be a valid configuration w.r.t. Definition 4.10. In such a configuration, every feature is interpreted as either selected

(t) or deselected (f). For a DSPL, it has to be differentiated between the configuration semantics of a feature.

At design time, a feature is selected or deselected for a product configuration in a *static* manner in accordance with Definition 2.1. A selected feature is permanently available at runtime and a deselected feature is not even deployed on the device. Such a configuration decision is permanent and may not be altered after the configuration process is finished.

The previously established interpretation of a configuration has to be extended to capture the dynamic aspect of a DSPL. In a DSPL a feature configuration is not necessarily static. Instead a feature may be dynamically reconfigurable at runtime in accordance with Definition 2.2 of a *dynamic feature*. To denote *reconfigurable* features at runtime, I introduce a new configuration terminology. At runtime, a reconfigurable feature is either *active*, e.g., the `Navigation` initialized and running, or it is *inactive*, e.g., the feature `Navigation` is idle and not in the main memory. Further, I refer to the valid configuration, which is currently executed at runtime, as *runtime configuration*.

A feature that is configured as deselected at design time is not available on the device, whereas a selected feature is *always* active at runtime. Thus, if a feature is selected or deselected it is *not* reconfigurable. In contrast to that, an inactive feature is a reconfigurable feature that is idle and not part of the current runtime configuration. An active feature is a reconfigurable feature that is part of the current runtime configuration.

In this regard, similar to an SPL, a DSPL defines a finite set $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ of supported features. During product (re-)configuration each feature variable $f_i \in \mathcal{F}$ corresponds to a product configuration parameter, i.e., a Boolean variable with either $f_i = \texttt{t}$ (true, i.e., active), or $f_i = \texttt{f}$ (false, i.e, inactive) as possible value bindings. The interpretation of a feature as active and inactive is not permanent but may be altered at runtime. In this regard, the Definition 4.10 of a valid configuration still applies to a DSPL, but features are assumed not to be selected but active and not to be deselected but inactive.

Notation 4.2 (Active and Inactive Features)**.** For a DSPL a valid configuration $\gamma \in \Gamma_{fm}$ interprets a feature variable $f \in \mathcal{F}$ at runtime as follows

- $\gamma(f) = \texttt{t}$ denotes feature $f \in \mathcal{F}$ to be active in γ, and
- $\gamma(f) = \texttt{f}$ denotes feature $f \in \mathcal{F}$ to be inactive in γ.

During a reconfiguration of a DSPL the interpretation of a set of feature variables in $\gamma \in \Gamma_{fm}$ is changeable to a subsequent configuration $\gamma' \in \Gamma_{fm}$. Thus, in contrast to the Notation 4.1 of (de-)selected features, an active feature is dynamically reconfigurable to be inactive and vice versa.

Example 4.3 (*Reconfigurability at Runtime*).

At runtime, the `Display Driver` are permanently active, e.g., the display driver feature `4,7` on a Nexus 4 device. Therefore, such a feature is selected at design time for a configuration that is specific for a Nexus 4 device.

In contrast to that, the features `WLAN AP` and `Cellular` of the Nexus DSPL are dynamically reconfigurable at runtime. They are constrained in an alternative group and, therefore, `WLAN AP` is active if `Cellular` is inactive and vice versa. If no WLAN access point is available in the surroundings of the device, a reconfiguration is executed to establish a connection via a cellular network. In this regard `Cellular` becomes active and `WLAN AP` becomes inactive.

PARTIAL CONFIGURATION. A three-valued DSPL configuration semantics provides the means to include partial configurations in the (re-)configuration process of a DSPL. In a standard SPL product configuration or a valid DSPL configuration, every feature variable is interpreted with a Boolean configuration value w.r.t. a valid configuration $\gamma \in \Gamma_{fm}$. However, DSPLs are (re-)configurable over multiple steps. At design time a DSPL is pre-configured according to the characteristics of the target device, e.g., features are bound to be deselected that are not supported by the device. Features that have to be dynamically reconfigurable as active and inactive at runtime are not pre-configured at design time and remain unconfigured. Thus, a DSPL that is deployed on a device is *partially configured* for a device. Further, a DSPL has to satisfy the requirements of a continuously changing context at runtime. Each context imposes specific requirements on a set of features, e.g., the context *car* imposes the requirement *"Navigation has to be active"*, whereas the remaining features are irrelevant for that context. Features that have not yet been configured are *unconfigured*, i.e., neither active nor inactive. Such features remain unconfigured until in a further *refinement* a configuration decision is made.

Notation 4.3 (Unconfigured Features)**.** A yet undecided configuration interpretation for feature $f \in \mathcal{F}$ is denoted by $\gamma(f) = \bot$, i.e., the feature f is *unconfigured*.

The introduction of an unconfigured features in the configuration semantics of a DSPL results in *partial*, yet configurable product configurations in $\Gamma_{\mathcal{F}}$. The subset of all valid partial configurations $\check{\Gamma}_{fm} \subseteq \Gamma_{\mathcal{F}}$ consists only of those partial configurations that satisfy the feature model specification *fm* and may still lead to a valid configuration. To distinguish between such variable configurations and a complete valid configuration, the definition of a *partial configuration* is introduced as follows.

Definition 4.16 (*Partial Configuration*)**.** A configuration $\gamma \in \Gamma_{\mathcal{F}}$ is an interpretation $\gamma : \mathcal{F} \to \mathbb{B}_{\bot}$ that assigns a three-valued Boolean value $\mathbb{B}_{\bot} = \{\mathtt{f}, \bot, \mathtt{t}\}$ to every feature variable $f \in \mathcal{F}$ of a propositional formula *fm*.

A configuration $\gamma \in \Gamma_{\mathcal{F}}$ is a *partial configuration* iff

$$\exists f \in \mathcal{F} : \gamma(f) = \bot.$$

Further, a configuration $\gamma \in \check{\Gamma}_{\mathtt{fm}}$ is a *valid partial configuration* iff

$$\exists f \in \mathcal{F} \,:\, \gamma(f) = \bot \text{ and } \gamma \models \mathit{fm}$$

holds.

The set of all valid partial configurations is denoted as

$$\check{\Gamma}_{\mathit{fm}}.$$

Thus, the valid set of *partial* configurations is denoted as $\check{\Gamma}_{\mathtt{fm}}$ and the set of all valid *complete* configurations is denoted as $\hat{\Gamma}_{\mathit{fm}}$ in the remainder of this thesis. Furthermore, I denote the set of all valid (partial *and* complete) configurations as

$$\Gamma_{\mathit{fm}} \;=\; \check{\Gamma}_{\mathit{fm}} \cup \hat{\Gamma}_{\mathit{fm}}$$

in the remainder of this thesis.

Example 4.4 (*Partial Configuration*).

A context states a set of requirements that are of relevance for the contextual situation. For instance, the context *Office* imposes the requirement *"To establish a connection, an* `Infrastructure`*-based communication has to be active and* `Game Hub` *has to be inactive"*. The remaining features such as `Phone Call`, `VoIP`, and `GPS` remain unconfigured. Therefore, this contexts corresponds the partial configuration $\gamma_{\mathrm{p}} \in \check{\Gamma}_{\mathit{fm}}$ for the feature variables $\mathcal{F}$ with the following interpretation (γ_{p}*(Nexus DSPL)=*`t`, γ_{p}*(Connectivity)=*`t`, γ_{p}*(Routing)=*`t`, γ_{p}*(Ad Hoc)=*`f`, γ_{p}*(Infrastructure)=*`t`, γ_{p}*(WLAN Ad Hoc)=*`f`, γ_{p}*(WLAN AP)=*`f`, γ_{p}*(Cellular)=*`t`, γ_{p}*(BGP)=*`t`, γ_{p}*(LAR)=*`f`, γ_{p}*(Sensors)=*`t`, γ_{p}*(Game Hub)=*`f`, γ_{p}*(Phone Call)=*$\bot$, γ_{p}*(Cellular Call)=*$\bot$, γ_{p}*(GPS)=*$\bot$, γ_{p}*(Gyroscope)=*`f`, γ_{p}*(Application)=*`t`, γ_{p}*(Navigation)=*$\bot$, γ_{p}*(VoIP)=*$\bot$).

Unconfigured features, such as `Cellular Call` and `VoIP`, specify the reconfiguration possibilities at runtime, i.e., executing a phone call either via the cellular network or via VoIP.

A configuration $\gamma \in \Gamma_{\mathcal{F}}$ defines an interpretation for the feature variables $f \in \mathcal{F}$. To evaluate a the set of conjunctive propositional formulae specifying a feature model *fm*, the variables $f \in \mathcal{F}$ have to be substituted with propositional constants or variables in $\Phi_{\mathcal{F}}$ according to the interpretation provided by a configuration γ.

CONFIGURATION REFINEMENT. Every partial configuration containing yet unconfigured features may be *refined* until all features are bound. Such a refinement process is executable over multiple steps. In each refinement step, unconfigured features are bound to be selected or deselected at design time and bound to be active or inactive at runtime. Therefore, refinements further restrict the amount of valid configurations until a single configuration remains, in which every feature is bound. Thus, every partial configuration is refined until a valid configuration is derived that is applicable for a (re-)configuration of the corresponding device at runtime. This induces an implicit notion of a configuration

refinement relation. Thus, a configuration γ' is a refinement of γ if every bound feature in γ is equally interpreted in γ'.

Definition 4.17 (*Configuration Refinement*)**.** Configuration $\gamma' \in \Gamma_{\mathcal{F}}$ is a *refinement* of configuration $\gamma \in \Gamma_{\mathcal{F}}$, denoted by $\gamma' \sqsubseteq \gamma$, iff

$$\forall f \in \mathcal{F} : \gamma(f) \in \mathbb{B} \Rightarrow \gamma'(f) = \gamma(f)$$

holds.

Thereupon, configuration γ' may also define further interpretations of features that are not yet configured in γ.

Example 4.5 (*Value Substitution of a partial Configuration in a Feature Model* fm). —— The feature variables of the formula $\varphi_1 \in \Phi$, with $\varphi_1 = (\mathit{Sensors} \rightarrow (\mathit{GPS} \vee \mathit{Gyroscope}))$, in the Nexus DSPL are substituted according to the partial configuration γ_p, with γ_p*(Sensors)*=`t`, γ_p*(Gyroscope)*=`f`, and γ_p*(GPS)*=$\bot$, as follows.

$$\varphi_1(\gamma_p) = (\mathtt{1} \rightarrow (\mathit{GPS} \vee \mathtt{0})$$

Thus, unconfigured features remain in the propositional formula of a feature model *fm* until they are configured to be active or inactive in a further refinement. For example, γ_r, with γ_r*(Sensors)*=`t`, γ_r*(Gyroscope)*=`f`, and γ_r*(GPS)*=`t`, is a refinement of γ_p, i.e., $\gamma_r \sqsubseteq_{\mathcal{F}} \gamma_p$. Note that $\gamma_p \notin \hat{\Gamma}_{fm}$ because it is a partial configuration.

Summarizing, I consider the following properties to hold for three-valued DSPL configuration semantics.

- the Boolean value domain $\mathbb{B}$ of a traditional SPL is extended for a DSPL to a three-valued domain $\mathbb{B}_\bot$ with a total order of $\mathtt{f} < \bot < \mathtt{t}$,
- feature *f* is an *active* feature iff $\gamma(\mathtt{f}) = \mathtt{t}$, with $f \in \mathcal{F}$ and $\gamma \in \Gamma_{\mathcal{F}}$,
- feature *f* is an *unconfigured* iff $\gamma(\mathtt{f}) = \bot$, with $f \in \mathcal{F}$ and $\gamma \in \Gamma_{\mathcal{F}}$
- feature *f* is an *inactive* feature iff $\gamma(\mathtt{f}) = \mathtt{f}$, with $f \in \mathcal{F}$ and $\gamma \in \Gamma_{\mathcal{F}}$
- a configuration $\gamma \in \hat{\Gamma}_{fm}$ is a valid runtime configuration iff $\forall f \in \mathcal{F} : \forall \gamma(f) \in \mathbb{B} \wedge \gamma \models \mathit{fm}$,
- a configuration $\gamma \in \check{\Gamma}_{fm}$ is partial and valid iff $\forall f \in \mathcal{F} : \exists \gamma(f) = \bot \wedge \gamma \models \mathit{fm}$ holds, and
- a configuration γ' is a refinement of a partial configuration γ, i.e., $\gamma' \sqsubseteq \gamma$, iff $\forall f \in \mathcal{F} : \gamma(f) \in \mathbb{B} \Rightarrow \gamma'(f) = \gamma(f)$ holds.

Although this formalization provides the necessary notation to describe a reconfiguration, i.e., *what* has to be adapted, it still remains unaddressed *when* a reconfiguration is triggered and *how* the requirements of a context affect that reconfiguration. Therefore, the concept of a DSPL is extended accordingly to handle changes in the contextual situations autonomously in the next section.

4.2 Context-Sensitive DSPLs

Existing DSPL approaches do not support an autonomic context-aware adaptation of mobile devices. The reconfiguration concept of a DSPL alone is not capable to execute an *autonomous adaptation*. A DSPL must be aware of its context in order to trigger a reconfiguration to decide when and how to adapt. Therefore, a DSPL has to be aware of which requirements are imposed by a contextual situation.

In the fundamentals of this thesis, I introduce context as a concept to describe any (external or internal) information which is computationally accessible and triggers the adaptations of the device [CLG+09]. Thus, a mobile device is able to detect changes in its context. For example, a Nexus 4 smartphone is able to detect the context `Home` according to the GPS position of the device or the WLAN access point. Further, I explained that a context imposes certain requirements on a device, e.g., the context `Car` requires *"the navigation system must be active"*. However, to trigger a corresponding reconfiguration of a DSPL, contexts have to be associated to features.

To enable an autonomous reconfiguration, this section introduced an integration of contextual information into DSPLs. It is discussed how a reconfiguration is executable autonomously based on a context model. Further, an integration of the context model into the existing definition of a feature model is introduced.

4.2.1 *Contexts to Execute an Autonomous Reconfiguration*

Every autonomous feature reconfiguration of a DSPL is based on the change between the currently active context and the target context. However, a contextual situation may be a combination of several contexts, e.g., the user is at *Home* working in his *Office*. Thus, it is possible that multiple contexts are active at the same time. Multiple contexts are compatible if the requirements of every single context are combinable to a set of non-contradicting requirements that has to be satisfied by the device. In contrast to that, multiple contexts are incompatible, if the requirements impose a contradiction to the specification of the DSPL.

A context aggregates information, which is computationally accessible, to requirements that are to be satisfied by the system. Multiple contexts are combinable iff the requirements do not contradict each other. Changes in the set of active contexts trigger a reconfiguration of the system. Thus, a context specifies requirements for a system, i.e., *how* specific features have to be configured. For example, the context `Car` requires *"the navigation system must be active"*. Therefore, the feature `Navigation` has to be active if this context is being entered. In contrast to that, features describe distinctive user-visible aspects, quality, or characteristics of system [KCH+90].

For a DSPL reconfiguration to be performed autonomously, an appropriate interface for recognizing context (de-)activations is to be provided. In this thesis, I abstract from the technical details and assume corresponding signaling events to be triggered by changes in the contextual environment, e.g., emitted by the build-in sensors of a device [BHS12]. For example, the context may be influenced by the available network capacity, the connectivity, the geographic position,

available services, and the current time [ADB+99, SLP04]. All of these aspects are sensable by a mobile device and, therefore, accessible for a derivation of the contextual situation.

Example 4.6 (*Contexts to Trigger Reconfigurations*).

Let us assume that the currently active context of a smartphone is `Office`. This contexts implies the following requirements *"(i) WLAN must be active, (ii) Phone Calls must be executable, (iii) Game Hub has to be deactivated"*. Hence, the smartphone is currently in a configuration state that satisfies these requirements.

The context `Car` imposes the following requirements *"(i) WLAN has to be deactivated, (ii) Navigation has to be activated"*. If the user leaves the office and enters a car, the device reconfigures itself accordingly, e.g., WLAN is deactivated and the Navigation system with GPS is activated. Both contexts are incompatible, thus they may not be active at the same time.

The smartphone recognizes if the context `Office` becomes inactive, if the respective WLAN access point is not available anymore. Similarly, the device recognizes if the context `Car` is active, if it is connected to the docking station of the car. If such a context becomes active or inactive, the device emits an event to the DSPL.

Summarizing, each particular context is associated with logical requirements, e.g., require-dependencies among contexts and features such as `Home` $\rightarrow$ `WLAN AP`, or a conflicts, i.e., excludes, such as `Office` $\rightarrow \neg$`Game Hub`. It is further possible for multiple contexts to be active simultaneously in arbitrary combinations imposing interfering requirements to features. For example, the contexts `Home` and `Office` may be active at the same time. Hence, contexts are organizable in a similar way as features are arranged in a feature model and by adopting context modeling techniques [ACG09, BBH+10, HT08].

4.2.2 *Integration of a Context-Model into a DSPL*

To autonomously reason about context-aware runtime (re-)configurations, I enrich a feature model $fm \in \mathcal{FM}_{\mathcal{F}}$ of a DSPL by integrating a further context model. Such a context model consists a finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ of atomic contexts potentially emerging at runtime as well as the additional sets of well-formed formulae to specify constraints between contexts and features, e.g., exclude and require, and constraints between contexts, e.g., an organization of contexts in an or-group. Therefore, the definition of a feature model according to Definition 4.7 is extended with additional context information, again represented as a feature model over contexts, i.e., a *context model*.

Figure 4.2 depicts the extension of the Nexus DSPL running example by enriching its feature model *fm* (left hand side) with a context model (right hand side) resulting in a context-aware feature model *cfm*. In this model, all contexts are organized in an *or-group* which allows an arbitrary combination of contexts and requires at least one context to be active at runtime. Additional constraints among contexts may be specified which is not further considered at this point.

Example 4.7 (*Context-Aware DSPL*).

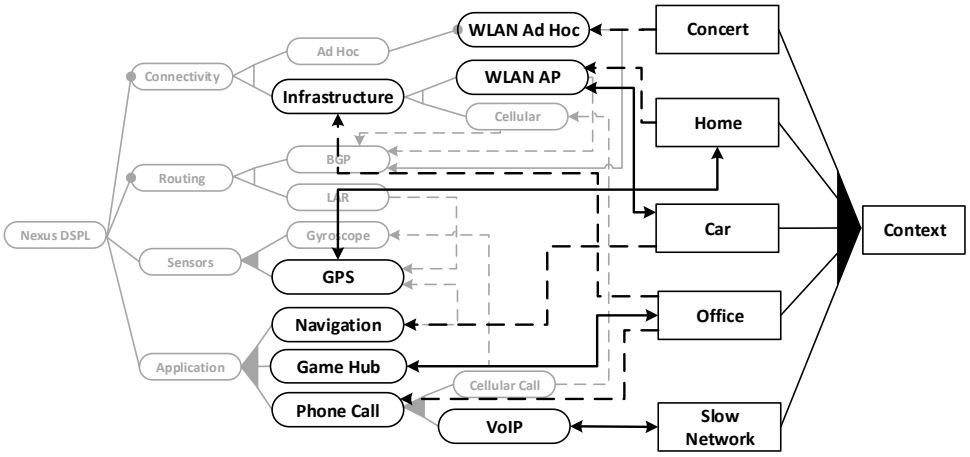


Figure 4.2: Mapping of Contexts to the Nexus DSPL

Figure 4.2 depicts the Nexus DSPL enriched with a context model. The left-hand side depicts the feature model of the Nexus DSPL (c.f. Figure 2.12). The right-hand side depicts the contexts for that DSPL, organized in an *or-group*. Therefore, at least one context has to be active and all context may be active at the same time iff the requirements do not lead to a contradiction. For example, the context `Slow Network` excludes the feature `VoIP` because the available bandwidth is too low to support communication via voice over IP. The context `Office` requires the feature `Phone Call` to be active, i.e., either `VoIP`, `Cellular Call` or both features have also to be active. Therefore, `Slow Network` and `Office` are combinable to describe a contextual situation where the user is in his office and has a slow Internet connection on his smartphone, i.e., {`Slow Network`, `Office`}. However, the combination of those contexts further restricts the configuration space of `Office` because a phone call via `VoIP` is excluded by `Slow Network`.

Further contexts are `Car`, which requires a `Navigation` system and excludes communication via `WLAN AP`, `Home`, which requires communication via `WLAN AP` and excludes the `GPS` sensor, and `Concert`, which requires a decentralized communication via `WLAN Ad Hoc`.

CONTEXTUAL CONSTRAINTS. For a seamless integration of contexts into a DSPL, I propose that a context model is combinable with a DSPL feature model, i.e., contexts variables $c \in \mathcal{C}$, contextual constraints $\tau \in \Phi_{\mathcal{C}}$, and context to feature constraints $\theta \in \Phi_{\mathcal{C} \cup \mathcal{F}}$, are part of a context-aware feature model formula *cfm* in addition to the feature variables $f \in \mathcal{F}$ and feature constraints $\varphi \in \Phi_{\mathcal{F}}$. In this regard, constraints among contexts are specified as a formula $\tau \in \Phi_{\mathcal{C}}$ by applying a restricted alphabet of a propositional formula (c.f. Definitions 4.1 and 4.2). Additionally, the logical connectives among features and contexts in the context-feature formula $\theta \in \Phi_{\mathcal{C} \cup \mathcal{F}}$ are limited to only express the following constraints

- $\theta = c \rightarrow f$, i.e., context c requires feature f and
- $\theta = c \rightarrow \neg f$, i.e., context c conflicts with feature f.

Such require and exclude constraints lead from the context model to the feature model. Further constraints are specifiable to restrict the combination of contexts $c \in \mathcal{C}$. For example, in the Nexus DSPL depicted in Figure 4.2 all contexts are organized in an or-group.

Definition 4.18 (*Context-Feature Formulae*)**.** A context-feature formula is a specific propositional formula according to Definition 4.2 and is restricted in its set of possible constraints as follows.

Let $c \in \mathcal{C}$ be a set of contextual variables and $f \in \mathcal{F}$ be a set of feature variables. The set $\varphi \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ of all well-formed propositional formulae over $\mathcal{C} \cup \mathcal{F}$ is inductively defined as follows

(1) $(\mathcal{C} \cup \mathcal{F}) \subseteq \Phi_{\mathcal{C}\cup\mathcal{F}}$, i.e., every atomic variable is also a well-formed formula,

(2) $\mathcal{W} \subseteq \Phi_{\mathcal{V}}$, i.e., every atomic constant is also a well-formed formula,

(3) if $\varphi \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ is a formula, then $(\varphi) \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ is also a formula, and

(4) if $\varphi \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ is a formula, then $\neg\varphi \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ is also a formula.

Constraints to restrict combinations amongst feature variables $\varphi \in \Phi_{\mathcal{F}}$ are specified via the connectives $\wedge, \vee, \veebar, \rightarrow$

(5) if $\varphi_1 \in \Phi_{\mathcal{F}}$ and $\varphi_2 \in \Phi_{\mathcal{F}}$ are formulae then $(\varphi_1 \star \varphi_2) \in \Phi_{\mathcal{F}}$ with $\star \in \{\wedge, \vee, \veebar, \rightarrow\}$ is also a formula to denote feature constraints.

Constraints to restrict combinations amongst context variables $\tau \in \Phi_{\mathcal{C}}$ are specified via the connectives $\vee, \rightarrow$

(6) if $\tau_1 = c \in \Phi_{\mathcal{C}}$ and $\tau_2 = c' \in \Phi_{\mathcal{C}}$ are formulae then $(\tau_1 \star \tau_2) \in \Phi_{\mathcal{C}}$ with $\star \in \{\vee, \rightarrow\}$ is also a formula to denote context constraints.

Constraints to specify require and exclude dependencies amongst context variables $\theta \in \Phi_{\mathcal{C}}$ feature variables $\theta \in \Phi_{\mathcal{F}}$ are specified via an implication $\rightarrow$

(7) if $\theta_1 = c \in \Phi_{\mathcal{C}}$ and $\theta_2 = f \in \Phi_{\mathcal{F}}$ are formulae then $(\theta_1 \rightarrow \theta_2) \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ is also a formula to denote constraints amongst contexts and features.

In the following, I denote the constraints solely between features as a single well-formed formula $\varphi \in \Phi_{\mathcal{F}}$, constraints between features and contexts as a single well-formed formula $\theta \in \Phi_{\mathcal{C}\cup\mathcal{F}}$, and constraints among contexts as a single well-formed formula $\tau \in \Phi_{\mathcal{C}}$.

Example 4.8 (*Contextual Formulas*).

As shown in Figure 4.2, *or-group* constraint is sufficient to restrict the combinations among contexts for the setting of the Nexus DSPL running example.

However, further constraints among contexts may be specifiable, e.g., a conflict such as `Office`$\rightarrow \neg$`Concert`.

Additionally, multiple constraints among feature and contexts are specified to describe the requirements a contexts imposes, e.g., `Home`$\rightarrow$`WLAN AP`.

Based on such a DSPL specification, it is possible to autonomously reason about reconfigurations that are triggered by contextual changes and execute a corresponding adaptation. For example, if the context `Home` is being entered, a reconfiguration is derivable which satisfies the new contextual situation.

THE CONTEXT-FEATURE MODEL. To integrate contexts and the respective contextual constraints into a feature model, it is required that the implication relation $c_r \rightarrow f_r \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ holds, where $c_r \in \mathcal{C}$ is the root context of a context-feature model *cfm* and $f_r \in \mathcal{F}$ is the root feature of a standard feature model *fm*.

A context-feature model *cfm* extends an existing feature model *fm* (c.f. Definition 4.7) with contextual variables $c \in \mathcal{C}$. Furthermore, contextual constraints are specified as well-formed formula $\theta \in \Phi_{\mathcal{C}\cup\mathcal{F}}$ over feature and context variables and as the formula $\tau \in \Phi_{\mathcal{C}}$ over context variables. The context variables are treated similarly to feature variables $\mathcal{F}$ which results in a context-feature model formula over of both sets $\mathcal{C}$ and $\mathcal{F}$. In this regard, the resulting feature model $cfm \in \mathcal{FM}_{(\mathcal{F}\cup\mathcal{C})}$ extended by context information is defined as follows.

Definition 4.19 (*Context-Feature Model*)**.** A context-feature model formula $cfm \in \mathcal{FM}_{\mathcal{C}\cup\mathcal{F}}$ corresponds to a standard feature model formula according to Definition 4.7 extended with context variables.

A *cfm* is specified via a single well-formed formula over context variables $\mathcal{C}$ *and* feature variables $\mathcal{F}$ by conjugating a feature model formula (*fm*) $\varphi \in \Phi_{\mathcal{F}}$, a context formula $\tau \in \Phi_{\mathcal{C}}$, and a context-feature formula $\theta \in \Phi_{\mathcal{C}\cup\mathcal{F}}$, i.e.,

$$cfm = \varphi \wedge \tau \wedge \theta.$$

The three-valued logic, c.f., Definition 4.11, applies to a context in the same manner as to a feature. Thus $\mathcal{C} \rightarrow \mathbb{B}_{\perp}$, a context may be `true`, i.e, *active*, `false`, i.e., *inactive*, or $\perp$, i.e., *unconfigured*. Consider the Nexus DSPL, if the user is at *Home*, the context `Home` is active, `Concert` is inactive, and `Slow Network` is unconfigured because it is irrelevant for the requirements of `Home`. Thus, `Slow Network` may become active at any time or stay inactive if the contextual situation `Home` is given.

The enrichment of the feature model specification by contextual requirements in a context-feature model *cfm* provides the means to comprehensively reason about changes in the context of a device in a consistent way as the variability specification and contextual requirements are solvable in a single step.

Contexts represent distinct states of the environment and context changes, therefore, they have an external cause. A single context is assumed to become active instantaneously after being recognized by the system interface after being entered (denoted by the event $\langle \oplus c \rangle$) and to become inactive (denoted by the event $\langle \ominus c \rangle$) when being left, with $c \in \mathcal{C}$. If the device emits such an event, the respective

requirements are derivable as a partial configuration of context variables. Based on the set of active contexts, a corresponding reconfiguration of features is triggerable in order to satisfy the contextual situation.

ENTERING AND LEAVING A CONTEXT. Entering or leaving a single context $c \in \mathcal{C}$ corresponds to a refinement of the currently active runtime configuration which may trigger a reconfiguration to satisfy the requirements of c. A context $c \in \mathcal{C}$ may only become active on a device if (i) the respective event is emitted after the context is being entered, (ii) the requirements imposed by c do not contradict *cfm*, and (iii) the activation of c corresponds to a valid refinement of the currently active configuration $\gamma \in \check{\Gamma}_{cfm}$. A detailed elaboration of contextual events and how they affect a reconfigurations is provided in Section 6.4 of this thesis.

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

- a feature model formula *fm* is enriched with contextual information to compose a context-feature model formula *cfm*,
- contexts represent distinct states of the environment and impose require and exclude constraints on features to satisfy that environment,
- contexts are organized in an *or-group* to ensure a combination of contexts,
- a context-feature model $cfm \in \mathcal{FM}_{\mathcal{C} \cup \mathcal{F}}$ is a well-formed formula over context variables $\mathcal{C}$ and feature variables $\mathcal{F}$,
- a context $c \in \mathcal{C}$ may be *active*, *inactive* or *unconfigured*, and
- the system emits an event if a single context changes, i.e., $\langle \oplus c \rangle$ if a context c becomes active and $\langle \ominus c \rangle$ if a context c becomes inactive.

Thus, the runtime configuration of a device depends on the set of currently active contexts. A reconfiguration is triggered when a context becomes active or inactive and the currently active configuration does not satisfy the new configuration interpretation of active contexts.

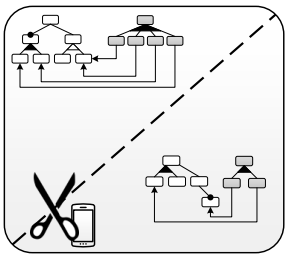
This concept of a context-aware DSPL provides the fundamental basis for my approaches to reduce the overall resource consumption of a DSPL-based adaptation process. The next part discusses my three optimization techniques of (i) a device-specific reduction of a feature model, (ii) an incomplete state space, and (iii) a prediction of runtime reconfigurations.

Part III

RESOURCE CONSTRAINT RUNTIME ADAPTATION

5

REDUCTION OF A FEATURE MODEL



The previous Part of this thesis introduced feature models as a fundamental basis for a DSPL-based adaptation approach. This chapter introduces the first of three techniques to satisfy the second goal G2 of this thesis, i.e., to reduce the resource consumption of a DSPL-based adaptation. To tackle this issue, I reduce the size of a feature model to a minimal set of contexts and features that are reconfigurable at runtime.

To improve the runtime characteristics of a DSPL reconfiguration, i.e., enhance responsiveness and reduce the computational effort for the derivation of a configuration, I propose a reduction technique that minimizes the context and feature variables in a context-feature model formula to a set of variables that are reconfigurable at runtime. A partial configuration that defines device specific characteristics serves as a criterion how the feature model formula is to be reduced. In this regard, features that are always active at runtime or features that are incompatible to a device are removed from the feature model formula. With such a reduction of the feature model to a set of reconfigurable features, the configuration space of the DSPL is reduced as well as the computational effort of computing a (re-)configuration at runtime. This improves the reconfiguration process at runtime in both responsiveness and resource consumption.

Figure 5.1 depicts an example how the Nexus DSPL may be reduced. In this example, the reduction criterion is an incompatibility of the device to WLAN-based features, i.e., `WLAN Ad Hoc` and `WLAN AP` are explicitly deselected from the DSPL by some developer. In this regard, the features `WLAN Ad Hoc` and `WLAN AP` may never become active at runtime. Based on the feature model specification of the Nexus DSPL, this has further implications, e.g., `Ad Hoc` may never become active and, because of the alternative-group constraint, `Infrastructure` has always to be active at runtime. Thus, the set of reconfigurable features is reduced. For example, the `Sensors` and `Application` features are still considered to be reconfigurable at runtime. Therefore, the 10 static non-reconfigurable features are removable from the feature model formula, which reduces the set of variables to 9 variables in the feature model formula that are reconfigurable at runtime.

Slicing is a technique that tackles the issue of reducing the size of a model by preserving the semantics of that model [LKR10]. Originally, slicing has been developed for the reduction of code statements without changing the behavior of the implementation [Wei81]. In this regard, slicing seeks to remove as many statements as possible w.r.t. a slicing criterion. The slicing criterion consists of

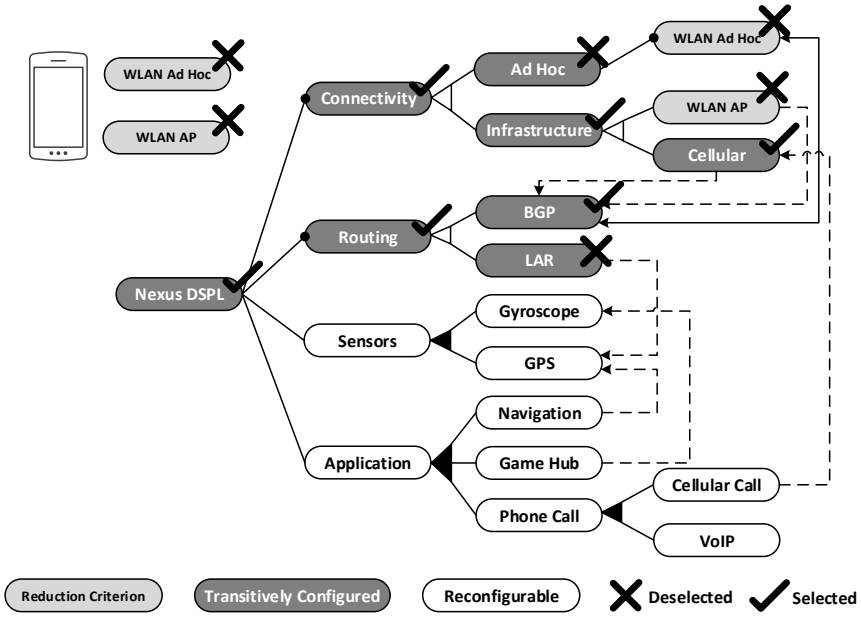


Figure 5.1: Reduction Possibility of the Nexus DSPL Feature Model

a subset of code statements of interest, e.g., a set of variables of interest and a program location. During the slicing process, every code statement is removed, which does *not affect* the code statements of the slicing criterion [Wei81]. The slice results in a subset of code statements of the original implementation.

In addition to code slicing, literature provides approaches to slice a feature model specification w.r.t. a set of features as slicing criterion. In this regard, a feature model is interpreted as a dependency graph. The amount of features is reduced to a subset of relevant features by preserving the semantic properties of the original specification [ACLF11, RSA11]. The overall idea behind the slicing of a feature model is similar to program slicing. However, for feature models the slicing is executed on the basis of a set of features that have to be excluded from the sliced feature model [ACLF11]. Although the intention of those approaches is similar to my approach, i.e., to reduce a feature model specification,

- they neglect the individual variability characteristics of a DSPL at runtime and
- they slice a feature model diagram, whereas I reduce the propositional formula representation of a feature model.

To tackle the issue of reducing a feature model formula w.r.t. features that are reconfigurable at runtime, I apply the three-valued logic established in the previous chapter. In this regard, features may be interpreted as selected, deselected, and reconfigurable. I use three-valued partial configurations as a reduction criterion for the original context-feature model to derive a device-specific reduced context-feature model. The resulting device-specific reduction consists of a subset of features that are reconfigurable at runtime. Thus, as a reduction criterion, selected and deselected features are removed from the reduced feature model.

This chapter introduces my concept to reduce a DSPL context-feature model specification according to the heterogeneous characteristics of an individual device as reduction criterion [SOS+12]. In this regard, the overall deployment of a DSPL specification on a set of heterogeneous devices is introduced. Afterwards, I discuss the reduction process of a context-feature model formula and a given partial configuration as reduction criterion. To ensure a correct reduction w.r.t. the original DSPL specification, I prove the correctness of my approach. Finally, I provide an evaluation based on a trade-off between costs and benefits of the reduction process.

5.1 Device Specific Reduction and Deployment

In SPL engineering, a device specific product configuration is derived and deployed on a set of heterogeneous devices, e.g., Nexus 4, 7, and 10. Such a software configuration may vary from device to device, depending on the hardware characteristics. There is no need to deploy a feature model on the device because every feature is configured statically and there is no variability at runtime. Thus, there is no need to derive a customized feature model specifying the runtime variability.

Similarly, a DSPL specification addresses the heterogeneity of a whole product line. The heterogeneity in the set of supported devices in the product line results in different adaptation behavior of each device. Since SPLs are intended to handle heterogeneity, my concept of a context-aware DSPL as introduced in Section 4.2 has to be adapted accordingly to support the individual runtime characteristics of each device in a product line. In this regard, the DSPL specification has to be *reduced* according to the capabilities and requirements of each individual device.

Figure 5.2 depicts the overall deployment process. A deployment server is equipped with the DSPL feature repository and the respective context-feature model. This server knows the capabilities and requirements of each target device. Based on these information the deployment server computes a reduction of the original feature model for each individual device.

Example 5.1 (*Device Specific Deployment*).

The off-the-shelf composition of software artifacts that are deployed on the mobile devices differs for each type of device in the product line. For example, *Research In Motion* offers a specific variant of *BlackBerry Storm* that does *not* provide any WLAN capabilities. Therefore, the respective drivers and applications that rely on aa WLAN chip are *not* deployed on such a device. In contrast to that, the *BlackBerry 9000* fully supports WLAN-based connections and, therefore, has additional features, such as applications (e.g. a WLAN scanner), menu entries (e.g. setup of a WLAN connection), etc. Therefore, the respective WLAN drivers, customized menus, and applications are deployed on every *BlackBerry 9000*. In this regard, a *BlackBerry 9000* is able to dynamically reconfigure between a `WLAN AP`-based connection and a `Cellular` connection at runtime.

The next section discusses in detail the concept of a feature model reduction and defines correctness properties for such a reduction. Further, an algorithm to de-

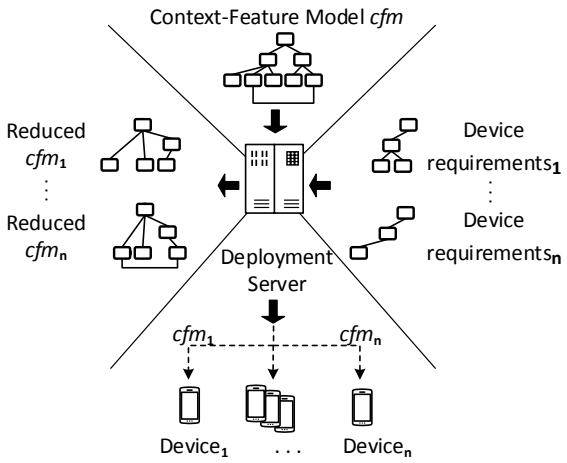


Figure 5.2: Deployment of Device Specific Feature Models

rive a reduced feature model is introduced that is executed before the reduced feature model is deployed on the respective target device.

5.2 FEATURE MODEL REDUCTION

The reduction of a feature model has to be *correct* in such a regard that only selected and deselected features are removed and the remaining variability constraints are preserved accordingly. Otherwise, a reduced feature model may lead to new runtime configurations that are not intended by the original specification of the DSPL.

According to Definition 4.19, a context-feature model *cfm* is a propositional formula over context variables $\mathcal{C}$ and feature variables $\mathcal{F}$. A configuration of a *cfm* corresponds to an interpretation of every variable by assigning a value from the domain of three-valued logics $\mathbb{B}_\perp$. Note that context and feature variables are treated equally in the following and I do not distinguish between them explicitly. Therefore, I denote a feature *or* context variable as $cf \in \mathcal{C} \cup \mathcal{F}$ in the following for short.

For a reduction of a feature model, a three-valued configuration interpretation of context/feature variables $cf \in \mathcal{C} \cup \mathcal{F}$ denotes the following.

- $\mathtt{t} = cf$ is configured to be selected,
- $\perp_R = cf$ is reconfigurable at runtime and
- $\mathtt{f} = cf$ is configured to be deselected.

Note that an explicit configuration of a context/feature as *reconfigurable* $\perp_R$ corresponds to the unconfigured notation $\perp$ in a three-valued logic, c.f., Notation 4.3.

Features are specified as reconfigurable at design time of the DSPL to express that a configuration decision for that feature still has to be made. A feature is

configured as reconfigurable if the developer intends a further configuration *refinement* at a later stage in the software life-cycle, e.g., a refinement by another developer. Additionally, if a feature is still specified as reconfigurable at the deployment of the DSPL, it is considered to be dynamically (de-)activatable at runtime. Thus, if a DSPL is deployed, a context/feature that *has* to be reconfigurable at runtime *has* to be specified as reconfigurable at design time.

Definition 5.1 (*Reconfigurable Contexts and Features*)**.** Given a partial configuration $\gamma \in \check{\Gamma}_{cfm}$ for a context-feature model. A context $c \in \mathcal{C}$ and feature $f \in \mathcal{F}$ may be dynamically (de-)activated if $\gamma(c) = \bot_R$ and $\gamma(f) = \bot_R$ holds at design time to denote a reconfigurability of c and f at runtime.

An explicit configuration of a feature as reconfigurable is only necessary for the extended satisfiability check to uncover transitive or direct contradictions between a reconfigurable feature configuration and a dependent selected or deselected feature.

Example 5.2 (*Reconfigurable Features*).

A partial configuration contains features that are to be configured at a later point, either at design time as (de-)selected or at runtime as (de-)activated. For example, the `Connectivity` features are specified as *reconfigurable* by an developer in order to dynamically reconfigure between `Ad Hoc` and `Infrastructure`-based communication at runtime. Therefore, such *reconfigurable features* are part of the DSPL that is deployed on the device, although they are neither selected nor deselected. A specification of `Infrastructure` as selected and `Ad Hoc` as reconfigurable contradicts the feature model constraints specified for the Nexus DSPL (c.f. Figure 2.12). In this case, either `Ad Hoc` has to be deselected or `Infrastructure` has also to be reconfigurable.

For a feature model reduction it is important that the reduction is executed *correctly*. In this regard, the set of valid configurations $\hat{\Gamma}_{cfm'}$ of a reduced context-feature model *cfm*′ has to be a subset of the original set of configurations $\hat{\Gamma}_{cfm'} \subset \hat{\Gamma}_{cfm}$ specified by the context-feature model *cfm*.

According to Definition 4.15 a substitution $cfm[\sigma_\gamma]$ of the variables $cf \in \mathcal{C} \cup \mathcal{F}$ in a context-feature model formula *cfm* with the interpretation of a configuration $\gamma \in \hat{\Gamma}_{\mathcal{C}\cup\mathcal{F}}$ replaces every variable in *cfm* with the respective interpreted value provided by γ, i.e., $cfm' = cfm[\sigma_\gamma]$. In this case *cfm*′ is derived from *cfm* by replacing all occurrences of context and feature variables $f \in \mathcal{C} \cup \mathcal{F}$ in *cfm* by

- `1` iff $\gamma(cf) = \texttt{t}$,
- cf iff $\gamma(cf) = \bot_R$, i.e., the variable is *not* replaced, and
- `0` iff $\gamma(cf) = \texttt{f}$.

As a reduction criterion, I use a partial configuration. With a substitution function σ, a reduced context-feature model *cfm*′ of the original context-feature model

cfm is derivable, based on such a reduction criterion. With this set of all partial configurations $\check{\Gamma}_{\mathtt{cfm}}$ as a possible reduction criterion, a reduced feature model is describable as follows.

A reduced feature model has to provide the option to derive a valid configuration. Thus, a reduced context-feature model *cfm'* is not reconfigurable if all variables have been substituted by constants according to the reduction criterion $\gamma \in \check{\Gamma}_{\mathtt{cfm}}$. In this regard, the set of valid configurations for *cfm'* becomes empty $\Gamma_{cfm'} = \emptyset$ because all variables have been substituted by constants (c.f. properties (1) and (2) of Definition 5.2).

Furthermore, a reduced feature model *cfm'* is *correct*, if the third property (3) of Definition 5.2 holds. This property states that the set of valid configurations $\hat{\Gamma}_{cfm'}$ of the reduced *cfm'* also satisfies the original *cfm* over a reduced set of variables $\mathcal{C}' \cup \mathcal{F}'$. The restriction of a configuration $\gamma\big|_{\mathcal{C}'\cup\mathcal{F}'}$ corresponds to a valid configuration $\gamma \in \hat{\Gamma}_{cfm}$, in which the set of interpreted context variables $\mathcal{C}$ and feature variables $\mathcal{F}$ of *cfm* is reduced to a subset $\mathcal{C}'$ and $\mathcal{F}'$ of contexts and features, respectively. For example, the configuration of the features marked as *reconfigurable* in Figure 5.1 corresponds to a valid configuration of the Nexus DSPL, e.g., such as depicted in Figure 2.13, in which the features marked as *selected* and *deselected* are excluded from the interpretation. In this regard, every configuration that satisfies the reduced *cfm'* also satisfies the original context feature model *cfm*.

Definition 5.2 (*Reduced Feature Model*)**.** Let $cfm \subseteq \mathcal{FM}_{\mathcal{C}\cup\mathcal{F}}$ be a context-feature model. Further, let $\gamma \in \check{\Gamma}_{cfm}$ be a partial configuration of *cfm* as reduction criterion. A context-feature model $cfm' \subseteq \mathcal{FM}_{\mathcal{C}'\cup\mathcal{F}'}$, with $\mathcal{C}' \subseteq \mathcal{C} \cup \mathcal{F}' \subseteq \mathcal{F}$, is a *reduction* of *cfm* w.r.t. the reduction criterion γ if the following properties hold

(1) there is at least one feature variable reconfigurable, i.e., $\exists f \in \mathcal{F} \,:\, \gamma(f) = \bot_{\mathrm{R}}$, or

(2) there is at least one context variable reconfigurable, i.e., $\exists c \in \mathcal{C} \,:\, \gamma(c) = \bot_{\mathrm{R}}$, and

(3) the set of complete configurations $\hat{\Gamma}_{cfm'}$ that satisfies *cfm'* over the reduced sets of context variables $\mathcal{C}'$ and feature variables $\mathcal{F}'$ also has to satisfy the original *cfm*, i.e., $\hat{\Gamma}_{cfm'} \subseteq \{\gamma\big|_{\mathcal{C}'\cup\mathcal{F}'}\}$, with $\gamma \in \hat{\Gamma}_{cfm}$.

To derive a DSPL that is specific for a device, the context-feature model specification has to be reduced accordingly. This process of reducing a feature model is discussed in the next section.

5.3 The Reduction Process

The goal of the reduction process is to derive a *correct* reduced feature model consisting only of reconfigurable features. The reduced feature model is correct if the set of valid configurations $\hat{\Gamma}_{cfm'}$ of a reduced context-feature model *cfm'* is a

subset of the original configurations $\hat{\Gamma}_{cfm}$ specified by the original context-feature model *cfm*.

The reduction of a feature model requires a partial configuration as a reduction criterion, which reflects the capabilities of the target device and the context-feature model specification of the respective DSPL. The reduction itself minimizes the number of feature and context variables in a feature model formula to a set of variables that are reconfigurable at runtime.

By applying a device-specific partial configuration at deployment, the reduced feature model corresponds to a pre-configured feature model, in which all selected and deselected context/features are removed. This is done via a substitution of the variables with Boolean constants {`0`, `1`} w.r.t. the interpretation in a partial configuration.

The reduction of a context-feature model *cfm* is illustrated in Figure 5.3. Starting with a *cfm* specified for the whole DSPL, a partial configuration $\gamma_d \in \check{\Gamma}_{cfm}$ specific for a device $d \in \mathcal{D}$ is derived as a reduction criterion before a reduced context-feature model is deployed on the device *d*.

Every reduction is derived w.r.t. the characteristics of a set of heterogeneous devices $\{d_1, d_2, \ldots, d_n\} \in \mathcal{D}$. The reduction process requires (i) a context-feature model *cfm* and (ii) a configuration $\gamma_d \in \check{\Gamma}_{cfm}$ as reduction criterion describing compatibility and variability for a specific device $d \in \mathcal{D}$ by using a three-valued interpretation. Finally, the reduced context-feature model *cfm'* is deployed on the respective target device $d \in \mathcal{D}$.

In this regard, contexts/features are either (i) bound to be selected (`t`), (ii) bound to be deselected (`f`), or (iii) are intended to be reconfigurable ($\perp_R$) at runtime in a partial configuration γ_d for a target device $d \in \mathcal{D}$. A feature has to be selected for a device if it has to be activated on that device at all time, e.g., the driver for the display on of a Nexus device. A feature has to be deselected if it is incompatible to the characteristics of a device, e.g., `NFC` is not supported by the hardware characteristics of a Nexus 7. Further, a feature is specified to be reconfigurable if it is intended to be dynamically (de-)activatable at runtime.

The same applies to the configuration of a context in a device specific configuration γ_d. A context is bound to be selected if it is assumed always to be active at runtime, e.g., the context `Car` is configured to be selected if the device is built *into* a car. A context has to be deselected if it is prohibited on a device or incompatible to a device, e.g., the context `Concert` may be considered to be prohibited on a business phone. A context is configured to be reconfigurable, if it may become active at runtime but does not have to be active at all time.

The derived reduced feature model corresponds to a minimal specification describing possible variability of a specific device to compute reconfigurations according to the dynamically changing context. A detailed reduction process to derive such a reduced feature model is given in Algorithm 1, which is discussed in the next section. The algorithm is based on the configuration semantics of a context-aware DSPL, introduced in Section 4.2.2 of this thesis.

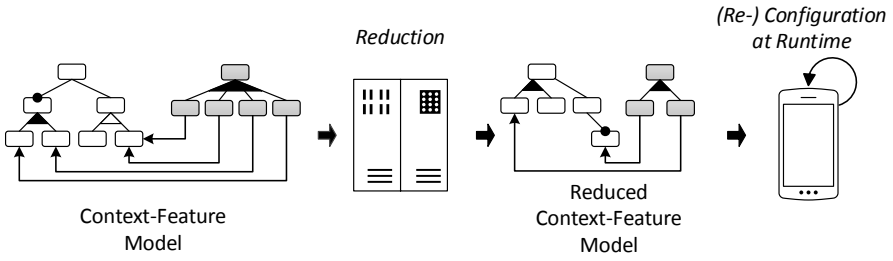


Figure 5.3: Reduction Process for a context-aware DSPL

5.3.1 *Implementation of a Feature Model Reduction*

With a given context-feature model *cfm* and a partial configuration $\gamma_d \in \check{\Gamma}_{cfm}$ for a device $d \in \mathcal{D}$ the Algorithm 1 computes a reduced feature model *cfm'* as follows.

To derive a reduction w.r.t. a partial configuration γ_d, the algorithm checks every feature $f \in \mathcal{F}$ *and* every context $c \in \mathcal{C}$ in the set of context-feature variables $\mathcal{C} \cup \mathcal{F}$ (line 4). Since contextual features and software features are treated equally by the algorithm, I refer to both as a feature.

The feature variables that are configured to be selected or deselected for the device specific configuration γ_d are directly substitutable with the respective `0` and `1` constants in the propositional formula representation *cfm*.

To identify reconfigurable features in the reduced feature model *cfm'*, every feature has to be evaluated if it is configurable as active (line 7–11) or as inactive (line 12–17) w.r.t. a device specific partial configuration $\gamma_d \in \check{\Gamma}_{cfm}$.

Whether a feature is reconfigurable or not w.r.t. *cfm* and γ_d is evaluated by the function `bound(f, i, γd, cfm)`. This function evaluates a feature variable $cf \in \mathcal{C} \cup \mathcal{F}$ according to its Boolean interpretation $i \in \mathbb{B}$ w.r.t. the given configuration γ_d and formula *cfm*. In my implementation, I use the constraint solver *SAT4J*[1] [BP10] to evaluate whether a feature has to be selected, deselected or is reconfigurable at runtime. In this evaluation, a feature variable *cf* is interpreted with the value *i*. To test if *cf* is reconfigurable *cf* is interpreted with `t` and `f` (line 7 and 12). If any of these interpretations does *not* satisfy *cfm*, the variable may only be substituted by the value assigned to *i*. Correspondingly, the assigned value of *i* is the only interpretation, which does satisfy *cfm* and, therefore, *cf* is *not* reconfigurable at runtime. In this regard, `bound()` is executed as follows.

$$\text{bound}(cf, i, \gamma_d, cfm) \begin{cases} \textit{return } \texttt{true} \textit{ if } \nexists \gamma \in \hat{\Gamma}_{cfm} \text{ with } \gamma(cf) = \texttt{f} \text{ and } \gamma \sqsubseteq \gamma_d \\ \textit{return } \texttt{true} \textit{ if } \nexists \gamma \in \hat{\Gamma}_{cfm} \text{ with } \gamma(cf) = \texttt{t} \text{ and } \gamma \sqsubseteq \gamma_d \\ \textit{return } \texttt{false} \textit{ otherwise} \end{cases}$$

[1] http://www.sat4j.org/

Thus, iff *cf* has to be implicitly bound to one specific value interpretation *i* the function `bound` is evaluated to `true`. Otherwise *cf* remains unconfigured, i.e., reconfigurable at runtime.

In addition to an explicit configuration of a feature to be selected or deselected for a specific device configuration γ_d, the developer may also configure a feature *explicitly* as reconfigurable, i.e., $\perp_R$, for that device. To cover such a configuration choice, I apply the three-valued logics from Definition 4.11 in the reduction process. Such an explicit configuration of a feature as reconfigurable may lead to new contradictions that are not identifiable by standard solving algorithms. Therefore, I added an additional condition if a feature that is neither explicitly configured to be selected or deselected, has to be transitively configured to be selected or deselected. These conditions (line 8–10 and 13–15) check whether this respective feature *cf* is specified as reconfigurable in the device configuration γ_d or not. If a feature has to be selected or deselected although it is intended to be reconfigurable, a contradiction exception is thrown.

Finally, the variable *cf* is substituted in *cfm'* by either by `1` or `0` (lines 11 and 16) if `bound()` is evaluated to `true` and no contradiction exception is thrown. In line 20 the reduced context-feature model *cfm'* is returned.

Algorithm 1 Derivation of a Reduced Feature Model *cfm'*

1: **Input:** $cfm \in \mathcal{FM}_{\mathcal{C}\cup\mathcal{F}}$;
   $\mathcal{C}$; $\mathcal{F}$; $\gamma_d \in \check{\Gamma}_{cfm}$;
2: **Output:** γ_d specific reduced feature model *cfm'*
3: **Init:** $cfm' := cfm$
4: **for all** $cf \in \mathcal{C} \cup \mathcal{F}$ **do**
5: **if** $(\gamma_d(cf) \in \{\mathtt{t}, \mathtt{f}\})$ **then**
6: $cfm' := cfm'[\sigma_{cf=\gamma_d(cf)}]$
7: **else if** (**bound**(cf, `t`, γ_d, cfm') **then**
8: **if** $(\gamma_d(cf) = \perp_R)$ **then**
9: Contradiction Exception
10: **end if**
11: $cfm' := cfm'[\sigma_{cf=\mathtt{1}}]$
12: **else if** (**bound**(cf, `f`, γ_d, cfm')) **then**
13: **if** $(\gamma_d(cf) = \perp_R)$ **then**
14: Contradiction Exception
15: **end if**
16: $cfm' := cfm'[\sigma_{cf=\mathtt{0}}]$
17: **end if**
18: **end for**
19: **Return** *cfm'*

Example 5.3 (*Reducing the Nexus DSPL*).

A device specific reduction of the Nexus DSPL context-feature model is depicted in Figure 5.4. The target device has no integrated WLAN chipset and therefore, the features `WLAN Ad Hoc` and `WLAN AP` are configured to be deselected

before the DSPL is deployed on the device. Thus, the partial configuration dictates a removal of `WLAN Ad Hoc` and `WLAN AP` as reduction criterion.

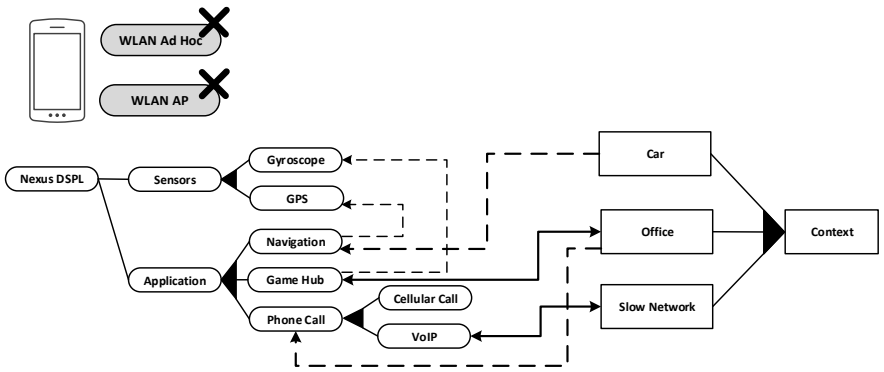


Figure 5.4: Reduced Feature Model for the Nexus DSPL in Figure 4.2

For instance, the feature `Connectivity` is bound to be selected for the target device, i.e., its variable is substituted by `1` in the propositional formula representation of the Nexus DSPL. Therefore, either the feature `Ad Hoc` or the feature `Infrastructure` has also to be bound to be selected for a valid configuration. Since `WLAN Ad Hoc` is deselected as a reduction criterion, `Ad Hoc` is substituted by `0` and `Infrastructure` by `1`. Thus, the respective alternative-group clause "$(\neg$`Connectivity` $\vee ($`Ad Hoc` $\wedge$ `Infrastructure`$) \vee ($`Infrastructure` $\wedge$ $\neg$`Ad Hoc`$))$" in the original *cfm* becomes "$(\neg$`1` $\vee$ $($`0` $\wedge$ `1`$)$ $\vee$ $($`1` $\wedge$ $\neg$`0`$))$" in the reduced *cfm*'. The substituted clause is always evaluated to `1`, which eases the solving process of *cfm*' in comparison with *cfm*. In this regard, the variables are *removed* and substituted by constants.

In comparison with the original specification depicted in Figure 4.2 it becomes apparent that features and contexts are removed from the feature model, in addition to the incompatible features `WLAN Ad Hoc` and `WLAN AP`. For example, the contextual situations `Home` and `Concert` are not supported by the target device since they directly depend on a WLAN connection. The only alternative to a WLAN-based connection is a `Cellular` communication. If one considers the additional constraint that at least one connection has to be selected, the feature `Cellular` is being selected and may not be reconfigured at runtime. Thus, while the `Connectivity` branch is removed from the context-feature model because all features are either bound to be selected or deselected before the software is deployed on the device.

This further implies that an explicit configuration of the feature `Cellular` to be reconfigurable leads to a contradiction in the transitive closure configuration if `Cellular` is selected. Similar to the transitive configuration of `Cellular`, the feature `Routing` is deselected and removed from the model. Therefore, the, still reconfigurable, child constraints are inherited to the root feature `Nexus DSPL`.

The next section provides a proof that my approach derives a *correct* reduced feature model *cfm'* w.r.t. the original context-feature model *cfm*.

5.3.2 *Proof of Correctness*

The reduced and, thus, smaller context-feature model *cfm'*contains only a subset $\mathcal{C}' \cup \mathcal{F}'$ of reconfigurable features and contexts of the original *cfm*. All selected contexts/features are permanently active at runtime and all deselected features are never deployed on the target device. Thus, if a reconfigurable context/feature becomes active and part of the configuration in a further refinement at runtime, the required features are already part of the runtime configuration.

Theorem 5.1. Let *cfm* be a context-feature model formula with $cfm \in \mathcal{FM}_{\mathcal{C}\cup\mathcal{F}}$. Any valid configuration $\gamma' \in \Gamma_{cfm'}$ of a reduced feature model $cfm' \in \mathcal{FM}_{\mathcal{C}'\cup\mathcal{F}'}$ w.r.t. to a given partial configuration $\gamma \in \check{\Gamma}_{cfm}$ is extendable to a configuration $\gamma^\circ \in \hat{\Gamma}_{cfm}$ of the original context-feature model *cfm* such that

$$\begin{aligned} &\forall\, cf' \in \mathcal{C}' \cup \mathcal{F}' :\ \gamma^\circ(cf') = \gamma'(cf') \\ &\forall\, cf \in \{\mathcal{C} \cup \mathcal{F}\} \backslash \{\mathcal{C}' \cup \mathcal{F}'\} :\ \gamma^\circ(cf) = \gamma(cf) \end{aligned} \tag{5.1}$$

Given that any configuration $\gamma^\circ \in \hat{\Gamma}_{cfm}$ with

$$\forall\, cf \in \{\mathcal{C} \cup \mathcal{F}\} \backslash \{\mathcal{C}' \cup \mathcal{F}'\} \ :\ \gamma^\circ(cf) = \gamma(cf) \tag{5.2}$$

is reducible to a $\gamma' \in \Gamma_{cfm'}$ then the following holds for a reduced *cfm'*

$$\forall\, cf' \in \mathcal{C}' \cup \mathcal{F}' \ :\ \gamma'(cf') = \gamma^\circ(cf') \tag{5.3}$$

with $\mathcal{C}'$ and $\mathcal{F}'$ as the remaining context/feature variables that are *not* substituted with a constant during the reduction of *cfm*.

For the proof of this theorem note the following property of a substitution σ. After the variables in *cfm* is substituted by the constants `0` and `1` w.r.t. a complete configuration $\gamma^\circ \in \Gamma_{\mathcal{C}\cup\mathcal{F}}$ (c.f. Definition 4.9) the solving of the substituted formula results either in a `0` if $\gamma^\circ \not\models cfm$ holds or in a `1` if $\gamma^\circ \models cfm$ holds.

Proof. That Theorem 5.1 holds is shown by contradiction as follows:

1. Obviously, there exists at most one $\gamma^\circ \in \hat{\Gamma}_{cfm}$ according to the extension of a γ' as assumed in Theorem 5.1 equation 5.2.
2. Let us assume that $\gamma^\circ \notin \hat{\Gamma}_{cfm}$

   $\Rightarrow\ cfm[\sigma_{\gamma^\circ}] \ =\ \texttt{0}$
   $\Rightarrow\ cfm[\sigma_\gamma][\sigma_{\gamma'}] \ =\ \texttt{0},$ with $cfm[\sigma_\gamma][\sigma_{\gamma'}] = cfm[\sigma_{\gamma^\circ}]$
   $\Rightarrow\ cfm''[\sigma_{\gamma'}] \ =\ \texttt{0}$ with $cfm'' = cfm[\sigma_\gamma]$
   $\Rightarrow\ cfm'[\sigma_{\gamma'}] \ =\ cfm''[\sigma_{\gamma'}] \ =\ \texttt{0}$ with $cfm' = cfm[\sigma_\gamma]$
   $\Rightarrow$ contradiction to $\gamma' \in \hat{\Gamma}_{cfm'}$ as required by Theorem 5.1 equation 5.3 $\lightning$

Analogously it can be shown that the first property specified in Equation 5.1 of Theorem 5.1 also as well. □

Hence, the reduction approach always results in a context-feature model that has less features, contexts, and constraints. Additionally, the remaining subset of valid runtime configurations of a reduced feature model *cfm'* is still consistent with the original specification *cfm*. In this regard, Algorithm 1 results in a reduced feature model *cfm'* such that each valid configuration of the original *cfm* that extends the partial configuration γ_d corresponds to a consistent configuration of the stepwise reduced feature model *cfm'*.

The next section provides insights about the effects of such a reduction and the extent of reduction in the resource consumption for a reconfiguration.

5.4 Evaluation

This section provides an evaluation for the reduction of a context-feature model. The evaluation illustrates the benefit of using a device specific reduction of the original feature model specification to execute reconfigurations at runtime instead of the original feature model. Therefore, the evaluation focuses on

1. a comparison of the time to compute a configuration,
2. resources (computational and memory) spent during such a computation, and
3. break-even between reduction-process and reconfigurations at runtime.

5.4.1 *Evaluation Setup*

As environment for the evaluation I use a server unit with 3.4GHz and 16GB of RAM. The test model is a feature model *fm* containing 72 features from the SPLOT[2] repository. Note that there are no representative context-feature models available besides the Nexus DSPL running example. Since my definition of contexts is directly integratable into a feature model as additional features and constraints, traditional feature models are usable as a valid substitute to simulate the reduction of context-feature models.

Device specific valid partial configurations $\gamma_d \in \check{\Gamma}_{fm}$ for a set of devices $d \in \mathcal{D}$ are randomly created by configuring features to be selected (20%) and deselected (5%). Since additional reconfigurable features may lead to contradictions during the reduction, every generated configuration $\gamma_d \in \check{\Gamma}_{fm}$ is considered to be satisfiable, reconfigurable features do not affect the outcome of this evaluation.

The reduction of a feature model *fm* according to $\gamma_d \in \check{\Gamma}_{fm}$ for a set of devices $d \in \mathcal{D}$ using the Algorithm 1 resulted in reduced feature models *fm'* with 32 features in average (44% less features/constraints). One evaluation run consisted of 300 (re-)configuration requests on both feature models *fm'* and *fm* using additional partial configurations $\gamma_c \in \check{\Gamma}_{fm'}$ with a set of contexts $c \in \mathcal{C}$ to simulate a continuous changing context. Each (re-)configuration represents a change in the contextual situation of the device and potentially triggers an adaptation on

[2] http://www.splot-research.org/

the device at runtime. Therefore, a complete configuration is computed, which is usable to execute an adaptation of the device.

In each evaluation run the *time* to compute a complete configuration and the executed *operational steps*, e.g., method calls, are evaluated. The operational steps are used as an abstraction for the consumed computational resources. To validate the results and to identify deviations, 500 evaluation runs are executed.

5.4.2 *Results*

The results for comparing the execution time for the computation of a complete configuration that satisfies the contextual requirements in comparison for *fm* and and a reduced *fm'* are depicted in Figure 5.5. During the computation of 300 reconfigurations triggered by the contextual changes in a partial context-configuration γ_c a single derivation of a complete configuration is approximately 60% faster if a reduced feature model *fm'* is used instead of the original feature model *fm*. In this regard, the computation of a suitable configuration based on *fm'* takes about 1.4 milliseconds, whereas the computation based on *fm* takes about 3.5 milliseconds. The variances result from the different, randomly generated contextual requirements $c \in \mathcal{C}$ since they differ in their size between 3 to 12 arbitrarily configured features in the respective partial context-configuration $\gamma_c \in \check{\Gamma}_{fm}$.

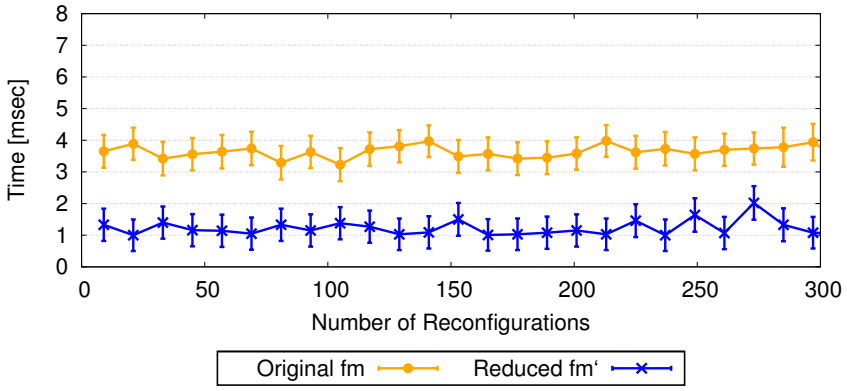


Figure 5.5: Comparison of Average Computational Time

Figure 5.6 depicts the amount of executed operations over 300 reconfiguration requests for the derivation of a satisfiable runtime configuration for *fm* and *fm'*. Again, the computation based on a reduced feature model *fm'* outperforms the computation based on the original feature model *fm*. During the processing of 300 reconfigurations 66% less operations are executed if *fm'* is used. This implies that less computational power is utilized and, therefore, the solving of contextual requirements based on a reduced feature model demonstrates to be more energy efficient. In contrast to the runtime comparison, the variance in the amount of executed operations is about four times higher if the original feature model *fm* is used. In this regard, in the worst-case scenario a *fm* based reconfiguration executes 38% more operations that in the worst-case of a *fm'* based reconfiguration.

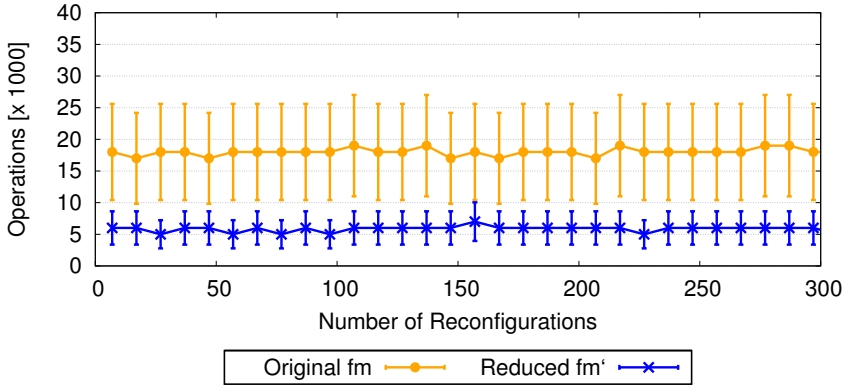


Figure 5.6: Comparison of Average Number of Executed Operations

The memory consumption over 300 reconfigurations that are either based on the original feature model *fm* or the reduced feature model *fm'* is depicted in Figure 5.7. The computation of a configuration consumes in average 23% less memory if *fm* is used instead of *fm'*. The worst-case consumption for both feature models is 200MB and 150MB, respectively for *fm* and *fm'*. The cause of saw-toothed periodic trend of both plots is to be found in the Java garbage collector as well as the used solver implementation SAT4J [BP10]. Data that is collected during the continuous reconfigurations with SAT4J are discarded whenever a certain threshold in the utilized memory is exceeded. This threshold depends on the complexity of the feature model and is regulated by SAT4J internally.

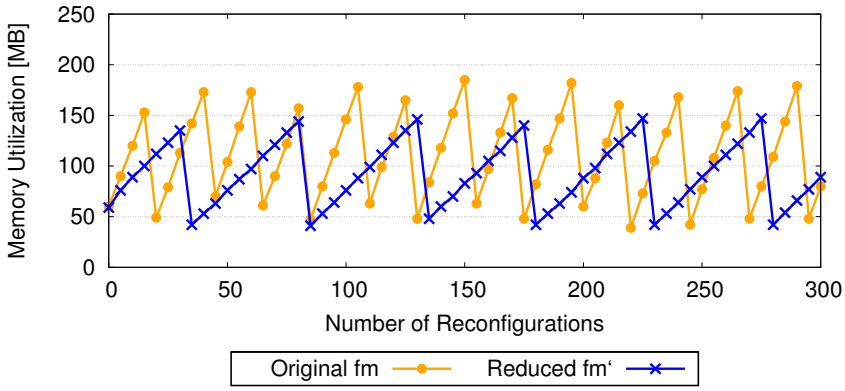


Figure 5.7: Comparison Memory Consumption during Reconfigurations

The process of reducing a feature model itself is costly. Algorithm 1 executed $2,264 \times 10^3$ operations to reduce *fm* according to a given partial device configuration γ_d. The computation of a valid runtime configuration required 17×10^3 and 8×10^3 operations in average, based on *fm* and *fm'* respectively. This implies that one has to execute approximate 180 reconfigurations at runtime using *fm'* to regain the resources spent for the reduction of *fm*. Figure 5.8 depicts the continuous progress of saving computational resources until a break-even point is achieved with the resources spent to reduce a feature model. It takes about 180 reconfigu-

rations based on *fm'* until the resources spent to compute the reduced *fm'* are met. These results show that a reduction of feature models should be computed on a device that has the sufficient resources, e.g., a deployment server. Nevertheless, in this particular evaluation setup the computation of a configuration is up to 60% faster and uses up to 34% less computational resources if a reduced feature model is used instead of the originally specified feature model.

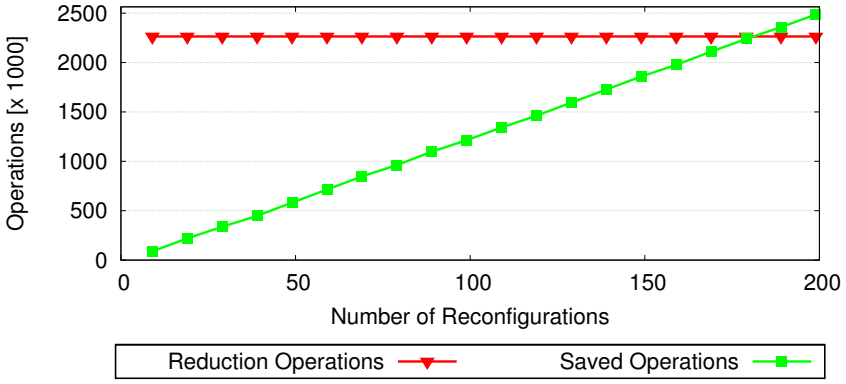


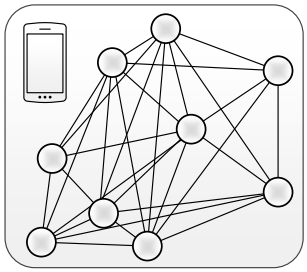
Figure 5.8: Break-Even Point Pre-Computation and On-Demand Reconfigurations

Summarizing, my reduction algorithm derives the minimal set of static features w.r.t. a device configuration. In this regard, I maximize the amount of features and contexts that are reconfigurable at runtime. Therefore, incompatible features and contexts as well as features and context that have to be active at all time are removed during the reduction process. Furthermore, additional conflicting requirements are discovered if a feature or context is explicitly configured to be reconfigurable at runtime, which may impose a contradiction to the remaining interpretations of features and contexts to be selected or deselected. The reduction process results always in a correct reduction of the original feature model, i.e., every configuration that is derived based on the reduced feature model is also derivable from the original feature model specification.

Although the reduction process lowers the computational efforts for deriving a configuration, the process of a DSPL reconfiguration at runtime still remains open. The next chapter introduces a concept to specify such reconfiguration behavior of a DSPL-based on a state-transition system and introduces concepts to improve this reconfiguration process at runtime.

6

STATE SPACE REDUCTION



A particular runtime configuration of a DSPL is obtained by binding all variability, i.e., by interpreting each feature as active or inactive according to the requirements imposed by a contextual situation. Thereby, the constraints imposed by a feature model restrict the *configuration space* of a DSPL to a subset of *valid* configurations. For instance, the Nexus DSPL comprises 19 features and, therefore, potentially allows 2^{19} feature combinations. However, the constraints imposed by the context-feature model depicted in Figure 4.2 restrict the configuration space to only 66 valid runtime configurations.

In the previous Chapter 5, I introduced an approach to reduce a context-feature model specification to a set of context and feature variables that are reconfigurable at runtime. Therefore, the approach reduces the *computational effort* for the *derivation* of a runtime configuration. However, the state space of valid configurations for a context-feature model may still be considerably large and contain configurations that are never required or are equivalent for the contextual situations emerging at runtime. To further minimize the resource consumption of a DSPL-based adaptation process, I investigate reduction techniques to minimize the computational effort as well as the memory utilization of a reconfiguration at *runtime* in this chapter.

According to the second goal G2 on page 8 of this thesis, i.e., the minimization of the resource consumption of an adaptation of a mobile device at runtime, reconfigurations have to be computed and performed in a way that (i) causes no influences on unaffected functionality of the device and, at the same time, (ii) respects the inherent resource limitations of the mobile device. In this regard, a successful application of DSPLs in the domain of mobile devices is faced with the following challenges.

- Performing seamless adaptations to consecutively satisfy complex (re-)configuration requirements of interfering and ever-changing contexts [OGC12].
- Coping with resource constraints that restrict computational runtime capabilities for complex reconfiguration planning tasks [HPS12].

To tackle these issues, recent research in the domain of DSPLs proposes model-based approaches for defining pre-planned reconfiguration scenarios [WDSB09, Hel12, DPS12]. The reconfiguration capabilities of a device are specified via a transition system as a part of the required adaptation knowledge for the DSPL.

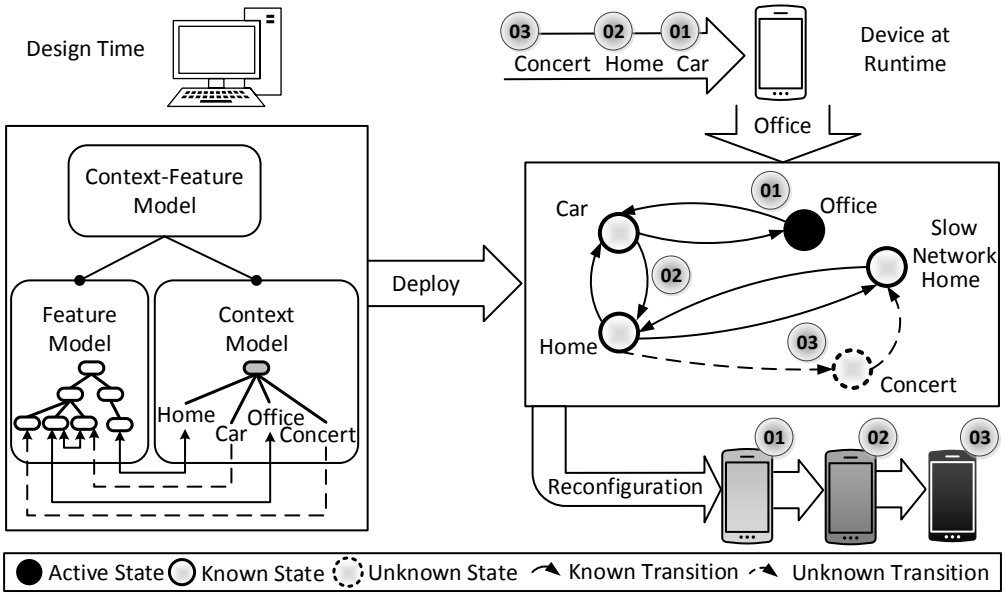


Figure 6.1: Overview of the reduction of a DSPL configuration state space. The left-hand-side depicts a context-feature model specification at design-time, which is used to deploy a context-specific *incomplete* configuration state space on the device. This state space is used to reconfigure the device at runtime based on changes in the contextual situation, as depicted on the right-hand side. Since the state space is incomplete, states may be unknown at runtime and have to be derived on-demand if the respective contextual situation emerges.

In such a transition system, a configuration is represented by a state and each potential reconfiguration is represented by a transition.

However, recent approaches for DSPL-based adaptive systems do not address challenges imposed by mobile devices, such as a limited amount of battery and restricted computational capabilities. In Chapter 4, I tackled the first goal G1 of this thesis, i.e., the provision of an autonomous model-based adaptation process by introducing the concept of a context-aware feature model. However, to execute such a context-aware adaptation of mobile devices based on a context-feature model, one has to consider that it is neither eligible 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, as related approaches do [FFF+13, Hel12]. To overcome these deficiencies, I propose a model-based DSPL framework for adaptive, resource-constrained devices incorporating

- context-aware reconfiguration planning based on a context-feature model and
- techniques for a reduction of configuration state spaces based on a transition system specification.

The right-hand side of Figure 6.1 depicts an example scenario for potential reconfiguration triggered by contextual changes based on a pre-computed transition

system. Therein, every state represents a complete configuration, which is suitable for a contextual situation. Every transition constitutes a potential reconfiguration. Based on the constraints imposed by the context-feature model a context-specific, a tailored transition system is derivable.

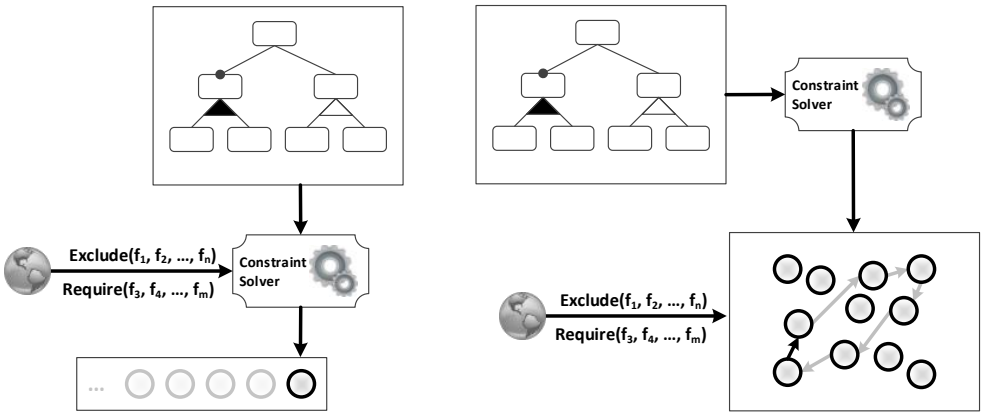
A transition system specifies the possible reconfiguration behavior of a device, i.e., which reconfigurations are executable w.r.t. to the continuously changing contextual situation of a device. Such a specification allows for tailoring an *incomplete* configuration state space on the basis of a context-aware feature model and a reduction criterion constituting a trade-off between comprehensively pre-computed pre-configurations and on-demand derivation of configurations at runtime. In this regard, a complete state space is reduced to a set of representative configuration states, to cover emerging contextual situations. Thereby, the amount of states is reduced from 66 potential states of the Nexus DSPL to 4 states, which cover the contextual situations {`Office`}, {`Car`}, {`Home`}, and {`Home`, `Slow Network`}, as depicted in Figure 6.1. Thus, instead of utilizing every valid configuration of a DSPL, this transition system is restricted to a pre-planned choice of configuration states and reconfiguration possibilities at runtime. Therefore, the state space is incomplete and configurations for contextual situations, which were not considered during the pre-planning, have to be derived on-demand. For instance, a configuration state for the context {`Concert`} is not part of the deployed transition system and has to be derived and integrated into the transition system on-demand if the context {`Concert`} is being entered (c.f. transition ⑬) for the first time.

The transition system depicted in Figure 6.1 further illustrates possible restrictions in the reconfiguration behavior of a device. For example, there is no transition between {`Office`} and {`Home`}. Instead, a sequence of transitions has to be executed, i.e., from {`Office`} the transitions ⑪ and ⑫ have to be executed to reach {`Home`}. Such a path of transitions may be required to handle technical issues such as a protocol hand-over between a seamless reconfiguration from a cellular phone call to a VoIP phone call.

Note that some ideas and figures of this chapter have appeared in my previous work on context-aware DSPLs [SLR13]. This chapter is organized as follows. At first, the concept of a configuration state space is introduced. Then, the state space is extended to a transition system by specifying reconfiguration transitions between configuration states. Afterwards, the state space is reduced to an incomplete state space via a context-coverage criterion. To further reduce the size of a state space and gain flexibility at runtime, I introduce a technique to abstract a configuration state to a partial state. To derive such a state abstraction, three techniques are discussed. Since reconfigurations are now to be executed in an incomplete state space with states that are abstracted to partial states, a detailed elaboration of the reconfiguration semantics for such a model is given. Further, correctness properties of the established reconfiguration framework are proven to ensure that (i) *no invalid* configuration states may become active and (ii) that *every valid* configuration state may *potentially* become active. Finally, insights into the implementation are given, before the established approach is evaluated to discuss trade-offs between costs of a pre-computation and possible benefits at runtime.

6.1 Complete State Space

A reconfiguration of a DSPL is triggered whenever the contextual requirements imposed on the device change in such a manner that the current device configuration does not satisfy these requirements. In such a case, the device has to compute a suitable configuration that satisfies the contextual situations emerging at runtime [FFF+13]. To derive a suitable configuration that satisfies the feature model constraints as well as the imposed contextual requirements, constraint solvers such as introduced in Section 4.1.3 may be used. However, deriving a suitable configuration with a constraint solver implies a computational effort with *every* reconfiguration, as depicted in Figure 6.2(a). Configurations are computed on-demand and may further be computed repeatedly, e.g., if the user changes from the contextual situation {`Home`} to {`Office`} and back to {`Home`}.



(a) On-Demand Configuration (b) Pre-Planned State Space

Figure 6.2: On-Demand Configuration and Pre-Planned State Space

To avoid such computational efforts at runtime, every valid feature model configuration may be computed in a pre-planning step [Hel12]. A constraint solver is capable to compute every valid interpretation of the feature variables that satisfy the constraints in the feature model. This resulting *configuration state space* of valid configurations is deployable on the device.

The configuration state space $\mathcal{S}$ of a device consists of a set of *configuration states* and an *error state*. Configuration states $s \in \mathcal{S}$ correspond to valid configurations $\gamma \in \hat{\Gamma}_{fm}$ of a feature model *fm*. In this regard, both notations $s \in \mathcal{S}$ and $\gamma \in \hat{\Gamma}_{fm}$ are used as synonyms in the remainder of this thesis. In addition to that, the state space $\mathcal{S}$ contains an error state $s_\epsilon \notin \hat{\Gamma}_{fm}$. A configuration state space is complete if the state space contains a state for each valid configuration $\gamma \in \hat{\Gamma}_{fm}$ of a feature model.

Instead of executing a constraint solver to reconfigure a device, the state space, which resides on the device, is searched for a suitable configuration state as depicted in Figure 6.2(b). Although such a configuration state space may consist of numerous configurations, the computational efforts to derive a configuration is

shifted from runtime to deployment time. Thereby, memory consumption w.r.t. the number of states and additional efforts for searching a suitable target state are traded for the computational efforts imposed by a constraint solver.

Summarizing, a *configuration state space* $\mathcal{S}$

- consists of a set of *configuration states* $s \in \mathcal{S}$, with s reflecting a valid configuration $\gamma \in \hat{\Gamma}_{fm}$,
- is *complete* iff $\forall \gamma \in \hat{\Gamma}_{fm} \ : \ \exists s \in \mathcal{S}$ with $\gamma = s$ and is *incomplete* otherwise, and
- both symbols of a state s and the respective configuration γ are used as synonyms.

Example 6.1 (*On-Demand Reconfiguration vs. Pre-Computed State Space*).

The feature model for Nexus DSPL depicted in Figure 2.12 consists of 19 features. The state space would contain 2^{19} configurations, i.e., 524,288 configurations, without the constraints imposed by the feature model. However, the constraints of the Nexus DSPL feature model restrict the amount of valid configuration to 66. Figure 6.3 depicts one of those states containing an extract of a runtime configuration for the Nexus DSPL.

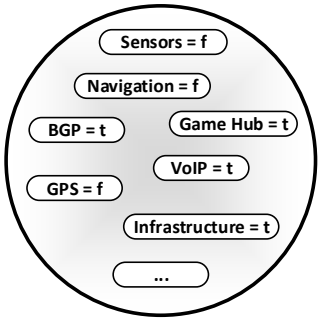


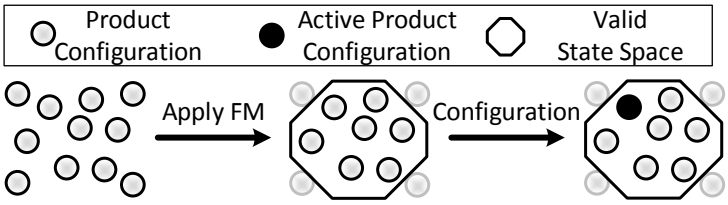
Figure 6.3: Example Configuration State

For a DSPL-based adaptation either that state space has to be kept in memory or every configuration has to be derived on-demand whenever the requirements imposed on the device change. A reconfiguration within the state space implies a search within the 2^{19} valid states for a state that satisfies the requirements imposed by the currently active contextual situation of the device. In contrast to a search for a suitable target configuration, a constraint solver checks every clause in the propositional formula, e.g., the Nexus DSPL consists of 49 clauses, *several* times to derive a suitable configuration.

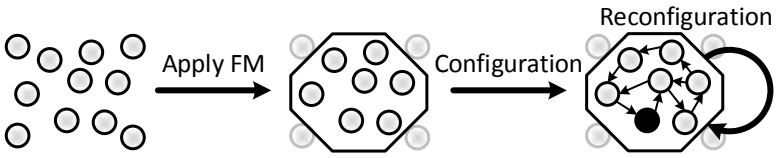
The next section introduces the concept of a DSPL transition system by enriching a state space with reconfiguration transitions. Afterwards, this concept is extended to handle autonomous, context-aware reconfigurations.

6.1.1 *Extension of a State Space to a Transition System*

As depicted in Figure 6.4(a), a product configuration of an SPL is obtained by binding all variability, i.e., by either selecting, or deselecting each provided feature according to customer-specific product requirements. The constraints imposed by a feature model restrict the number of valid configurations of the SPL to a subset of *valid* configuration states, thereby forming a *state space* of configurations.



(a) SPL Configuration State Space



(b) DSPL (Re-)Configuration

Figure 6.4: (Re-)Configuration in an SPL and DSPL

Figure 6.4(b) depicts the extension of an SPL to become a DSPL. A DSPL allows a product to be not only configured once at design time, but rather by supporting flexible reconfigurations at runtime [BSBG08] within the state space of configurations. In that manner, a reconfiguration from a current configuration to a subsequent configuration corresponds to a transition from a *source configuration state s* to a *target configuration state s'* executed at runtime. By enriching the state space with reconfiguration-transitions, the state space becomes a transition system.

Transition systems [HMP92, Kel76] are used to specify the behavior of a system on the basis of state-transition graphs. The actual runtime properties of interest are associated with state labels rather than transition labels of paths of subsequent state-transitions. A DSPL describes the operation configuration states of a device, i.e., which features are (in-)active in combination. Therefore, a state is labeled with the interpretation of the features of a DSPL to denote a valid configuration (c.f. Definition 4.8). The possible reconfiguration behavior between the configuration states of a DSPL, i.e., the entering and leaving of states, is defined by the (unlabeled) transitions of such a transition system.

Bruns et al. propose to use a *Kripke Structure* ($\mathcal{KS}$) [BG99] to specify a transition system. Every state $s \in \mathcal{S}$ of such a $\mathcal{KS}$ corresponds to some valid Boolean configuration interpretation $\gamma \in \Gamma_{fm}$ of feature variables $\mathcal{F}$. Additionally, the set of states $\mathcal{S}$ contains an error state s_ϵ In this regard, every state of a $\mathcal{KS}$ corresponds to a configuration state of a state space as previously explained.

A reconfiguration is expressed via a transition relation $s_1 \longrightarrow s_2$ between two states in $\{s_1, s_2\} \in \mathcal{S}$. Thus, the interpretation of feature variables assigned to a state $s \in \mathcal{S}$ fully specify an operational configuration state of the device and the transitions are capable to express the concept of a DSPL reconfiguration.

Definition 6.1 (*Kripke Structure ($\mathcal{KS}$) [BG99]*)**.** A *Kripke Structure* $\mathcal{KS}$ is a tuple $(\mathcal{S}, \longrightarrow)$, where

- $\mathcal{S} = \hat{\Gamma}_{fm} \cup \{s_\epsilon\}$ is a finite set of *configuration states*, and
- $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a *transition relation*.

This $\mathcal{KS}$ as specified above is capable to be *non-deterministic*, i.e., a state may have more than one outgoing transition. Note that a reconfiguration based on such a transition system is discussed in detail later in this thesis in Section 6.4.

Each state is defined by its Boolean interpretation of features and denotes a configuration $\gamma \in \hat{\Gamma}_{fm}$ of feature variables w.r.t. a feature model formula *fm*. The device has to be able to deal with possible failures, such as non-resolvable conflicts of contextual requirements or erroneous features, e.g., a defect in the WLAN chip. Therefore, an additional error state is added to the set of states $\mathcal{S}$. Such an error state is denoted by $s_\epsilon \in \mathcal{S}$ as part of the $\mathcal{KS}$, with $s_\epsilon \notin \hat{\Gamma}_{fm}$.

As SAS continuously adapt themselves, it is further safe to assume that a $\mathcal{KS}$ has no terminating state by requiring the transition system to be *fully path-connected*, i.e., every state $s \in \mathcal{S}$ has to the possibility to reach every other state eventually via some path. Note that this assumption does not restrict the expressiveness of a $\mathcal{KS}$. A terminating state, e.g., an error state s_ϵ, in which the device got stuck, is expressible with a self-transition to denote a continuous loop in such a terminating state.

The $\mathcal{KS}$ may also be *fully connected*, i.e., every state is able to reach every other state in $\mathcal{S}$ with one transition, as proposed by [Hel12]. If there are no additional external constraints, a state has a transition to every state in the transition system, including itself, i.e., a self-transition, which results in 66^{66} transitions for the Nexus DSPL. However, the amount of transitions are restrictable by specifying constraints over the reconfigurations. For instance, a device may not directly switch from a source configuration, in which `Cellular Call` is currently used, to a target configuration, in which `VoIP` is used, to execute a phone call without interrupting the call. Instead, this reconfiguration has to be executed via an intermediate configuration state, in which `Cellular Call` *and* `VoIP` are both active, to initiate a handover of the currently ongoing call. Note that such restrictions are of no further concern for the Nexus DSPL running example.

A $\mathcal{KS}$ is the basis to (i) dynamically reconfigure a DSPL and (ii) to satisfy the continuously changing contextual requirements. If a reconfiguration is triggered, the state space $\mathcal{S}$ has to be searched to find an appropriate configuration state of the device that satisfies the requirements imposed by the emerged contextual situation. If no such state exists, the imposed requirements are not satisfiable w.r.t. the specification of the DSPL.

A reconfiguration transition $s \rightarrow s'$ implies a change in the interpretation of a *set* of features. Therefore, not only the interpretation of one feature may have to be adapted but the interpretation of multiple features. Assuming that the features `Navigation` and `GPS` are inactive in the currently active source state s, a transition to a target state s' where `Navigation` and `GPS` are reconfigured, i.e., changed from being inactive to active, is executed as soon as the navigation system is activated, e.g., if the smartphone is put into the docking station of a car.

The next section discusses potential effects of a context-aware DSPL on the state space and the respective reconfiguration transition system as a $\mathcal{KS}$. Further, the next section elaborates how a transition is triggered by a change of a contextual situation.

6.1.2 *Context-Enriched State Space*

According to the first goal G1 of this thesis, a context-aware DSPL has to be capable to *autonomously* react on changes in the contextual situation. Such a DSPL is able to execute reconfigurations for potentially compatible or incompatible contexts emerging at runtime without external assistance. For example, if a user enters the context `Home` and the throughput of his Internet connection is below a certain threshold, the context `Slow Network` becomes active in combination with the context `Home`. Based on the context-feature model, a corresponding configuration is derivable, which satisfies the requirements of the imposed contextual situation.

The design of a $\mathcal{KS}$ provides the basis to specify the reconfiguration behavior of a DSPL w.r.t. the contextual constraints specified in a context-feature model. In a context-enriched $\mathcal{KS}$, an association between states and contexts is derivable, i.e., which states may be chosen as target states for a contextual situation. Such an association between contexts and states is accomplished by context-coverage criteria, e.g., every single context, every valid pair-wise combination of contexts, every valid three-wise combination of contexts, etc. For example, the requirements imposed by the contextual situation {`Home`, `Slow Network`} are satisfiable by three states of the Nexus DSPL, whereas the context `Home` and `Concert` are incompatible because they rely on different communication infrastructures.

A $\mathcal{KS}$ as defined in Definition 6.1 is used to represent context-aware reconfiguration processes as specified by a context-feature model *cfm*. Each configuration state $s \in \mathcal{S}$ corresponds to a configuration $\gamma \in \hat{\Gamma}_{cfm}$ of features *and* contexts of a context-feature model *cfm* as defined in Definition 4.19. Therefore, the interpretation of features as well as supported contexts are encoded in a state $s \in \mathcal{S}$, i.e.,

- $s(f) = \mathtt{t}$ holds iff feature $f \in \mathcal{F}$ is active in the state s and
- $s(c) = \mathtt{t}$ holds iff context $c \in \mathcal{C}$ is supported by the state s.

A *context-enriched configuration state* $s \in \mathcal{S}$ is a state that reflects a configuration $\gamma \in \hat{\Gamma}_{cfm}$, for all feature variables $f \in \mathcal{F}$ *and* context variables $c \in \mathcal{C}$. Thus, a context-aware state space consists of states that fully configure all feature variables and context variables of a context-feature model *cfm*.

A state of the configuration state space is used to satisfy the requirements imposed by the currently active *contextual situation* of a device. Such a contextual situation C^+ is specified as a subset $C^+ \subseteq \mathcal{C}$ of contexts, which are currently active in the contextual environment of a device. For instance, assume that a user is at `Home` and working in his `Office`. This contextual situation is denoted as $C^+ = \{\texttt{Home}, \texttt{Office}\}$.

Notation 6.1 (Contextual Situation)**.** A *contextual situation* is denoted as a subset of contexts $C^+ \subseteq \mathcal{C}$.

Each context of a contextual situation C^+ has to be active or irrelevant in a state $s \in \mathcal{S}$ that is a suitable state for that situation, i.e.,

$$\forall c \in C^+ \; : \; s(c) \geqslant \bot.$$

Note that the case of interpreting a context as *irrelevant* $s(c) = \bot$ is discussed in detail with the introduction of a *partial state* in Section 6.3. For now, let us assume a contextual situation consists of a set of active contexts.

Every contextual situation C^+ reflects the requirements imposed by a set of active contexts. For example, for a state $s \in \mathcal{S}$ to be suitable for the contextual situation C^+=$\{\texttt{Office}, \texttt{Home}\}$ s has to interpret the individual contexts as $s(\texttt{Office}) = \texttt{t}$ and $s(\texttt{Home}) = \texttt{t}$.

The concept of an autonomous context-aware DSPL is depicted in Figure 6.5. In contrast to a standard DSPL state space depicted in Figure 6.4(b) the pre-planned context-aware state space divides the state space into several subsets. Each context-specific subset consists of configuration states that satisfy the requirements imposed by a contextual situation.

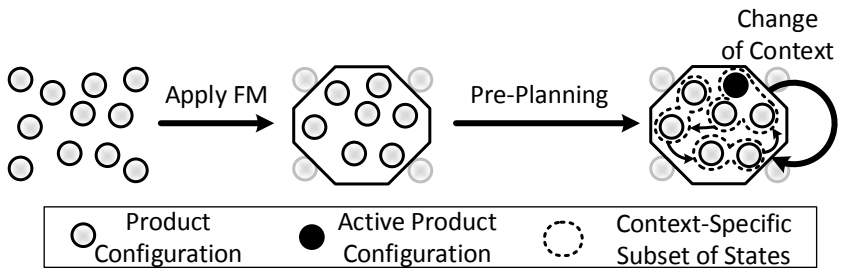


Figure 6.5: Context-Aware DSPL

Multiple configuration states may satisfy the requirements imposed by *one* contextual situation C^+. Thus, a subset of states $S \subseteq \mathcal{S}$ may be suitable for a contextual situation C^+.

A change in the contextual situation of a device, therefore, may be satisfiable by a set of states $S \subseteq \mathcal{S}$. In this regard, a transition may be executed from a currently

active state $s \in \mathcal{S}$ to one of the states $s' \in S$ that satisfy the requirements of the emerging contextual situation, as depicted on the right-hand side in Figure 6.5.

Example 6.2 (*Context Specific State Space*).

The context `Office` requires an `Infrastructure`-based communication and some possibility to execute a `Phone Call`. Additionally, the `Game Hub` is excluded from the set of suitable states that satisfy `Office`. The context `Office` is satisfiable by a set of 24 states.

In contrast to the state depicted in 6.3, which illustrates a configuration state for a standard DSPL, the state of a context-aware DSPL contains the configuration of the software features as well as of the contextual features. Figure 6.6 depicts a context-aware state for the Nexus DSPL, indicating an extract of the complete configuration for the contextual situation {`Office`}.

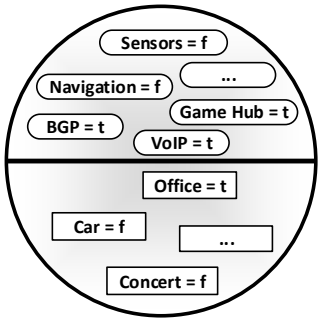


Figure 6.6: Configuration State for the Context `Office`

The contextual situation consisting of the contexts `Slow Network` and `Office` results in the intersecting subset of states that satisfy both states, consisting of six states. A combination of the contexts `Office` and `Concert` results in an empty set of suitable configuration states. Both contexts exclude each other, since `Concert` (implicitly) requires an `Ad Hoc`-based communication instead of an `Infrastructure`-based communication.

Thus, the state space may contain multiple states that satisfy a contextual situation or even states that are incompatible to any possible contextual situation. Hence, there are states in the state space, which are expendable. Therefore, a criterion to minimize the state space w.r.t. the requirements imposed by a contextual situation is discussed in the next section.

6.2 Incomplete State Space

The variability of a DSPL is specified at design time based on a context-feature model. This concept of a context-aware DSPL has the potential to rigorously pre-plan efficient reconfigurations at runtime by focusing on the aspects of an autonomous adaptation resulting from changes of the contextual environment, as previously introduced. However, to reduce the state space of a DSPL a certain

criterion is required, e.g., to identify removable states. If states are removed from the state space of a $\mathcal{KS}$, the state space becomes *incomplete*.

Recent literature shows that a mobile device, and with it the respective user, moves in certain contextual patterns [JL10, MSRJ12, Ver09]. Accordingly, some contexts are more likely to occur than others, e.g., a user is more often at *home* than in a crowded area, such as a *concert*. Thus, the state space may be reduced w.r.t. the contexts emerging at runtime. An initial set of configuration states that cover certain contextual situations has to be pre-planned in order to provide the necessary adaptation knowledge if contexts become dynamically (in-)active at runtime. For example, a suitable configuration state for contextual situations such as {`Office`} or {`Home, Slow Network`} have to be pre-planned in order to support a corresponding reconfiguration between those contextual situations. However, for every contextual situation that emerges at runtime and that is not part of the context-coverage criterion applied during the pre-planning, a suitable configuration has to be computed on-demand and integrated into the $\mathcal{KS}$.

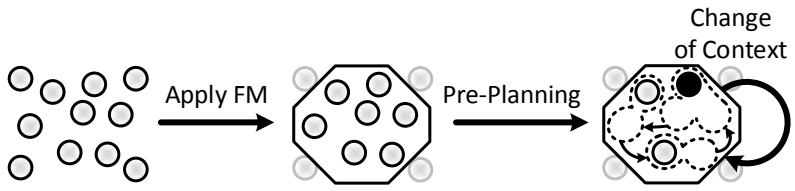


Figure 6.7: Context-Aware DSPL, Incomplete State Space

Figure 6.7 depicts this concept of an incomplete state space. Similarly to the complete state-space of a context-aware DSPL depicted in Figure 6.5, reconfiguration transitions are executed w.r.t. changes in the contextual situation. However, in contrast to a complete state space, there is only one configuration state for each contextual situation instead of a set of suitable configuration states.

In the following, I compare reconfigurations based on an incomplete state space to existing techniques and elaborate how a complete state space may be reduced to a representative set of configuration states that cover certain contextual situations.

6.2.1 *On-Demand, Pre-Planning, and a Hybrid Combination*

A complete $\mathcal{KS}$ provides the information necessary to execute a reconfiguration for every possible contextual situation emerging at runtime. The reduction of the state space to an incomplete state space relies on the observation that the set of valid configuration states of a feature model often consists of more configurations than those required by the contextual situations emerging at runtime. If the $\mathcal{KS}$ has a state for every valid configuration of a DSPL then the requirements imposed by a contextual situation may be satisfiable by several states, e.g., the contextual situation {`Car`} in the Nexus DSPL is satisfiable by 21 states. However, contextual situations that do not fulfill the criterion to reduce the state space to an incomplete state space during the pre-planning are also not part of the resulting $\mathcal{KS}$. For example, the contextual situation {`Home, Office`} is not explicitly covered by

an incomplete state space if one considers only single contexts. Thus, an appropriate reconfiguration for missing contextual situations is only executable *after* a corresponding state is computed *on-demand* at runtime and integrated into $\mathcal{KS}$.

In this regard, I extended the two existing techniques to handle runtime reconfigurations of a DSPL, (i) on-demand [FFF+13, Elk10] and (ii) a complete pre-planning [Hel12], by a third *hybrid* combination of both techniques to handle runtime reconfigurations based on an incomplete state space.

- **Complete Pre-Planning.** Potential reconfigurations of a DSPL are completely determined by a pre-planned $\mathcal{KS}$ containing every valid configuration state and every valid reconfiguration transition as depicted in Figure 6.8(a). Although it is not necessary to derive a configuration at runtime, the $\mathcal{KS}$ may become very complex. Such complexity increases the efforts for the search of a target configuration and a higher utilization of the main-memory.

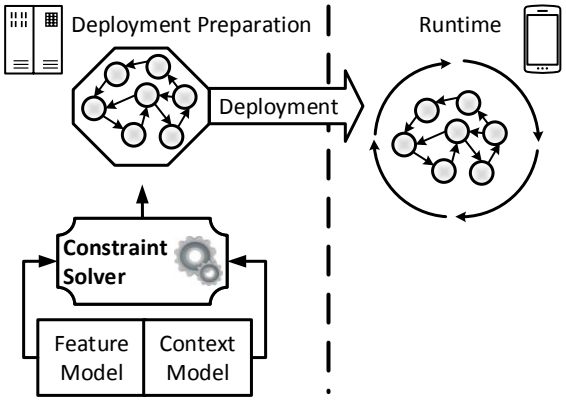
- **On-Demand.** A not yet computed reconfiguration appropriate for a particular contextual situation emerging at runtime may be also computed on-demand, e.g., by invoking a constraint solver at runtime, each time the contextual situation changes as depicted in Figure 6.8(b). Only during the derivation of a configuration this approach imposes a considerable utilization of main memory. The permanent memory consumption of the state space is very low because the state space is initially empty and extended according to the contextual situations emerging at runtime. However, the continuous derivation of a valid configuration drains the battery of a mobile device.

The usage of an incomplete state space represents a combination of both approaches with the benefit that the disadvantages, i.e., search effort, memory utilization, and drain of battery, are mitigated.

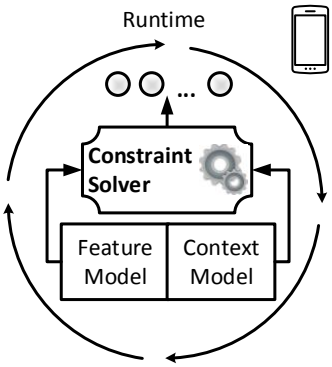
- **Hybrid Combination.** In pre-planned *incomplete* $\mathcal{KS}$ not all reconfigurations are fully explored, i.e., some valid configurations are not reachable by corresponding transitions as illustrated in Figure 6.8(c). Thus, the usage of a solver is required at runtime. Considering the assumption that (i) a user mostly moves in the same contextual pattern and (ii) the $\mathcal{KS}$ is equipped with a set of states that cover certain contextual situations, the necessity to extend the transition system at runtime is reducible to a minimum in comparison to an on-demand solving approach. At the same time, the search efforts within a state space and memory consumption of the state space are also reduced in comparison to a complete pre-planning approach.

Hence, to avoid an explicit representation of the *entire* configuration state space, states may be removed w.r.t. a certain context-coverage criterion. If a contextual situation emerges that is not covered by such an incomplete state space, a state is computed on-demand and integrated into the transition system accordingly.

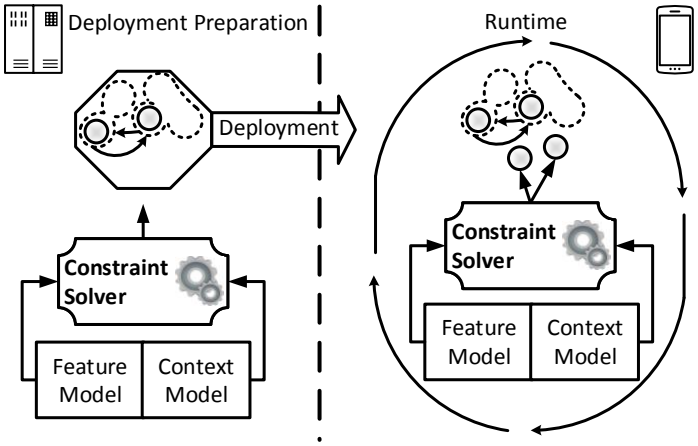
The derivation of an incomplete state space requires a criterion, which is used to identify suitable states that remain in the incomplete state space. As an example for such a reduction criterion, context-coverage criteria are discussed in the next section. Note that further criteria are imaginable that fit the individual needs



(a) Complete



(b) On-Demand



(c) Hybrid Combination of Both

Figure 6.8: Strategies for Reconfiguration at Runtime

of the application domain such as a selection criterion for states based on the popularity at runtime. However, in this thesis I only elaborate the contextual coverage reduction criterion because I did not investigate runtime feedback to determine the popularity of certain states.

6.2.2 *Context Coverage Criteria*

To reduce the state space to an incomplete state space, a criterion is required to select the states that are kept in the state space. Therefore, I propose to pre-plan an incomplete state space that covers certain contextual situations, which may emerge at runtime. In order to systematically control the reduction of a state space, the amount of contexts in a contextual situation may be limited. Therefore, an automatically generatable set of context combinations $\mathbb{C}$ is required, describing a set of contexts combinations to denote contextual situations which may emerge at runtime.

In this regard, the power-set $\mathbb{P}(\mathcal{C})$ of the contexts $\mathcal{C}$ is computed to define a set of context combinations $\mathbb{C}$. The power-set provides the set of all subsets of $\mathcal{C}$, including the empty set $\emptyset$. Since the empty set does not contain any contexts the empty set $\emptyset$ is excluded from $\mathbb{C}$. Note that $\mathbb{C}$ may also contain combinations of contexts, which are *not* satisfiable by a context-feature model. As intended by the context-feature model specification, such contextual restrictions are not coverable by the incomplete state space.

The power-set $\mathbb{P}(\mathcal{C})^k$ is restricted by the parameter k, which limits the size of all subsets. With this, the limited (power-set) of context combinations $\mathbb{C}^k$ is denoted as follows.

Notation 6.2 (Context Coverage Criterion)**.** With a given $k \in \mathbb{N}$, the combinatorial context coverage $\mathbb{C}$ is denoted as a limited power-set of context combinations C^+

$$\mathbb{C}^k := \mathbb{P}^k(\mathcal{C})\backslash\emptyset = \{C_1^+, \ldots, C_k^+ \mid C_i^+ \subseteq \mathcal{C} \text{ with } i = 1 \ldots k\}.$$

Thereby, each set of contexts $C^+ \in \mathbb{C}^k$ is limited in its size by k

$$\forall C^+ \in \mathbb{C}^k \; : \; |C^+| \leqslant k.$$

Note that a coverage criterion of k always subsumes all criteria that are smaller than k, e.g., a k=2 also covers all context combinations that are covered by a k=1.

For example, an extract of a limited context combination for the Nexus DSPL is the set $\mathbb{C}^k = \{\{$`Car`$\}, \{$`Office`$\}, \{$`Home`$\}, \{$`Concert`$\}, \{$`Slow Network`$\}, \{$`Car, Office`$\}, \{$`Car, Home`$\}, \{$`Car, Concert`$\}, \{$`Car, Slow Network`$\}, \{$`Office, Concert`$\}, \{$`Office, Home`$\}, \{$`Office, Slow Network`$\}, \ldots\}$ for k=2. As previously explained, this set contains context combinations, which are incompatible w.r.t. the context-feature model of the Nexus DSPL depicted in Figure 4.2, such as {`Office`, `Concert`}. In such a case, no state is derived and the combination is discarded.

A coverage of every *single* context, i.e., setting k=1, offers a radical reduction of the state space. In addition to this one-wise coverage of single contexts, arbitrary

combinations of contexts, i.e., k>1, may emerge at runtime. Hence, appropriate configurations satisfying the k-wise combinations of contexts have to be provided in order to execute the respective adaptation to a changing contextual situation. Such a k-wise combinatorial context coverage criterion may restrict the computation of (sub-)sets of configurations at design time, whereas configurations for uncovered combinations beyond k-wise may, again, be computed on-demand as depicted in Figure 6.8(c).

A suitable trade-off between memory consumption utilized by a large $\mathcal{KS}$ and computation of reconfigurations at runtime is to be found by means of an appropriate combinatorial context parameter k. In this regard, the state space of the Nexus DSPL is reducible from 66 states to an incomplete state space of 12 states for k=2 for the cost of computing a reconfiguration for every contextual situation emerging at runtime, which consists of more than 2 active contexts.

Example 6.3 (*Context Coverage*).

The Nexus DSPL allows a combination of up to three contexts. For example, the contexts `Concert`, `Slow Network`, and `Car` represent a valid 3-wise combination. The constraints in the context-feature model depicted in Figure 4.2 allow all of those three contexts to be active at the same time.

The power-set $\mathbb{P}$ for these three contexts is listed in Table 6.1, in which every active context is considered to be part of a contextual situation and every inactive context is not considered to be part of a contextual situation. To derive a $\mathcal{KS}$ with an incomplete state space for a 3-wise context coverage, every combination of active contexts ($\oplus$) is used to compute and integrate a suitable configuration state.

`Concert`	`Slow Network`	`Car`
$\ominus$	$\ominus$	$\oplus$
$\ominus$	$\oplus$	$\ominus$
$\ominus$	$\oplus$	$\oplus$
$\oplus$	$\ominus$	$\oplus$
$\oplus$	$\oplus$	$\ominus$
$\oplus$	$\oplus$	$\oplus$

Table 6.1: Power-Set 3-wise Context Combination

In the Nexus DSPL all the listed contextual situations are valid w.r.t. the context-feature model. Thus, 6 states are derived at design-time to cover the contextual situations listed in Table 6.1 at runtime.

If a contextual situation is satisfiable by several configuration states, further reduction or optimization strategies may be applied for a more detailed selection of configuration states. For example, a set of states that satisfies the requirements imposed by a contextual situation may be reduced to one representative state with the non-functional properties, which are considered to be most suitable for the contextual situation, e.g., the state with the least energy consumption.

In the process of deriving an incomplete state space w.r.t. a set of contextual combinations $\mathbb{C}^k$ it may occur that the same device configuration is applicable for different contextual situations. The reason for this is that not only a single context combination is representable by a set of configuration states, as stated on page 105, but also a *single* configuration state may be suitable for *multiple* context combinations. In this regard, a given configuration $\gamma\big|_{\mathcal{F}} \in \Gamma_{cfm}$, which interprets only the subset of features $\mathcal{F}$ of a context-feature model $cfm \subseteq \mathcal{FM}_{\mathcal{C}\cup\mathcal{F}}$, may support multiple combinations of active contexts $C^+ \in \mathbb{C}_i$ in a configuration state $s_i \in \mathcal{S}$. According to Notation 6.1, each of these context combinations may be interpreted as a contextual situation, in which every context $c \in C^+$ is either interpreted as active, $\gamma(c) = \mathtt{t}$, or as irrelevant, $\gamma(c) = \bot$, for that contextual situation. Thus, a state s_i may be suitable for multiple interpretations of a contextual situation, in which $s_i(c) \geqslant \bot$ holds at least for every active context $c \in C^+$, and s_i corresponding to a valid interpretation of features $s_i \sqsubseteq \gamma\big|_{\mathcal{F}}$.

Definition 6.2 (*Valid Context Combinations for a Configuration State*)**.** Let $cfm \subseteq \mathcal{FM}_{\mathcal{C}\cup\mathcal{F}}$ be a context-feature model and $\gamma\big|_{\mathcal{F}} \in \Gamma_{cfm}$ be a valid configuration of the features $\mathcal{F}$ of $\mathcal{FM}$.

The set of valid context combinations $\mathbb{C}_i$ for a configuration state $s_i \in \mathcal{S}$ is characterized as follows

$$\mathbb{C}_i = \{C^+ \subseteq \mathcal{C} \mid \forall c \in C^+ \; : \quad s_i(c) \geqslant \bot \wedge s_i \sqsubseteq \gamma\big|_{\mathcal{F}}\},$$

with s_i refining a given configuration $\gamma\big|_{\mathcal{F}}$ of features.

Correspondingly, $\mathbb{C}_i$ consists of a set of context combinations C^+, which in turn consists of a set of contexts. For example, $\mathbb{C}_i$ ={{`Home`}, {`Slow Network`}, {`Home`, `Slow Network`}} for the state s_i depicted in Figure 6.15.

The specification of a context-feature model *cfm* restricts the overall possibilities for a valid combination of contexts, e.g., with constraints between contexts and features θ, constraints between features φ, and constraints between contexts τ. Based on the previous definition of valid context combinations for a configuration state and the Notation 6.2 of limited context combinations the definition of all *valid context combinations* $\mathbb{C}_{cfm}$ for a context-feature model *cfm* is introduced. A context combination $C^+ \subseteq \mathbb{C}_{cfm}$ is valid if a valid configuration $\gamma \in \hat{\Gamma}_{cfm}$ exists, in which every single context $c \in C^+$ is active, $\gamma(c) = \mathtt{t}$. Further, the notation for a *limited* combinatorial subset of all valid combinations $\mathbb{C}^k_{cfm}$ that may contain up to *k*-context elements per combination is defined as follows.

Definition 6.3 (*Valid Combinatorial Context Combinations for cfm*)**.** With a given context-feature model *cfm* and a set of contexts $\mathcal{C}$, the set of *all valid* context combinations C^+ is denoted as follows.

$$\mathbb{C}_{cfm} = \{C^+ \subseteq \mathcal{C} \mid \exists \gamma \in \hat{\Gamma}_{cfm},\ \forall c \in C^+ : \gamma(c) = \mathtt{t}\}$$

in which $\hat{\Gamma}_{cfm}$ denotes the set of all valid configurations for *cfm*. Furthermore, the combinatorial subset

$$\mathbb{C}^{k}_{cfm} = \{C^{+} \in \mathbb{C}_{cfm} \mid k \geqslant |C^{+}|\}$$

denotes the set of valid k-wise combinations of contexts.

An incomplete state space has the disadvantage that a configuration state has to be derived on-demand if a contextual situation is not covered by a criterion used to reduce the state space. The next section, therefore, introduces the concept of a partial state to reduce the amount of reconfigurations by making a state more flexible w.r.t. contextual changes.

6.3 Abstraction with Partial States

Up to this point of the thesis, the states of a DSPL state space correspond to complete configurations, which are applicable as a configuration utilized at runtime. An incomplete state space reduces the size of a complete state space by removing states that are considered to be irrelevant or redundant. Such a reduction is executed based on a reduction criterion. Correspondingly, an incomplete state space is derived using knowledge available at design time. However, according to research challenge two (RC2), adaptation behavior is not entirely plannable prior to runtime. Therefore, the adaptation knowledge should provide a certain flexibility w.r.t. unplanned reconfigurations.

For example, based on a context-coverage criterion, the state space provides suitable states for contextual situations such as {`Home, Office`}. These pre-planned, representative states only cover contextual situations for which they have been derived. If a user has individual requirements that do not contradict the requirements imposed by a contextual situation but diverge from the representative state for that contextual situation, a new configuration has to be derived. For example, the contextual situation {`Home, Office`} may be coverable by a state, in which the feature `Gyroscope` is active, and a state, in which it is inactive. Based on a combinatorial context coverage, the incomplete state space provides only one state for that contextual situation. Now let us assume that `Gyroscope` is configured as inactive in the state that is chosen for {`Home, Office`}. If the user wants to do some workout during the lunch break and further wants to track his calory consumption, the feature `Gyroscope` needs to be activated. To satisfy this user requirement, a constraint solver needs to be executed to derive a new configuration state.

To tackle this problem of lacking flexibility in an incomplete state space, I propose the concept of a *partial state*. According to Definition 4.16, DSPL configurations may be partial, e.g., if a feature is irrelevant for a contextual situation. In the following, I denote such contexts and features that are (arbitrarily) configurable in a state as *unrestricted* contexts and features.

Such unrestricted contexts and features are exploitable to derive a partial state and, thereby, gain flexibility in the state space. Therefore, it is possible to cover *more* configurations with the *same* number of (partial) states. In this regard, it is also possible to cover the *same* number of configurations with *less* (partial) states.

Therefore, partial states have the additional benefit of further reducing the size of a state space.

In the following, the concept of a partial state is introduced to (i) gain flexibility and (ii) further reduce the size of a state space. In order to derive such a partial state an extensive usage of constraint solvers is required. Therefore, details on techniques to compute partial states by using constraint solvers are discussed.

6.3.1 *Unrestricted Features and Partial States*

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 valuation of c may be left open (*unrestricted*) in that state, i.e., a context change affecting c does not require a state change and no reconfiguration is to be performed. Accordingly, setting a feature f to unrestricted in a state denotes that the actual configuration decision for f does not affect the context requirements. In this regard, $s(f) = \bot_U$ denotes a state whose active contexts do not depend on whether f is active or not, i.e., f is an unrestricted feature. Further, $s(c) = \bot_U$ denotes states whose feature selections are not affected when the context c is entered or left, i.e., c is an unrestricted context. Thus, a partial state, therefore, subsumes a set of concrete states being equivalent w.r.t. to

- features that are configured to be (in-)active and/or
- context that are being entered/left.

In contrast to variable features that are used as a criterion to reduce a context-feature model in Chapter 5, an unrestricted feature is not a choice of the developer. Whether a context or feature in a state $s \in \mathcal{S}$ is identifiable as unrestricted depends on (i) the constraints imposed by *cfm* and (ii) the interpretation of remaining features and contexts in s.

For example, in the partial state depicted in Figure 6.9 the features `Navigation` and `VoIP` are unrestricted. In this regard, a partial state $s \in \mathcal{S}$ is a state that corresponds to a valid partial configuration $\gamma \in \check{\Gamma}_{cfm}$.

Since every unrestricted feature $s(f) = \bot_U$ and unrestricted context $s(c) = \bot_U$ are arbitrarily reconfigurable, an interpretation of these context features as active *and* inactive must both satisfy the context-feature model formula. Therefore, a *partial state* represents a third type of a state in addition to a *complete configuration state*, and an *error state*.

Since partial states are abstract states, they do not fully specify a reconfiguration of all features. Therefore, a partial state has to be further refined until a runtime configuration is derived. However, since the unrestricted features of a partial state may be configured *arbitrarily*, the derivation of such a runtime configuration does *not* require the execution of a constraint solver. For example, a reconfiguration between the four states subsumed by the partial state depicted in Figure 6.9 is executable without the need for a single solver call.

Analogously to Notation 6.2 for the limited combinatorial coverage of contexts, the pre-planning of partial states is also measurable by a combinatorial coverage criterion of features *and* context that are simultaneously unrestricted in a partial

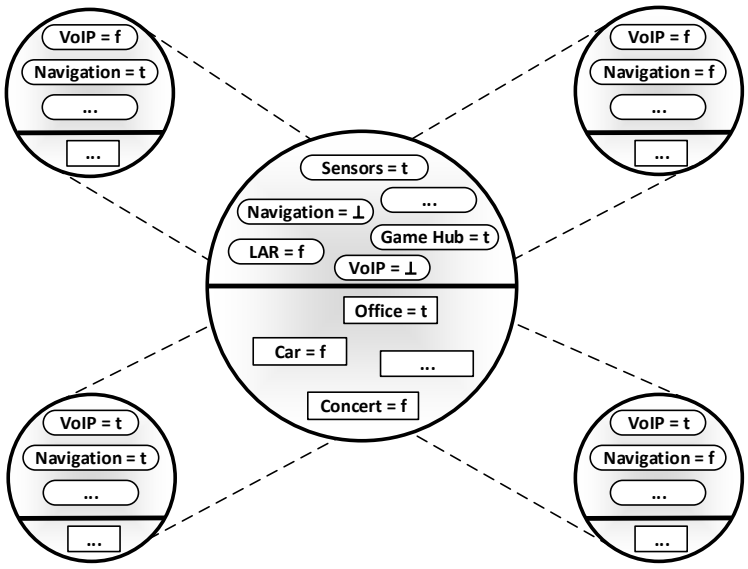


Figure 6.9: Partial State Abstraction of four fully Configured States

state. The limitation of combinatorial unrestricted contexts/features is denoted by the parameter l. For example, the partial state depicted in Figure 6.9 has an upper bound of unrestricted features of l=2.

The computation of partial states containing an l-wise combination of unrestricted contexts/features is very expensive. It requires to analyze whether any possible interpretation of the l unrestricted contexts/features satisfy the given context-feature model *cfm*. It is important that *all* contexts or features, which are unrestricted in a partial state, are *arbitrarily configurable* in *combination*, i.e, they have to be independent from each other. However, once identified, l-wise partial states reduce

- the state space by subsuming 2^l states into one and
- runtime solving efforts by gaining flexibility in the incomplete state space

and, thereby, supporting unplanned adaptation request. If a single feature or context is unrestricted it abstracts from two fully configured states.

Example 6.4 (*Partial State*).

An example for a pair-wise combination of unrestricted features, i.e., $l = 2$, are `Navigation` and `VoIP` of the Nexus DSPL as depicted in the state $s \in \mathcal{S}$ in Figure 6.9. All possible configuration combinations of those two features, i.e., {s(`Navigation`)=`t`, s(`VoIP`)=`t`}, {s(`Navigation`)=`f`, s(`VoIP`)=`t`}, {s(`Navigation`)=`t`, s(`VoIP`)=`f`}, {s(`Navigation`)=`f`, s(`VoIP`)=`f`}, result in a valid configuration w.r.t. the constraints imposed by the *cfm* in Figure 2.12. Thus, a partial state with a combinatorial unrestricted of two abstracts from four fully configured states.

In contrast, the two features `Navigation` and `Sensor` may not be unrestricted in the same state. Even if both of them may be set to unrestricted individually,

the constraints `Navigation`→`GPS`→`Sensor` prohibits both features from being unrestricted in a partial state.

Hence, a partial state provides the means for a further reduction of the size of a state space in accordance with the concept of an incomplete state space. By the subsumption of multiple states in a single partial state the overall number of states is reduced.

A brute force approach to identify combinations of unrestricted contexts and features requires the derivation of every configuration possibility of a combination contexts and/or features in *cfm*. For example, a combination of up to two features/contexts in the Nexus DSPL results in 276 solver calls to derive partial states that may contain up to two unrestricted features and 2,024 solver calls to derive partial states that may contain up to three unrestricted features. To avoid such an exponential blow-up of computational efforts, constraint solving techniques, as previously introduced in Section 4.1.3, are exploited to identify unrestricted features in the next section.

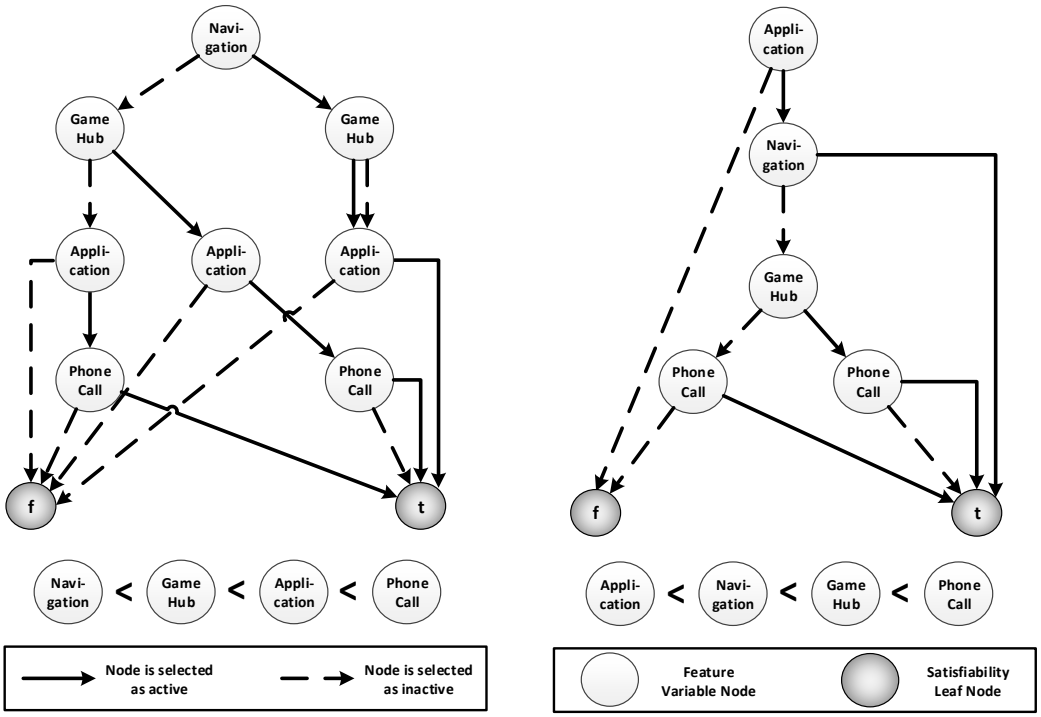
6.3.2 *Identification of Unrestricted Features*

Constraint solvers are used to derive a configuration w.r.t. a given context-feature model formula *cfm*, as explained in Section 4.1.3 of this thesis. Constraint solvers that are based on binary decision diagrams (BDDs) may be optimized by arranging the variables in a certain order [FYBSV93, ZBC96]. However, techniques for re-ordering variables are not only applicable to optimize the size of the BDD but are also exploitable to identify unrestricted features.

IMPACT OF VARIABLE ORDERING. For a given context-feature model there is exactly *one* BDD for *one* specific ordering of the variables. BDDs that are based on the same feature model but have a different ordering of the variables usually differ in their number of nodes and, therefore, also in their memory consumption. The problem of finding an optimal ordering of the variables for a feature model with a minimal number of nodes is considered to be NP-complete [BW96].

The identification of unrestricted context/feature variables depends on the ordering of a BDD. For each ordering the set of unrestricted variables is different in its composition of variables as well as in its size. A BDD orders the variables in a hierarchical manner.

For example, the feature variable order depicted in Figure 6.10(a) is `Navigation < Game Hub < Application < Phone Call`. A feature variable is identifiable as unrestricted for a partial state if all paths result in a satisfiable configuration, i.e., the ⓣ-leaf node in Figure 4.1(b). If `Navigation`, `Application`, and `Game Hub` are interpreted as active, all paths below `GPS` lead to ⓣ. Therefore, `Phone Call` is unrestricted for this specific ordering of variables.



(a) Ordering One unrestricted Feature (b) Ordering Two unrestricted Features

Figure 6.10: BDDs with Different Ordering of Variables

Example 6.5 (*Impact of Variable Order*).

Figure 6.10 depicts two BDDs for the same extract of the Nexus DSPL with different ordering. Both BDDs correspond to the or-group {`Navigation, Game Hub, Phone Call`} of the feature `Application`.

Figure 6.10(a) depicts a BDD for an random ordering of the variables. Following incoming edges of the satisfiable-leaf ⓣ one can recognize that by interpreting the features `Navigation`, `Game Hub`, and `Application` as active, `Phone Call` becomes irrelevant, i.e., `Phone Call` is unrestricted w.r.t. the remaining interpretation of features. In contrast to the ordering in Figure 6.10(a), the ordering depicted in Figure 6.10(b) results in two unrestricted features. Following the incoming edges of the satisfiable-leaf ⓣ in backward direction one can recognize that by interpreting `Navigation` and `Application` as active, the features `Game Hub` and `Phone Call` become unrestricted.

These examples illustrate that a feature is easier to identify as unrestricted if it is at the lower end of the variable ordering. A brute-force approach to identify unrestricted features is to generate BDDs according to all possible variable orderings, which would result in $|\mathcal{F} \cup \mathcal{C}|$ BDDs. Each BDD is parsed to find unrestricted features, starting with the satisfiable leaf-node ⓣ.

In contrast to existing approaches, I do not aim for an ordering of the variables that leads to a minimal BDD [FYBSV93, ZBC96]. Instead it is my goal to find as much combinatorial unrestricted features as possible to maximize the variability in a partial state in an efficient manner.

DERIVING PARTIAL STATES. To maximize the amount of unrestricted features the different orderings of the context and feature variables of a context-feature formula *cfm* have to be analyzed. However, the overall best result is only achievable if *all* the different orderings of features are analyzed, which corresponds $|\mathcal{F} \cup \mathcal{C}|$ possibilities. A structured process to maximize the amount of unrestricted features in an efficient manner without investigating all possible variable orderings is introduced in the following.

The process how unrestricted features are discovered and maximized for a single configuration is depicted in Figure 6.11. A given context-feature model is transformed into a BDD representation with an initial ordering of the variables that is random. Based on that ordering and a given partial configuration, e.g., provided by context coverage criterion to derive an incomplete state space, unrestricted contexts and features are identified.

Since I aim at maximizing the number of unrestricted features for a partial state, the variables have to be reordered to maximize the number of those features. The approach refines the given partial configuration by configuring features that are necessary to be active or inactive in order to support other features to be unrestricted.

The approach computes a set of partial states. Each of those partial states correspond to a refinement of the given partial configuration and each partial state differs in its set of unrestricted features. This set may be ordered according to the number of unrestricted features contained in each partial state. If there are several partial states with the same amount of unrestricted features, a state may either be randomly selected or selected based on the features that are unrestricted. For example, it is more likely that `Phone Call` and `Navigation` are reconfigured dynamically at runtime than `Game Hub`. In such a case, a partial state is chosen, in which `Game Hub` is active and `Phone Call` as well as `Navigation` are unrestricted. The derived partial states are integrated into the incomplete state space that is deployed on the devices in order to (i) further reduce the size of and search efforts in the state space as well as (ii) gain additional flexibility at runtime. To avoid an analysis of *every* possible variable ordering, I developed three techniques to derive a set of different variable orderings, which are based on

- a randomized order of variables,
- a heuristic to order variables, and
- a genetic algorithm to order variables,

are introduced in the following.

RANDOM ORDERING. The random-based strategy of ordering the feature variables for a BDD is limited by the number of iterations n. This strategy computes a set of n randomly generated variable orderings, which are used to identify unrestricted features. The set does not contain any redundant variable orderings.

HEURISTIC ORDERING. A heuristic is derivable from the constraint density of the feature variables. A feature that is part of more constraints is less likely

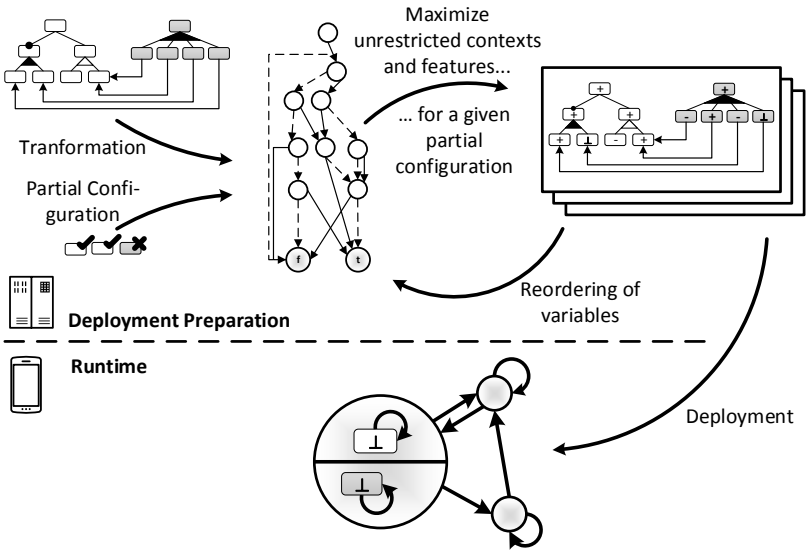


Figure 6.11: Overview of the identification of unrestricted variables. During the deployment preparation, a feature model formula is transformed into a BDD representation. Based on a given partial configuration, the variables in the BDD are re-ordered several times to identify the configuration, contains the most unrestricted variables. This refined partial configuration is then integrated into the state space and deployed on the mobile device for a runtime reconfiguration.

configurable as unrestricted because other features depend on it, i.e., require it or exclude it. In this regard, I derived a heuristic that relies on the assumption that it becomes less likely for a feature to be unrestricted the more constraints a feature is associated with.

The ordering of the feature variables corresponds to the constraint density of the features. Depending on the constraint density, features are ordered in a descending manner. In this regard the following heuristic h is applied to derive an ordering of the context and feature variables $cf \in \mathcal{C} \cup \mathcal{F}$ for a context-feature model cfm

$$h(cf) = \mathtt{count}(cf, cfm).$$

Thus, the heuristic counts the occurrence of a variable cf in the propositional formula representation of a context-feature model.

Example 6.6 (*Metric Variable Ordering*).

Figure 6.12 depicts an example for the variable ordering for the features of the `Application` or-group `Navigation`, `Game Hub`, and `Phone Call`. The constraints of the feature model are transformed into a Conjunctive Normal Form (CNF) representation of the context-feature model formula. The heuristic is applied on that CNF and counts the occurrence of the variables in the formula. For example, `Application` is part of five constraints. The group-features `Navigation`, `Game Hub`, and `Phone Call` occur equally two times in the CNF.

This underlines my assumption that it is less likely to configure a feature as unrestricted if it has a high constraint density. `Application` is not identifiable as unrestricted, whereas the features `Navigation`, `Game Hub`, and `Phone Call` have the same likelihood to be unrestricted. As long as one of these three features is active, the remaining two features are unrestricted.

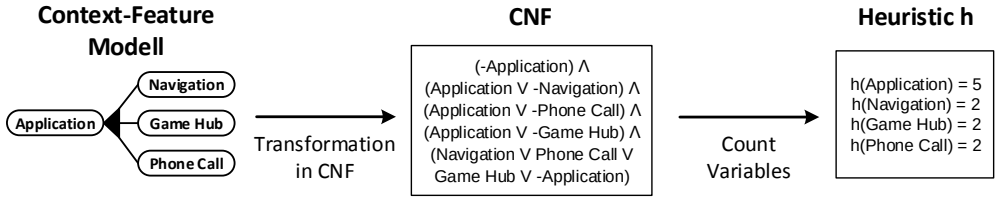


Figure 6.12: Variable Reordering According to Metric

Genetic Ordering. The third strategy to identify unrestricted context and feature variables is based on a modified genetic algorithm [RN04]. Genetic algorithms are used as search algorithms and are inspired by the evolutionary theory. The search is executed iteratively over so called generations of result sets.

The genetic algorithm is parameterized by the following three parameters

- the size of a population p contained in a generation. This corresponds to the number of variable orderings in a generation.
- the number of generations n that are created. With each generation, the amount of unrestricted features may increase.
- the probability of a mutation μ of the variable ordering in a generation.

Figure 6.13 depicts on overview of the iterative process to derive a genetic ordering of variables in order to maximize unrestricted contexts/features. This process is summarizable in five distinctive steps.

(i) The approach starts with an initial *start population* that represents the first generation. Every population consists of a set of p different BDD orderings of the variables. For example, Figure 6.13(i) has a p of four. Each of those four orderings is randomly created for the initial population.

(ii) A *fitness function* computes a weight for every ordering. Since it is my goal to maximize the amount of unrestricted variables the fitness function rewards an ordering, which leads to a higher amount of unrestricted variables. The orderings in a population are analyzed as previously illustrated in the example depicted in Figure 6.10. For example, in Figure 6.13(ii), the third ordering has the highest probability for unrestricted feature variables, whereas the second ordering has the lowest probability.

(iii) The *selection* uses the weight of the fitness function to identify the variable orderings with the highest probability for unrestricted variables. Variable

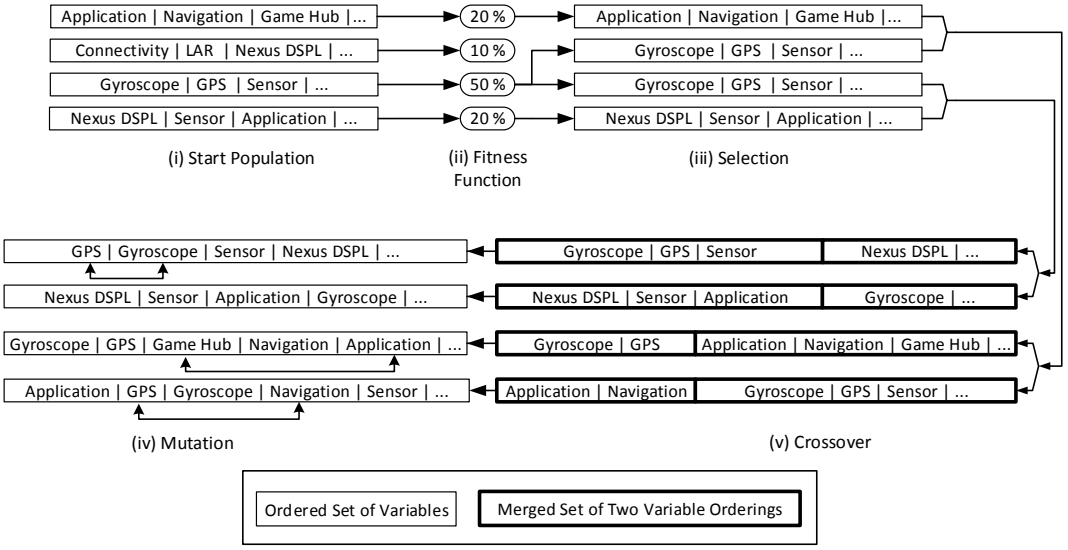


Figure 6.13: Computation of a Successor Generation Computed (adopted acc. [RN04])

orderings with a low probability to contain unrestricted variables are discarded and replaced with a variable ordering with a high probability for unrestricted variables. For example, the selection depicted in Figure 6.13(iii) discards the second variable ordering because it has only a probability of 10% for unrestricted features.

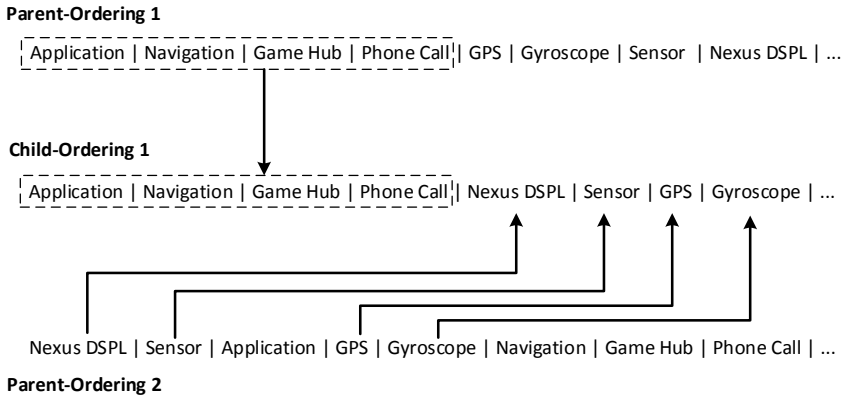
(iv) During the *crossover* pairs of variable orderings within a population are combined, which results in a new child-ordering of variables for every pair of the parent-orderings. To derive two new child-orderings, a point of intersection between those two parent-orderings is randomly generated. For the first child the variable ordering of the first parent is inherited up to this point of intersection. The variables after that point of intersection are inherited w.r.t. the second variable orderings, iff they are not already part of the child ordering. Analogously the inheritance is exchanged for the second child-ordering and the variable ordering of the second parent-ordering is inherited instead. For example, the crossover depicted in 6.13(iv) merges the orderings of the first and second variable ordering up to a point of intersection of three to derive two new child orderings. A more detailed example for the crossover of different variable orderings is depicted in Figure 6.14 and is elaborated in the example below.

(v) During the *mutation* the child-orderings resulting from the crossover are randomly changed w.r.t. a mutation probability μ, i.e., the position of variables are reordered randomly based on μ. For example, the mutation depicted in Figure 6.13(v) exchanges the first and second feature in the first variable ordering, whereas the second variable ordering is not mutated.

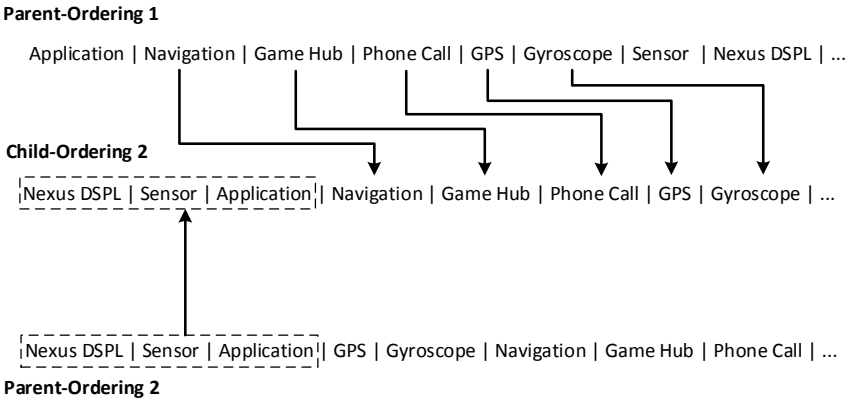
After the mutation is finished for every ordering within a generation, the derivation of a new succeeding generation is finished and the successor generation

becomes the new start population. This process is iteratively repeated until n generations have been analyzed. With every derivation of a new population the amount of unrestricted variables may only become higher because populations with a lower amount of unrestricted variables are discarded.

Example 6.7 (*Crossover of Variable Orderings*).



(a) First Child-Ordering



(b) Second Child-Ordering

Figure 6.14: The Modified Crossover of Variable Orderings (adopted acc. [ZBC96])

Figure 6.14 depicts a detailed example of a crossover between two orderings *parent-ordering 1* and *parent-ordering 2*. The derivation of the first *child-ordering 1* is depicted in Figure 6.14(a). The point of intersection is (randomly) set after the fourth feature. Accordingly, the first four features of *parent-ordering 1* are directly inherited to *child-ordering 1*, i.e., `Application` < `Navigation` < `Game Hub` < `Phone Call`. The remaining features are ordered w.r.t. the second *parent-ordering 2*, i.e., `Nexus DSPL` < `Sensor` < `GPS` < `Gyroscope`.

The derivation of the second *child-ordering 2* is analogously depicted in Figure 6.14(b). In this case, *child-ordering 2* is ordered w.r.t. a point of intersection after the third feature in *parent-ordering 2*. Thus, the first three features of *parent-ordering 2* are inherited to *child-ordering 2* and the remaining features are ordered w.r.t. *parent-ordering 1*.

Summarizing, the incomplete state space *reduces* the size of a configuration state space for a DSPL and, therefore, also the search efforts to identify a suitable target configuration for a reconfiguration. The reduction is achieved by selecting representative states w.r.t. the contextual specification. By applying a context-coverage criterion, an incomplete state space is pre-plannable that covers certain combinations of contexts, which may emerge at runtime as a contextual situation. However, since the state space is incomplete an on-demand derivation of a suitable state for a context that is not pre-planned may be necessary. To further minimize the necessity of such on-demand derivations of configuration states, I introduced the concept of a partial state. With partial states the pre-planned incomplete state space becomes more flexible regarding unplanned contextual changes. A partial state contains unrestricted features that are irrelevant for that particular state. In this regard, a partial state abstracts from several completely configured states and, therefore, further reduces the size of a state space. This has the benefit, that *every* unrestricted feature or unrestricted context is arbitrarily reconfigurable *without* the need to execute a solver call. Unrestricted context and feature variables may be identified based on the ordering of variables in the BDD representation of a propositional formula. I introduced three different approaches to identify unrestricted features, based on (i) a random ordering, (ii) a heuristical ordering, and (iii) a genetic ordering.

However, the semantics of a standard reconfiguration transition of a *KS* as provided by Definition 6.1 has to be adapted to handle reconfigurations in a state space containing partial states. Therefore, I introduce the semantics of a reconfiguration transition for a state space that is incomplete and contains partial states. Further, to ensure the soundness of my approach, I prove the correctness of fundamental reconfiguration properties in the next section.

6.4 Three-Valued Reconfiguration Semantics

In addition to a standard $\mathcal{KS}$, which is based on a complete state space, Bruns et al. also propose to use a partially defined *Kripke Structure* ($\mathcal{PKS}$) [BG99] as a concept to specify transition systems with *partial states*. With such a $\mathcal{PKS}$ it is possible to abstract traditional states to partial states by applying a three-valued logic (c.f. Definition 4.11).

As introduced in Definition 6.1 of a standard Kripke Structure $\mathcal{KS}$, every state $s \in \mathcal{S}$ in a Partial Kripke Structure $\mathcal{PKS}$ is labeled with a configuration $\gamma \in \hat{\Gamma}_{cfm}$. However, in contrast to a standard $\mathcal{KS}$, the states correspond to a variable interpretation according to the *three-valued logics domain* $\mathbb{B}_{\perp}$. In this regard, every state $s \in \mathcal{S}$ denotes an valid interpretation of context variables $\mathcal{C}$ and feature variables $\mathcal{F}$. Note that for a $\mathcal{PKS}$ such a state corresponds either to a valid complete configuration $\gamma \in \hat{\Gamma}_{cfm}$, i.e., a complete state, or a valid partial configuration $\gamma \in \check{\Gamma}_{cfm}$, i.e., a partial state. Thus, the set of (partial) states is defined as

$$\mathcal{S} \subseteq \Gamma_{cfm} \cup s_{\epsilon}, \text{ with } \Gamma_{cfm} = \hat{\Gamma}_{cfm} \cup \check{\Gamma}_{cfm}.$$

Additionally, the set of states $\mathcal{S}$ contains an error state s_ϵ, with $s_\epsilon \notin \Gamma_{cfm}$. A reconfiguration is expressed via a transition relation $s \longrightarrow s'$ between two states s, s' in $\mathcal{S}$.

Definition 6.4 (*Partial Kripke Structure [BG99]*)**.** A *partial Kripke Structure* $\mathcal{PKS}$ is a tuple $(\mathcal{S}, \longrightarrow)$ where

- $\mathcal{S} \subseteq \Gamma_{cfm} \cup \{s_\epsilon\}$ is a finite set of *(partial) configuration states*, and
- $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a *transition relation*.

Thus, with a standard $\mathcal{KS}$ the state space $\mathcal{S}$ specifies *which* configuration states are possible for an SAS and transitions denote the possible reconfiguration behavior of an SAS. However, a $\mathcal{PKS}$ is further capable to denote partial configurations via the concept of a partial state, which may be dynamically refined at runtime.

A $\mathcal{PKS}$ is used to represent context-aware reconfiguration processes as specified by a context-feature model *cfm* introduced in Definition 4.19. Each state $s \in \mathcal{S}$ of the configuration space of a $\mathcal{PKS}$ corresponds to (i) a valid complete configuration $\gamma \in \hat{\Gamma}_{cfm}$ or (ii) a valid partial configuration $\gamma \in \check{\Gamma}_{cfm}$ of features *and* contexts. In case of (i) a state $s \in \mathcal{S}$ encodes the interpretation of features as well as supported contexts supported at runtime, i.e.,

- $s(f) = \mathtt{t}$ holds iff feature $f \in \mathcal{F}$ is active in the current configuration,
- $s(c) = \mathtt{t}$ holds iff context $c \in \mathcal{C}$ is supported by the current configuration,
- $s(f) = \mathtt{f}$ holds iff feature $f \in \mathcal{F}$ is inactive in the current configuration, and
- $s(c) = \mathtt{f}$ holds iff context $c \in \mathcal{C}$ is incompatible with the current configuration.

In case of (ii) a state s further denotes if features and contexts are arbitrarily reconfigurable and, thereby, if they are irrelevant for the remaining configuration in s, i.e.,

- $s(f) = \bot_U$ holds iff feature $f \in \mathcal{F}$ is irrelevant for the current configuration and
- $s(c) = \bot_U$ holds iff context $c \in \mathcal{C}$ is irrelevant for the current configuration.

Each path of consecutive transitions corresponds to possible sequences of reconfigurations resulting from changes in the contextual situation of the device.

6.4.1 *Reconfiguration Transitions*

To enable reconfigurations, which are based on a $\mathcal{PKS}$, specific reconfiguration semantics are necessary. The reconfiguration from a source state $s \in \mathcal{S}$ to a target state $s' \in \mathcal{S}$ corresponds to a change in the interpretation of features and contexts. Thereupon, the target state s' may also refine features that are unrestricted in s.

The first property (1) of a reconfiguration transition $s \longrightarrow s'$, with $\{s, s'\} \subseteq \mathcal{S}$, states that potentially three kinds of changes are imposed onto a current state s during a reconfiguration, i.e.,

(i) a subset of interpreted contexts/features in s becomes unrestricted in s',

(ii) a subset of unrestricted contexts/features in s is either active or inactive in s', and

(iii) the interpretation of some contexts/features is changed in s'.

The second property (2) requires that every possible target state s' is either refineable $\gamma \sqsubseteq s'$ to a complete configuration $\gamma \in \hat{\Gamma}_{cfm}$ that is valid w.r.t. the context-feature model *cfm* or corresponds to a complete valid configuration $\gamma = s'$ in the first place. Thus, every reconfiguration transition is executed either to a valid complete state, to a valid partial state, or to an error state.

If the target state s' of a reconfiguration is a partial state, the state is *always arbitrarily* refineable to a complete configuration $\gamma \in \hat{\Gamma}_{cfm}$. For instance, let us assume that for the contextual situation {`Home`} `VoIP` is the only unrestricted feature. This configuration has to be refined if the context `Slow Network` becomes active. In this case, `VoIP` has to be inactive in the target configuration for the contextual situation {`Slow Network`, `Home`}. This second property is based on the characteristics of a partial state, i.e., that a feature and context may only be unrestricted if they are irrelevant for the remaining configuration, and that no state in the state space contradicts the constraints imposed by the context-feature formula *cfm*.

In the following, this concept of a reconfiguration transition is applied to the definition of a $\mathcal{PKS}$, in which each state $s \in \mathcal{S}$ corresponds to a valid (partial or complete) configuration $\gamma \in \Gamma_{\mathtt{cfm}}$.

Definition 6.5 (*Reconfiguration $\mathcal{PKS}$*)**.** Let *cfm* be a context-feature model. Each reconfiguration $s \longrightarrow s'$ in $\mathcal{PKS}$ is characterized by the following two properties.

(1) Let $\mathcal{C}' \cup \mathcal{F}'$ be a set of contexts/features that are to be reconfigured, i.e., $\mathcal{C}' \cup \mathcal{F}' = \{cf \in \mathcal{C} \cup \mathcal{F} \mid s(cf) \neq s'(cf)\}$, such that

$$\begin{aligned} & (\forall cf' \in \mathcal{F}' \cup \mathcal{C}' : s(cf') \in \mathbb{B} \Rightarrow s'(cf') = \bot_{\mathrm{U}}) && \text{(i)} \\ \vee\ & (\forall cf' \in \mathcal{F}' \cup \mathcal{C}' : s(cf') = \bot_{\mathrm{U}} \Rightarrow s'(cf') \in \mathbb{B}) && \text{(ii)} \\ \vee\ & (\forall cf' \in \mathcal{F}' \cup \mathcal{C}' : s(cf') \in \mathbb{B} \Rightarrow s'(cf') \in \mathbb{B}) && \text{(iii)} \end{aligned}$$

(2) $\forall s' \in \mathcal{S}\ :\ \exists \gamma \in \hat{\Gamma}_{cfm}$ such that $\gamma \sqsubseteq s'$ or $\gamma = s'$ holds.

Since unrestricted features are arbitrarily configurable, the configuration of a feature that is unrestricted in the target state may be inherited from the source state. For example, let us assume that the device is in a state, which is suitable for the contextual situation {`Home`, `Slow Network`}. If the context `Slow Network` is left, the feature `VoIP` becomes unrestricted according to the context `Home`. Therefore, a reconfiguration of `VoIP` is not necessary and it may stay inactive.

Reconfigurations are caused by changes in the contextual situation $C^+ \subseteq \mathcal{C}$ of a device (c.f. Notation 6.1). Whenever the contextual situation changes the device emits a corresponding event $\langle \chi\, c \rangle$, $\chi \in \{\oplus, \ominus\}$ to denote the change, where $\langle \oplus \mathrm{c} \rangle$ denotes context $c \in \mathcal{C}$ to become active. If a single context becomes active, it is part of the contextual situation of a device, i.e., $c \in C^+$. In contrast to that, $\langle \ominus \mathrm{c} \rangle$ denotes context $c \in \mathcal{C}$ to be left, i.e., to become inactive. In this case, the context is not part of the contextual situation anymore, i.e., $c \notin \mathcal{C}$. The resulting contextual situation potentially requires a feature configuration that is currently not supported by the active state. Therefore, a change in the feature configuration is necessary via appropriate transitions to a target state, in which the active contexts $c \in C^+$ are either active or at least unrestricted.

Notation 6.3 (Context Events as Reconfiguration Trigger)**.** A context event $\langle \chi\, c \rangle$ adapts the contextual situation of a device $C^+ \subseteq \mathcal{C}$ as follows

- if $\chi = \oplus$ the context c is added to the contextual situation $C^+ \cup c$, or
- if $\chi = \ominus$ the context c is removed from the contextual situation $C^+ \backslash c$

Note that for the latter case, i.e., $\chi = \ominus$, a reconfiguration is not necessarily being triggered. The reason for this is the restriction of contextual constraints in a context-feature model defined in Definition 4.18. The constraints between contexts and features are limited to dependency relations between contexts and features, e.g., `Home` *requires* `WLAN`, and incompatibility relations, e.g., `Office` *excludes* `Game Hub`. A require relation *from* a feature *to* a context, e.g., `WLAN` *requires* `Home`, is *not* allowed. This restriction seems natural for an SAS since the contextual situation dictates the reconfiguration of a system and not the other way around. Thus, if a context $c \in \mathcal{C}$ is left, i.e., $\langle \ominus c \rangle$ occurs, the requirements imposed by the contextual situation C^+ are relaxed. Therefore, a reconfiguration to a new target state $s' \in \mathcal{S}$ is not required because the current state still satisfies C^+.

Whenever a context event is emitted, a (partial) target state $s' \in \mathcal{S}$ has to be identified that satisfies the requirements of the resulting contextual situation. Furthermore, some configuration $\gamma \in \hat{\Gamma}_{cfm}$ is required, which refines the target state s' to a *complete* configuration denoting a valid interpretation of *every* feature and context. This refined configuration γ is applicable to *execute* the reconfiguration at runtime since it *fully* encodes the configuration state of a device at runtime. Note that with the properties of a partial state, such a configuration γ is derivable without any computational efforts since features and contexts, which are unrestricted in s', may be interpreted arbitrarily.

To denote such reconfigurations between (partial) states $s \in \mathcal{S}$ and complete configurations $\gamma \in \Gamma_{cfm}$ refining a state $\gamma \sqsubseteq s$, a transition system is required consisting of a set of states $\mathcal{Z} \subseteq \mathcal{S} \times \Gamma_{cfm}$, with $z \in \mathcal{Z} = (s, \gamma)$. Furthermore, transitions $\xrightarrow{\langle \chi c \rangle} \subseteq \mathcal{Z} \times (\{\oplus, \ominus\} \times \mathcal{C}) \times \mathcal{Z}$ are used to specify a reconfiguration between states $\{(s,\gamma)\} \in \mathcal{Z}$. Those transitions are labeled with events $\langle \chi c \rangle$, with $c \in \mathcal{C}$, to denote a contextual change, which triggers the transition.

Definition 6.6 (*Context Reconfiguration Transition System*)**.** Given a transition system encoding (partial) states $s \in \mathcal{S}$ with some complete configuration $\gamma \in \hat{\Gamma}_{cfm}$ refining s and a set of transitions triggered by a contextual change event $\langle \chi c \rangle$, i.e., a transition system with

- a set of states $\mathcal{Z} \subseteq \mathcal{S} \times \hat{\Gamma}_{cfm}$, with $z = (s, \gamma)$ such that $\gamma \sqsubseteq s$ holds and
- a set of transitions $\xrightarrow{\langle \chi c \rangle} \subseteq \mathcal{Z} \times (\{\oplus, \ominus\} \times \mathcal{C}) \times \mathcal{Z}$.

For example, let us assume that the contextual situation C^+={`Home`} is currently active. Furthermore, the feature `VoIP` is active in γ and unrestricted in the source state s. If the event $\langle \oplus$`Slow Network`$\rangle$ is emitted, a target state s' has to be found, which is suitable for the contextual situation C^+={`Home`, `Slow Network`}. According to the context-feature model depicted in Figure 4.2 the reconfiguration $(s, \gamma) \xrightarrow{\langle \oplus \texttt{Slow Network} \rangle} (s', \gamma')$ has to result in a target state s' in which `VoIP` is inactive. Since $\gamma' \sqsubseteq s'$ has to hold, `VoIP` has to be reconfigured from currently active in γ to inactive in γ'.

A first property (1) of such contextual reconfigurations $(s, \gamma) \xrightarrow{\langle \chi c \rangle} (s', \gamma')$ is that the interpretation of contexts and features $cf \in \mathcal{C} \cup \mathcal{F}$ in the source configuration γ do not have to be adapted if $s'(cf) = \bot_U$ holds, i.e., the respective context or feature is unrestricted in s'. Otherwise, cf has to be reconfigured according to the target state s', i.e., $\gamma' = s'(cf)$.

A second property (2) is that the target state s' has to be suitable for the contextual situation C^+_{next} emerging from the contextual change$\langle \chi c \rangle$. Thus, for a target state s' to be suitable for C^+_{next} every context $c \in C^+_{next}$ of the contextual situation has to be either unrestricted or active in s'. Revisiting the previous example, after the change event $s \xrightarrow{\langle \oplus \texttt{Slow Network} \rangle} s'$ is emitted, s'(`Slow Network`) $\geqslant \bot_U$ and s'(`Home`) $\geqslant \bot_U$ has to hold in the target state s'.

For SAS is reasonable to assume that the states are *fully path-connected*, i.e., every state is reachable eventually from some source state via a *sequence* of consecutive reconfiguration transitions. For example, for a reconfiguration from a `VoIP` call to a `Cellular` call a protocol hand-over has to be executed via multiple transitions. More precisely, the reconfiguration starts in a configuration, in which `VoIP` is active and `Cellular` is inactive, followed by a configuration, in which `VoIP` and `Cellular` are both active to execute the hand-over, to result in a configuration in which `VoIP` is inactive and `Cellular` is active. To denote such a *reconfiguration path* from a source state s_0 to a target state s_n triggered by a *single* contextual event $\langle \chi c \rangle$, I write $s_0 \xRightarrow{\langle \chi c \rangle} s_n$ for short.

Definition 6.7 (*Contextual Reconfiguration*)**.** Given a context reconfiguration transition system $\mathcal{Z}$ with a set of (partial) states $s \in \mathcal{S}$ and some complete configuration $\gamma \in \hat{\Gamma}_{cfm}$ refining every state s. Furthermore a set of transitions is given which are triggered by a contextual change event $\langle \chi c \rangle$.

Let $(s,\gamma) \xrightarrow{\langle \chi c \rangle} (s',\gamma')$ be a reconfiguration for a contextual change event $\langle \chi c \rangle$ triggered by the emerging contextual situation C^{+}_{next}.

(1) A reconfiguration $(s,\gamma) \xrightarrow{\langle \chi c \rangle} (s',\gamma')$ potentially enforces a currently active configuration γ to be adapted to a subsequent configuration γ' such that for each context and feature $cf \in \mathcal{C} \cup \mathcal{F}$ it holds

$$\gamma'(cf) = \begin{cases} \gamma(cf) & \text{if } s'(cf) = \bot_U \\ s'(cf) & \text{else.} \end{cases}$$

(2) For a reconfiguration to be valid, the target state s' has to be suitable for the contextual situation C^{+}_{next}, i.e.,

$$\forall c \in C^{+}_{next} \; : \; s'(c) \geqslant \bot_U$$

has to hold, with C^{+}_{next} denoting the contextual situation *after* $\langle \chi c \rangle$ has been emitted.

Some finite sequence of consecutive reconfigurations, i.e., a *reconfiguration path*, leading from a source state s_0 to a target state s_n for a *single* contextual change $\langle \chi c \rangle$ is denoted as

$$s_0 \xRightarrow{\langle \chi c \rangle} s_n = s_0 \to s_1 \to \cdots \to s_n$$

for short, with $s_i \in \mathcal{S}$ and $\{i, n\} \in \mathbb{N}$.

Note that if $s = s'$ holds for a contextual change, the respective reconfiguration corresponds to a self-transition. Hence, a reconfiguration of a feature is *not* necessary if either the interpretation of features are the same in source and target state or features that have to be reconfigured are unrestricted in the target state.

The characteristics of a reconfiguration as introduced in the previous definitions still apply for every single reconfiguration on a path $s_0 \xRightarrow{\langle \chi c \rangle} s_n$ between a source state s_0 and target state s_n.

Example 6.8 (*Reconfiguration of Partial States*).

Figure 6.15 depicts the reconfiguration possibilities for the partial state s. The context `Home` is active, `Office` is inactive and `Slow Network` is unrestricted. Thus, if the device enters the context `Slow Network` $\langle \oplus$`Slow Network`$\rangle$, or the device leaves the context `Slow Network`, the partial state has not to be reconfigured, i.e., a self-transition is executed. The same holds if the device emits an event that `Office` is left $\langle \ominus$`Office`$\rangle$ and `Home` is entered $\langle \oplus$`Home`$\rangle$. For all these contextual changes, none of the features have to be reconfigured.

If the context {`Office`} becomes active $\langle \oplus$`Office`$\rangle$ in addition to the currently active context `Home`, another state has to be found that satisfies the requirements of the new contextual situation {`Home`, `Office`}. A reconfiguration may be executed to every target state, in which `Home` and `Office` are either unre-

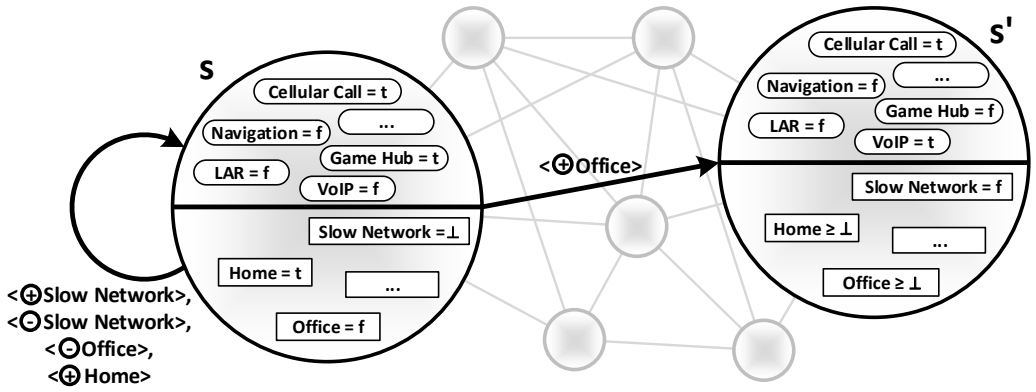


Figure 6.15: Reconfigurations in a Partial Kripke Structure

stricted or active, i.e., $s(\texttt{Home}) \geqslant \bot_U$ and $s(\texttt{Office}) \geqslant \bot_U$ holds. In this example, the state s does not satisfy this contextual situation because `Game Hub` is active, which contradicts the context `Office`. However, the state s' satisfies the contextual situation {`Home`, `Office`}. Thus, if $\langle \oplus \texttt{Office} \rangle$ is emitted the transition $s \xrightarrow{\langle \oplus \texttt{Office} \rangle} s'$ is executed. With this transition not only the contextual situation changes but also the interpretation of features is adapted, e.g., `Game Hub` is reconfigured to be inactive.

To ensure soundness of the concept of a DSPL reconfiguration transition system based on three-valued logics, the next section discusses fundamental correctness properties of the proposed formal reconfiguration framework.

6.4.2 *Correctness of the Reconfiguration Transition System*

For the approach to be correct it is required that the approach executes appropriate valid reconfigurations, in case they exist, for every possible sequence of context changes potentially emerging at runtime starting in some state $s_0 \in \mathcal{S}$. Therefore, the following lemma shows that reconfigurations of a $\mathcal{PKS}$ lead to a valid configuration state with respect to a context-feature model *cfm*.

Lemma 6.1. Let $(s, \gamma) \xRightarrow{\langle \chi c \rangle} (s', \gamma')$ be a reconfiguration of a $\mathcal{PKS}$ for *cfm*. From $s \in \Gamma_{cfm}$ follows that $s' \in \Gamma_{cfm}$.

Proof. Follows from the second property (2) of Definition 6.5, i.e., that a target state s' is either refineable to any of the complete configurations $\gamma' \in \hat{\Gamma}_{cfm}$, or corresponds to a complete configuration $\gamma' \in \hat{\Gamma}_{cfm}$ in the first place. □

As a consequence, all possible sequences of reconfiguration paths for a set of consecutive contextual change events

$$(s_0, \gamma_0) \xRightarrow{\langle \chi_0 c_0 \rangle} (s_1, \gamma_1) \xRightarrow{\langle \chi_1 c_1 \rangle} \cdots \xRightarrow{\langle \chi_{k-1} c_{k-1} \rangle} (s_k, \gamma_k)$$

starting in a state $s_0 \in \Gamma_{cfm}$ that is valid w.r.t. context-feature model *cfm* always (inductively) lead to valid subsequent (partial) states $s_i \in \hat{\Gamma}_{cfm}$ and complete configurations $\gamma_i \in \hat{\Gamma}_{cfm}$, $1 \leqslant i \leqslant k$.

Those sequences are caused by consecutive contextual change events $\langle \chi_i \, c_i \rangle$, i.e., starting from a set $C_0^+ \subseteq \mathcal{C}$ denoting the initially active contextual situation, subsequent contextual situations are given as

$$C_{i+1}^+ = \begin{cases} C_i^+ \cup \{c_i\} & \text{if } \chi_i = \oplus, \\ C_i^+ \setminus \{c_i\} & \text{if } \chi_i = \ominus, \end{cases}$$

where $1 \leqslant i \leqslant k$.

For the semantics of a $\mathcal{PKS}$ to yield *sound* reconfigurations it is required that every state $s_{i+1} \in \mathcal{S}$ reached after i contextual changes satisfies the requirement of the corresponding combination of contexts in the target contextual situation $C_{i+1}^+ \in \mathbb{C}_{i+1}$. In this case, $\mathbb{C}_{i+1}$ denotes the set of all valid context combinations for a target state s_{i+1} w.r.t a given context-feature model *cfm* (c.f. Definition 6.2 of valid context combinations $C_i^+ \in \mathbb{C}_{cfm}$ for a state $s_i \in \mathcal{S}$).

Theorem 6.1 (Soundness)**.** Let $(s_i, \gamma) \xRightarrow{\langle \chi_i c_i \rangle} (s_{i+1}, \gamma_{i+1})$ be a reconfiguration of a $\mathcal{PKS}$ for *cfm*. From $s_i \in \Gamma_{cfm}$ and $C_i^+ \in \mathbb{C}_i$ it follows that $s_{i+1} \in \Gamma_{cfm}$ and $C_{i+1}^+ \in \mathbb{C}_{i+1}$.

Proof. Follows Lemma 6.1 and the second property (2) of Definition 6.7, i.e., that every context of a contextual situation has to be either unrestricted or active in the target state of a reconfiguration. □

Thus, every reconfiguration triggered by a change in the contextual situation $\langle \chi_i c_i \rangle$ results in a (path of consecutive) transitions $s_i \Rightarrow s_{i+1}$ from a source state (s_i, γ) to a target state (s_{i+1}, γ_{i+1}) iff the combination of contexts describing the target contextual situation is satisfiable by the constraints specified in *cfm*.

Concerning the *completeness* of the approach, note that not every contextual situation $C^+ \subseteq \mathcal{C}$ potentially arising at runtime is necessarily supported by a given *cfm* due to (i) explicit constraints τ among contexts and (ii) implicit constraints θ caused by conflicting context-feature requirements (c.f. Definition 4.18 of contextual constraints). Thus, my approach is *incomplete* w.r.t. the set of *all* potentially emerging contextual situations $\mathbb{C}$. Due to the constraints imposed by a *cfm* my approach is only capable to cover all *valid* contextual situations $\mathbb{C}_{cfm}$ (c.f. Definition 6.3) emerging at runtime. Even if the state space $\mathcal{S}$ does not provide a suitable state $s \in \mathcal{S}$ for a contextual situation $C^+ \in \mathbb{C}_{cfm}$ emerging at runtime, such a state s is dynamically derivable by executing a constraint solver at runtime. Thus, for contextual situations $\mathbb{C}_{cfm}$ that are *valid* w.r.t. *cfm* my approach is *complete*.

As shown in the previous Theorem 6.1, the reconfiguration semantics for $\mathcal{PKS}$ is sound. Although, this theorem restricts the amount of reconfigurations to those reconfigurations, which yield valid configuration states, the reconfiguration semantics for $\mathcal{PKS}$ is complete w.r.t. all valid combination possibilities of the con-

texts in $\mathbb{C}_{cfm}$ (c.f. Definition 6.3 of valid combination of contexts). In this regard, the following theorem shows that a reconfiguration $\mathcal{PKS}$ is complete if a reconfiguration to a valid target state is *always* triggerable by a contextual change iff emerging contextual situations are valid w.r.t. *cfm*, i.e., $C^+ \in \mathbb{C}_{cfm}$ holds.

Theorem 6.2 (Completeness)**.** Let $\langle \chi_i \, c_i \rangle$ be a change from a valid contextual situation $C_i^+ \in \mathbb{C}_{cfm}$ to a valid contextual situation $C_{i+1}^+ \in \mathbb{C}_{cfm}$. Then there exists a reconfiguration $(s_i, \gamma_i) \xRightarrow{\langle \chi_i c_i \rangle} (s_{i+1}, \gamma_{i+1})$ such that

- $s_{i+1} \in \Gamma_{cfm}$,
- $\gamma_{i+1} \in \hat{\Gamma}_{cfm}$,
- $\gamma_{i+1} \sqsubseteq s_{i+1}$, and
- $\forall c \in C_{i+1}^+ \, : \, s_{i+1}(c) \geqslant \bot$

hold.

Proof. Proving *completeness* requires

(i) that for every potential complete configuration $\gamma \in \hat{\Gamma}_{cfm}$ there is a (partial) state $s \in \mathcal{S}$ such that $\gamma \sqsubseteq s$ holds and

(ii) that the states $\mathcal{S}$ of $\mathcal{PKS}$ are *fully path-connected*.

According to Definition 6.3, from $C_i^+ \in \mathbb{C}_{cfm}$ and $C_{i+1}^+ \in \mathbb{C}_{cfm}$ it follows that corresponding states $s_i \in \Gamma_{cfm}$ and $s_{i+1} \in \Gamma_{cfm}$ exist, in which $\forall c \in C_{i+1} \, : \, s_{i+1}(c) \geqslant \bot$ holds if requirement (i) is satisfied. Provided the $\mathcal{PKS}$ satisfies requirement (ii), at least one reconfiguration sequence obeying the constraints of a reconfiguration $\mathcal{PKS}$ as required by Definition 6.5 exists leading from (s_i, γ_i) to (s_{i+1}, γ_{i+1}). Such a reconfiguration *always* satisfies Definition 6.7 of a reconfiguration if the change in the contextual situation (c.f. Notation 6.3) triggered by a contextual change event results in a valid contextual situation $C_{i+1}^+ \in \mathbb{C}_{cfm}$. □

The concept of a partial Kripke Structure $\mathcal{PKS}$ is used to specify the reconfiguration behavior of a DSPL. This approach is summarizable as follows

- the reconfiguration behavior of a context-aware DSPL is specified as a $\mathcal{PKS}$ on the basis of a context-feature model *cfm*,
- the states $\mathcal{S}$ of a $\mathcal{PKS}$ correspond to a set of valid partial and complete configurations $\mathcal{S} \subseteq \Gamma_{cfm}$ of a *cfm*,
- a (partial) state $s \in \mathcal{S}$ is arbitrarily refinable $\gamma \sqsubseteq s$ to a complete configuration $\gamma \in \Gamma_{cfm}$ *without* the need to execute a constraint solver,
- a single transition $s \xrightarrow{\langle \chi c \rangle} s'$ is triggered by entering $\langle \oplus c \rangle$ or leaving $\langle \ominus c \rangle$ a context,

- if $\langle \oplus c \rangle$ is emitted (i) c is added to the contextual situation C^+ and (ii) it must hold that $\forall c \in C^+ : s'(c) \geqslant \perp_U$ in the target state s',
- if $\langle \ominus c \rangle$ is emitted c is removed from the contextual situation C^+, and
- any sequence of potential context changes $\langle \chi_0 c_0 \rangle$, $\langle \chi_1 c_1 \rangle, \cdots, \langle \chi_n c_n \rangle$ supported by a *cfm* results in the appropriate sequence of reconfiguration transitions $s_0 \xrightarrow{\langle \chi_0 c_0 \rangle} s_1, s_1 \xrightarrow{\langle \chi_1 c_1 \rangle} s_2, \cdots, s_{n-1} \xrightarrow{\langle \chi_n c_n \rangle} s_0$ in the $\mathcal{PKS}$.

Thus, for a context-feature model *cfm*, a corresponding $\mathcal{PKS}$ is derivable that specifies reconfigurations denoting changes in the contextual situation of a device, which are supported by the DSPL. Either such reconfigurations already exists in a pre-planned $\mathcal{PKS}$, or, in case of an incompletely pre-planned state space, an appropriate reconfiguration definitely exists and, therefore, may be dynamically added to $\mathcal{PKS}$ at runtime. In the next section, a construction algorithm for such a $\mathcal{PKS}$ is presented, incorporating trade-offs between a pre-planned reconfigurations and runtime evolutions of the deployed $\mathcal{PKS}$.

6.5 IMPLEMENTATION OF A CONTEXT-AWARE RECONFIGURATION PROCESS

This section describes DSPL design and implementation strategies based on context-aware reconfiguration planning using a $\mathcal{PKS}$ specification as previously introduced. Based on a context-feature model *cfm* an appropriate preparation of a $\mathcal{PKS}$ is guided by criteria for

(1) pre-planning a subset from all valid configurations $\hat{\Gamma}_{cfm}$ in *cfm* and deriving the corresponding set of states $\mathcal{S} \subseteq \hat{\Gamma}_{cfm}$ of configuration states suitably covering contextual situations potentially emerging at runtime and

(2) computing partial states $s \subseteq \check{\Gamma}_{cfm}$ to maximize reconfiguration flexibility and minimize memory consumption at runtime w.r.t. contextual situations, which are not pre-planned.

This section introduces two algorithms to achieve these two criteria. The first algorithm computes a $\mathcal{PKS}$ based on a context-coverage criterion and an upper limit for the number of unrestricted features in a state. The second algorithm handles the runtime adaptation of an SAS based on a pre-computed $\mathcal{PKS}$.

6.5.1 *Pre-Planning of a $\mathcal{PKS}$*

For (1), the pre-computation of a $\mathcal{PKS}$ state-transition system, the $\mathcal{PKS}$ provides the means for a trade-off between pre-computation of an incomplete state space at design-time and on-demand computation at runtime, which is discussed in the following.

At runtime, reconfigurations are enforced by contextual changes in the contextual situation of a device C^+. Thus, appropriate context-aware criteria for tailoring incomplete state spaces that suitably cover contextual situations potentially emerging at runtime are derivable by considering contexts c from a set of available

contexts $c \in \mathcal{C}$. Further, contexts $c \in \mathcal{C}$ may emerge in a combinatorial manner in such a way that a set of contexts $c \in C^+ \subseteq \mathcal{C}$ is active at the same time describing a contextual situation. Corresponding criteria select subsets $\mathbb{C}^k \subseteq \mathbb{C}$ of context combinations covering any k-wise combination of contexts. If such a combination combination $C_i^+ \in \mathbb{C}^k$ is compatible with the context-feature model *cfm* a corresponding state $s_i \in \mathcal{S}$ is derived and integrated into the transition system.

For (2), the parameter $0 \leqslant l \leqslant |\mathcal{F} \cup \mathcal{C}|$ denotes the maximum number of features $\mathcal{F}$ and/or contexts $\mathcal{C}$ concurrently set to unrestricted ($\perp_U$) during the identification of partial states. Therefore, for each complete configuration $\gamma_i \in \hat{\Gamma}_{cfm}$ that corresponds to a state $s_i \in \mathcal{S}$ obtained in step (1), a set of configurations $\check{\Gamma}^{(i,l)} \subseteq \check{\Gamma}_{cfm}$ is computed containing exactly those partial configurations $\gamma \in \check{\Gamma}^{(i,l)}$

(i) with at most l parameters being set to $\perp_U$ and

(ii) in which for any possible configuration refinement $\gamma' \in \hat{\Gamma}_{cfm}$ of the partial configuration γ with $\gamma' \sqsubseteq \gamma$, it holds that γ' is a complete runtime configuration $\gamma' \in \hat{\Gamma}_{cfm}$.

Hence, each of the 2^l possible interpretation combinations of the l unrestricted contexts/features in γ result in a valid configuration. As a consequence, all states corresponding to configurations γ' being refinement of some partial state in the set $\Gamma^{(i,l)} \subseteq \check{\Gamma}_{cfm}$ are removable from the $\mathcal{PKS}$ since they are abstracted by the corresponding partial state.

Algorithm 2 summarizes the pre-computation of a $\mathcal{PKS}$ for a given *cfm* and context-combination parameter k and the parameter for the upper limit of unrestricted contexts/features l. According to the Definition 6.4 of a $\mathcal{PKS}$, the algorithm results in an $\mathcal{PKS}$ state-transition system.

INCOMPLETE STATE SPACE COMPUTATION. To derive an incomplete state space it is required that $k > 0$, i.e., at least for every single context $c \in \mathcal{C}$ a corresponding configuration is pre-computed that satisfies the requirements imposed by c in *cfm*. In the lines 5–10, for the set $C^+ \subseteq \mathbb{C}^k$ of context combinations, corresponding complete configurations $\gamma \in \hat{\Gamma}_{cfm}$ are computed, e.g., by executing a constraint solver[1]. If there is no state in the state space $\mathcal{S}$ that covers the context combination C^+ a new configuration is derived (line 6). If a complete configuration $\gamma \in \hat{\Gamma}_{cfm}$ for the context-combination C^+ is found, the configuration state s for γ is added to the set of states $\mathcal{S}$ (lines 7–8).

The computation of a representative configuration γ for a contextual combination C^+ in line 7 abstracts from details such as the selection of a representative state. Multiple selection strategies of a configuration, which satisfies C^+ are applicable here, e.g., (i) a random selection, (ii) the configuration, in which the least amount of features are active, or (iii) optimizing non-functional attributes such as costs or stability.

PARTIAL STATE ABSTRACTION. For an abstraction of a state to a partial state a parameter of $l = 0$ is permitted. However, in this case, no partial states are computed. In case of $l > 0$, the set $\check{\Gamma}^{(l,i)}$ of partial configurations is computed for every

[1] We used SAT4J [BP10] as constraint solver in our implementation.

Algorithm 2 $\mathcal{PKS}$ Pre-Computation

1: **Input:** $cfm \in \mathcal{FM}_{(\mathcal{F} \cup \mathcal{C})}$; $k, l \in \mathbb{N}$; $1 \leqslant k \leqslant |\mathcal{C}|$; $0 \leqslant l \leqslant |\mathcal{F} \cup \mathcal{C}|$
2: **Output:** $\mathcal{PKS} = (\mathcal{P}, \mathcal{S}, \longrightarrow)$
3: **Init:** $\Gamma_{cfm} := (\mathcal{F} \cup \mathcal{C}) \to \mathbb{B}_\perp$; $\mathcal{S} := \emptyset$; $\longrightarrow := \emptyset$;

4: *// Incomplete State Space Computation*
5: **for all** $C^+ \in \mathbb{C}^k$ **do**
6: **if** $\nexists s_i \in \mathcal{S} : C^+ \in \mathbb{C}_i$ **then**
7: **compute** $\gamma \in \hat{\Gamma}_{cfm}$ **where** $\forall c \in C^+ : \gamma(c) = t$
8: $\mathcal{S} := \mathcal{S} \cup \{s\}$ **where** $s = \gamma$ holds
9: **end if**
10: **end for**
11: *// Partial State Abstraction*
12: **for all** $s_i \in \mathcal{S}$ **do**
13: **compute** $\check{\Gamma}^{(i,l)}$
14: $\mathcal{S} := \mathcal{S} \cup \check{\Gamma}^{(i,l)}$
15: **end for**
16: $\overline{\Gamma} := \{\gamma' \in \hat{\Gamma}_{cfm} \mid \exists \gamma' \in \mathcal{S}, \gamma' \neq s \in \mathcal{S} : \gamma' \sqsubseteq s\}$
17: $\mathcal{S} := \mathcal{S} \setminus \overline{\Gamma}$
18: *// Add Transition Relation*
19: **for all** $(s, s') \in \mathcal{S}, s' \neq s$ **do**
20: **if** $s \longrightarrow s'$ is valid transition **then**
21: $\longrightarrow := \longrightarrow \cup \{s \longrightarrow s'\}$
22: **end if**
23: **end for**
24: **return** $\mathcal{PKS}$

state $s_i \in \mathcal{S}$ previously computed in lines 5–10 as described above. These partial configurations are added to the set of states $\mathcal{S}$ if they correspond to a partial state (line 12–15), i.e., if $s(f) = \perp_U$ or $s(c) = \perp_U$ holds for some features $f \in \mathcal{F}$ and contexts $c \in \mathcal{C}$, those features and contexts have to be arbitrarily reconfigurable in combination. Thereupon, the set $\overline{\Gamma}$ of redundant states, i.e., complete configuration states being subsumed by some newly added partial states, are removed from the set of states (line 16–17).

The computation of unrestricted features in line 13 is executable based on three different strategies, as previously introduced in Section 6.3.2, i.e., unrestricted features are identifiable via

- randomized variable ordering,
- heuristical variable ordering, or
- genetic variable ordering.

Note that the parameter l represents an upper-limit for those strategies. If at least l unrestricted contexts or features are identified for a certain partial state, the identification of further unrestricted contexts/features is finished. Thus, a set of partial states is being generated for a given configuration $\gamma \in \hat{\Gamma}cfm$. Algorithm 2

abstracts here from further details regarding the identification of unrestricted contexts/features and selection of partial states. Similar to the selection of representative configuration states in line 7 the partial states may also be systematically selected during their computation. A partial state may be chosen based on (i) the likelihood of an unrestricted feature to be reconfigured at runtime, (ii) the amount of unrestricted features, or (iii) non-functional aspects, such as costs or stability, of active/inactive features.

Finally, transitions $\longrightarrow$ are added between any pair of remaining states that are considered to be valid reconfigurations 19–23. Thus, the states $\mathcal{S}$ either become *fully connected* or the reconfigurability is restricted and the states $\mathcal{S}$ become *fully path-connected*, e.g., some reconfigurations between states are prohibited by the developer or by certain hardware characteristics of a device.

Thus, the derived incomplete $\mathcal{PKS}$ containing partial states provides the necessary means to reconfigure an SAS at runtime in a resource efficient and flexible manner. The next section discusses how an SAS is reconfigured based on such a $\mathcal{PKS}$ w.r.t. changes in the contextual situation.

6.5.2 *Runtime Reconfiguration based on a $\mathcal{PKS}$*

The DSPL-based reconfiguration process of an SAS at runtime is handled by Algorithm 3. The algorithm outlines the resulting context-aware reconfiguration process of an SAS at runtime on the basis of a $\mathcal{PKS}$ specification that is pre-computed by Algorithm 2. Starting with an initial configuration $\gamma_0 \in \hat{\Gamma}_{cfm}$ and a start state $s_0 \in \mathcal{S}$ such that $\gamma_0 \sqsubseteq s_0$, the device configuration $\gamma \in \hat{\Gamma}_{cfm}$ is continuously adapted corresponding to the active state $s \in \mathcal{S}$ of the reconfiguration $\mathcal{PKS}$.

Reconfiguration transitions $\Rightarrow$ are potentially released in the control loop (line 3–24) due to the instantaneous occurrence of context changes, such as leaving $\langle \ominus c \rangle$ or entering $\langle \oplus c \rangle$ of a context c. Such changes in the contextual situation are denoted as $\langle \chi c \rangle$ (line 4). The set of the currently active context combination C^+ describing the contextual situation, is adjusted with every contextual change $\langle \chi c \rangle$ in line 5. A reconfiguration becomes necessary for a context change $\langle \oplus / \ominus c \rangle$ if the current state does not support the requirements of the resulting context combination (line 6), where two cases arise.

(1) The $\mathcal{PKS}$ contains at least one reconfiguration transition to an appropriate target state s' $s \stackrel{\langle \chi c \rangle}{\Longrightarrow} s'$ (line 7–8).

(2) The $\mathcal{PKS}$ contains neither an appropriate reconfiguration nor a suitable target state, which triggers an on-demand computation of an appropriate runtime configuration $\gamma' \in \hat{\Gamma}_{cfm}$ (line 10–25).

In the first case (1) more than one suitable target state is available and, therefore, multiple reconfigurations are executable. In such a case, multiple strategies are applicable to select an appropriate the target state for the reconfiguration, e.g., the state

- with the least change impact w.r.t. the currently active state,

Algorithm 3 Runtime Reconfiguration and $\mathcal{PKS}$ On-Demand Extension

Input: $cfm \in \mathcal{FM}_{(\mathcal{F} \cup \mathcal{C})}$; $\mathcal{PKS}$
Init: $s := s_0$; $\gamma := \gamma_0$; $C^+ := \emptyset$

loop
  await $\langle \chi c \rangle$
  $C^+ := C^+ \cup c$ if $\chi = \oplus$ **and** $C^+ := C^+ \setminus c$ if $\chi = \ominus$
  if $(\chi = \oplus$ && $s(c) < \bot_U) \,||\, (\chi = \ominus$ && $s(c) > \bot_U)$ **then**
    if $\exists\, s' \in \mathcal{S} : s \xRightarrow{\langle \chi c \rangle} s'$ **then**
      choose $s' \in \mathcal{S}$ **where** $s \xRightarrow{\langle \chi c \rangle} s'$
    else
      compute $s' := \gamma' \in \Gamma_{cfm}$ **where**
      $\gamma'(c) = \mathtt{t}$ if $\chi = \oplus$ **and** $\gamma'(c) = \mathtt{f}$ if $\chi = \ominus$
      and $\forall c' \in C^+, c' \neq c : \gamma'(c') \geqslant s(c')$
      if $\gamma' = \epsilon$ **then**
        $\mathcal{S} := \mathcal{S} \cup s_\epsilon$, $\longrightarrow := \longrightarrow \cup \{s \longrightarrow s_\epsilon\}$
      else
        $\mathcal{S} := \mathcal{S} \cup s'$
        for all $s'' \in \mathcal{S}, s'' \neq s'$ **do**
          if $s' \longrightarrow s''$ is valid transition **then**
            $\longrightarrow := \longrightarrow \cup \{s' \longrightarrow s''\}$
          end if
          if $s'' \longrightarrow s'$ is valid transition **then**
            $\longrightarrow := \longrightarrow \cup \{s'' \longrightarrow s'\}$
          end if
        end for
      end if
    end if
    $s := s'$
    for all $f \in \mathcal{F}$ **do**

$$\gamma(f) = \begin{cases} \gamma(f) & \textbf{if } s'(f) = \bot_U \\ s'(f) & \textbf{else} \end{cases}$$

    end for
  end if
end loop

- with the most unrestricted contexts/features, i.e., the most flexible state, or
- with the best non-functional runtime properties, e.g., the one with the least energy consumption.

In case of an on-demand computation (2) of a configuration γ' that is not yet part of the incomplete $\mathcal{PKS}$ state-transition system, the computation either

- *fails*, denoted by $\gamma' = s_\epsilon$, due to an unsatisfiable contextual situation C^+ w.r.t. *cfm* (line 14–15), or
- it succeeds and returns a new state s' and a corresponding transition to reach s' from the current state s (line 16–24).

To keep track of erroneous context combinations transition $s \longrightarrow s_\epsilon$ are added leading to a special error state $s_\epsilon \notin \Gamma_{cfm}$ (lines 13–14). In contrast to that, a suitable target state s' is derived and added to the set of states $\mathcal{S}$ (line 16) if for new contextual situation $C^+ \in \mathbb{C}_{cfm}$ holds. Furthermore, *valid* reconfiguration transitions, i.e., reconfigurations that are executable w.r.t. the characteristics of the device, are added leaving from s' to some state s'' and leading to s' from some state s'' (lines 17–24). Thereby, the newly added state s' becomes *fully path-connected* in $\mathcal{PKS}$.

If an appropriate reconfiguration $s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$ is identified, its target state s' becomes the new active state (line 27). The currently active configuration γ is adapted to the new configuration of the target state s' in line 28. Every feature that is configured differently in and is not an unrestricted feature in s' has to be reconfigured such that $\gamma \sqsubseteq s'$ holds.

Note that for a reconfiguration $s \overset{\langle \chi c \rangle}{\Longrightarrow} s'$ leading from a state s to a state s', both being pre-computed according to Algorithm 2, either

- $s = s'$ holds if s already supports $\langle \chi c \rangle$, or
- $s \Longrightarrow s'$ holds due to the fully path-connected transition system.

To provide further insights into the implementation an evaluation incorporating trade-offs between a pre-computed reconfigurations and runtime extension of the deployed $\mathcal{PKS}$ is presented in the next section.

6.6 Evaluation

I evaluate my concepts of an incomplete state space and partial states based on the parameters of a k-wise context combination and l-wise upper limitation of unrestricted contexts/features. The goal of the evaluation is to investigate the impact of k and l and the resulting trade-offs in the resource consumption at runtime. Therefore, I analyze the following three aspects

(1) the estimated memory required for deploying a $\mathcal{PKS}$ state space pre-computed during the planning phase,

(2) the estimated processing efforts for performing on-demand reconfigurations caused by simulated contextual changes emerging at runtime, and

(3) the identification of unrestricted contexts/features based on the variable ordering.

6.6.1 *DSPL Pre-Planning at Design Time*

The configuration state space of $\mathcal{PKS}$ depends on the corresponding context-feature model *cfm*. The potential state space reduction to an incomplete state space covering *k*-wise context combinations depends on the number of features and contexts and the complexity, i.e., the constraint density of *cfm* as well as the choice of *k*.

In order to investigate the benefits of an incomplete state space, the extent of the reduction of suitable states w.r.t. a context-coverage criterion *k* is evaluated. Further, the possibility of unrestricted contexts/features w.r.t. an upper limit *l* is evaluated.

REDUCTION OF A STATE SPACE SCALING ACROSS NUMBER OF FEATURES. Figure 6.16 depicts the reduction of a complete state space to an incomplete state space scaling across the number of features in a feature model *cfm* for a *k* of 1, 2, 3, and 6. As a baseline, the size of the *complete* state space is also included. The feature models are generated using the BeTTy-Framework[2] with a Cross-Tree-Constraint Ratio (CTCR) of 15%. For each measurement point, 30 feature models are generated. The results are averaged across these sets.

The feature models from the test data set are enriched to a context-feature model *cfm* with a context-to-feature ratio of $\frac{1}{4}$, i.e., for every four features there is one context. Further, each context may have up to two constraint relations to the set of features $\mathcal{F}$, i.e., require and exclude edges. In the following I refer to this ratio as the *Context-Cross-Tree Constraint Ration (context-CTCR)*. A context-CTCR of 2 is considered to be standard for this evaluation.

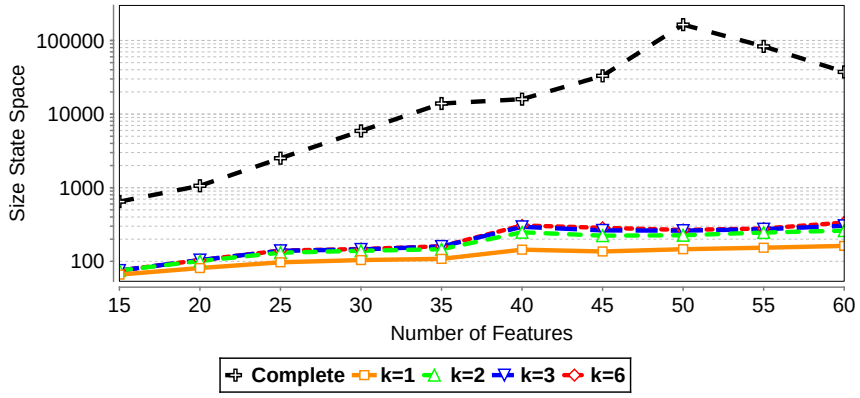


Figure 6.16: State Space Reduction Scaling across Number of Features

Figure 6.16 depicts a logarithmic vertical scaling across the size of the state space and a linear scaling across the number of features horizontally, i.e., starting with a feature model consisting of 15 features up to a feature model size of 60

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

features. Obviously, the state space generated for context combination $k = 1$ (covering single contexts) is the smallest and grows with increasing k.

The averaged maximum size of the complete state spaces corresponds to 105,000 states for a *cfm* with 50 features, whereas with $k = 6$ this state space is reduced significantly to 290 states, which corresponds to 0.2‰ of the original size. The difference between $k = 6$ and $k = 2$ is minimal, i.e., 290 states with $k = 6$ and 210 states with $k = 2$ for a *cfm* with 50 features. The size of a state space with $k = 6$ and $k = 3$ is nearly identical, e.g., 290 states with $k = 6$ and 285 states with $k = 3$. Overall it is safe to assume that the ratio, in which the size of the state space grows decreases with a greater k.

An interesting anomaly is the spike in the plot for the complete state spaces at a *cfm* with 50 features. Usually it is safe to assume that the size of the state space grows with the number of features since the size of the state space corresponds to $2^{|\mathcal{C} \cup \mathcal{F}|}$. However, this assumption neglects the individual constraints of a *cfm*. These constraints may lead to an anomaly that a *cfm* with less features may lead to more configuration states than a *cfm* with more features. For example, the generated *cfm* with 40 features leads to a larger complete state space in average than a *cfm* with 60 features in my evaluations.

The spike in the plot of the complete state spaces at a *cfm* with 50 features is independent from the reduced state spaces for $k = 1$, $k = 2$, $k = 3$, and $k = 6$. This implies that the state space increases while the amount of possible context combinations $\mathbb{C}^k_{cfm}$ remains stable.

REDUCTION OF A STATE SPACE SCALING ACROSS CTCR. Figure 6.17 depicts a plot that ranges across the context-CTCR horizontally with a fixed number of 35 features. The context-CTCR starts with 2, e.g., 1 excluding and 1 required feature per context, and is scaled up to 10. The size of the state space is depicted vertically using a linear scaling. As in the previous evaluation setup, the standard CTCR is set to 15%.

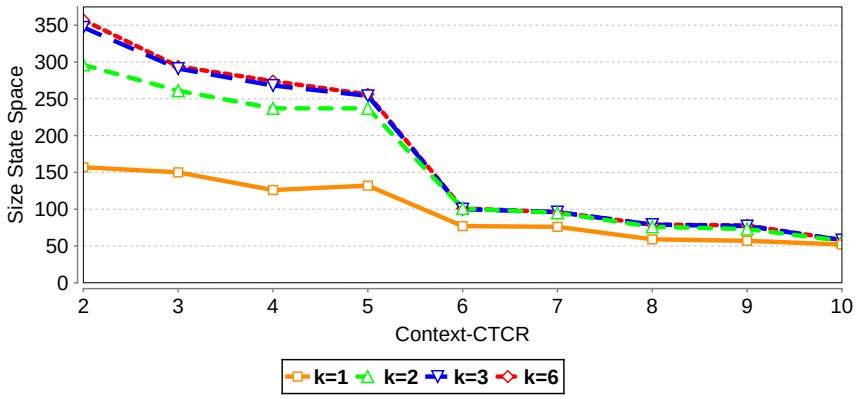


Figure 6.17: State Space Reduction Scaling across Context-CTCR

Figure 6.17 illustrates that the number of possible configurations decreases with increasing context-CTCR. Note that I left out the complete state space in this plot because the complete state space is not influenced by the context-CTCR and remains constantly at 105,000 states for this evaluation. As in the previous evalua-

tion, the most significant reduction of a state space is gained with $k = 1$ and the state space becomes larger with a higher k. However, the size of the state space continuously decreases with an increasing of the context-CTCR. For example, the state space for $k = 3$ drops by 40% between a context-CTCR of 5 and 6. Additionally, the state space for all context-coverage criteria, i.e., $k = 1$, $k = 2$, $k = 3$, and $k = 6$, converge to the same size. For example, with a context-CTCR of 10, the size of the state space is always 50.

The reason for this is that the possibility for a valid combination of contexts decreases with a high context-CTCR, i.e, with a context-CTCR > 10, only single contexts are coverable, i.e., a greater k than 1 is not possible.

PRESENCE OF UNRESTRICTED CONTEXTS/FEATURES. I further investigate the potential memory savings by state subsumptions using partial states. Therefore, the percentage of partial states in a state space, which contains either a single unrestricted context or feature ($l = 1$) or a pair-wise combination of contexts or features ($l = 2$) is measured.

Figure 6.18 shows the percentage of partial states within an incomplete state space for $k = 2$. For $l = 1$, i.e., one context or feature identifiable as unrestricted, about 50% of the states in the state space are partial states for a feature model size of 15 and 20. The remaining states are complete states for which it is not possible to derive a single unrestricted feature. The standard deviation for a feature model size of 20 indicates that up to 80% of the states may be abstracted to a partial state. If the combination of unrestricted features in a state is increased to $l = 2$, this amount of partial states decreases in most cases, e.g., for a feature model size of 20, 38% of the states are a partial state with a pair-wise combination of unrestricted features.

An anomaly regarding partial states is recognizable with a feature model size of 55. In this case, there are more partial states that contain a combination of unrestricted features than partial states that contain only a single unrestricted feature. The reason for this is the low amount of constraints in this set of feature models. With such a low amount of constraints it is more likely that features may become unrestricted in an *arbitrary* combination. Such a combinatorial set may become larger than a set of single unrestricted features and, therefore, there are more possibilities for combinations of unrestricted features.

Figure 6.19 shows the percentage of partial states that contain unrestricted contexts within an incomplete state space for $k = 2$, scaling over the context-CTCR. With a low contexts-CTCR more contexts may become unrestricted, either one-wise $l = 1$ or pair-wise $l = 2$, and, therefore, more states in a state space are abstracted to a partial state. If the context-CTCR increases the possibility for unrestricted contexts decreases. For example, with a context-CTCR of 2, 38% of the states are partial with a single unrestricted context and 20% of the states are a partial states, which contain a pair-wise combination of unrestricted contexts. In contrast to that, with a context-CTCR of 9, 32% of the states are partial with a single unrestricted context and a pair-wise combination of unrestricted context was impossible for this set of context-feature models. Thus, with every additional

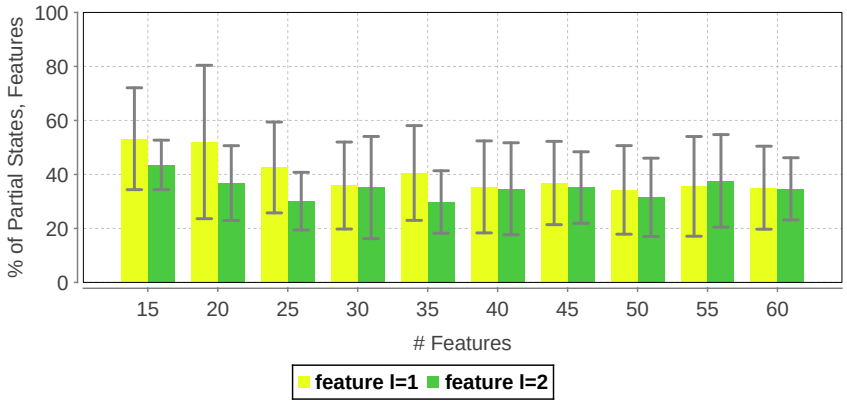


Figure 6.18: Presence of Unrestricted Feature Variables, Scaling across Number of Features

exclude a require constraint between a context and a feature, the possibility to identify unrestricted contexts decreases.

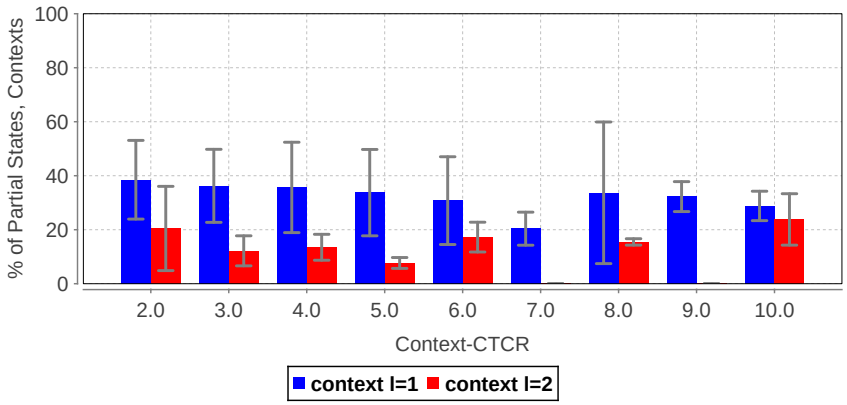


Figure 6.19: Presence of Unrestricted Context Variables, Scaling across Context-CTCR

Similarly, the impact of different choices for k and l is investigated w.r.t. the trade-off between resource consumption and computational efforts at runtime in the next section.

6.6.2 *DSPL Reconfiguration at Runtime*

An appropriate choice of pre-computation reduction criterion for a k-wise context combination and an upper bound of unrestricted combination of contexts/features l depends on the actual runtime behavior. In order to investigate the impact of an incomplete state space, the ratio of how many states are covered by the pre-planned incomplete state space w.r.t. a contextual coverage of k-wise combination and how many states are to be derived on-demand has to be evaluated. Therefore, a random trace of 500 contextual changes $\langle \oplus / \ominus \ context \rangle$ is generated. Further, the capabilities of partial states for a flexible reconfiguration at runtime is investigated by measuring the ratio of self-transitions, i.e., reconfiguring unrestricted features,

in comparison to reconfiguration transitions, i.e., reconfiguring features that are not unrestricted.

STATE COVERAGE AT RUNTIME. To investigate the benefits of an incomplete state space, incomplete state spaces are pre-computed for a k-wise context combination. These state spaces are analyzed w.r.t. the necessity to execute a solver call at runtime to execute a reconfiguration at runtime, ranging across the size of a feature model. Therefore, the frequency of on-demand configuration computations requiring solver calls at runtime is evaluated for $k = 1$, $k = 2$, $k = 3$, and $k = 6$. To simulate contextual changes, a random event trace is generated, containing 500 events of entering and leaving a context.

To have a baseline for comparison, a state space containing a single initial state is also considered for the evaluation. Correspondingly, this state space heavily relies on an on-demand computation of a configuration if the context changes. Note that the state space coverage of a complete state space is not evaluated because the coverage is, based on its completeness, always 100%.

Figure 6.20 depicts the results for ranging across the number of features of a feature model. In case of an initially empty state space, the coverage achieved is always below 10% and, therefore, more than 90% of the states are to be discovered at runtime by invoking a solver call. The coverage of an initially empty state is greater than 0% because (i) the generated trace of contextual changes revisits the contextual situations repeatedly and (ii) the state space is extended with an suitable state on-demand that remains in state space.

By increasing k, the initial state space coverage increases accordingly. For $k = 1$, between 30% and 70% of the states required at runtime have already been pre-computed and between 70% and 30% of the states required at runtime have to be derived at runtime. The results indicate that with $k = 1$ there is a high deviation in the results, indicating that the coverage may be worse in some cases. In contrast to that, for $k = 2$ the coverage ratio is between 70% and 90% and, therefore, only between 10% and 30% of the states in the state space after the trace has been processed, have been derived on-demand at runtime by invoking a solver call.

For $k = 3$ and $k = 6$, the coverage ratio is nearly 100%. To be exact, with an exception for the dataset of a feature model size of 30, the pre-computed state space for $k = 6$ always covers completely the states required at runtime. In this regard, there is basically no need to derive a new state on-demand at runtime if the context-coverage ratio is larger than 3. This implies that for the randomly generated trace of 500 events, a context combination of up to three contexts was not exceeded in most cases.

RECONFIGURATION RATIO AT RUNTIME WITH PARTIAL STATES. In addition to the benefit of a reduction in the memory consumption, partial states improve the flexibility of a state space at runtime. A partial state is an abstraction of multiple fully configured states. Therefore, the necessity to execute a reconfiguration to a new target state is reduced if unrestricted features or contexts are

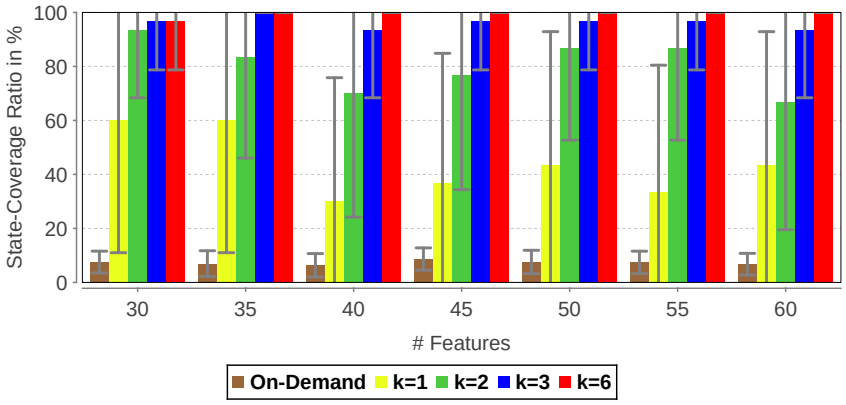


Figure 6.20: State Coverage at Runtime

affected from the change in the contextual situation. In such a case a self-transition is executed instead of a reconfiguration.

Figure 6.21 illustrates the accumulated number of reconfigurations triggered by context change events, ranging from 0 to 500 events. The plot illustrates that with increasing number of unrestricted contexts or features l in a partial state, less reconfigurations are executed and more self-transitions are executed instead. After 500 events, 500 reconfigurations are executed if no unrestricted contexts or features are available in any state within the state space. In contrast to that, 450 reconfigurations are executed if there is at least one unrestricted context/feature per partial state, i.e., 50 self-transitions are executed instead, and about 400 reconfigurations are executed if two unrestricted contexts/features are available, i.e., 100 self-transitions are executed instead. The plot further illustrates that this tendency, in which reconfigurations are reduced, further increases the more events are emitted.

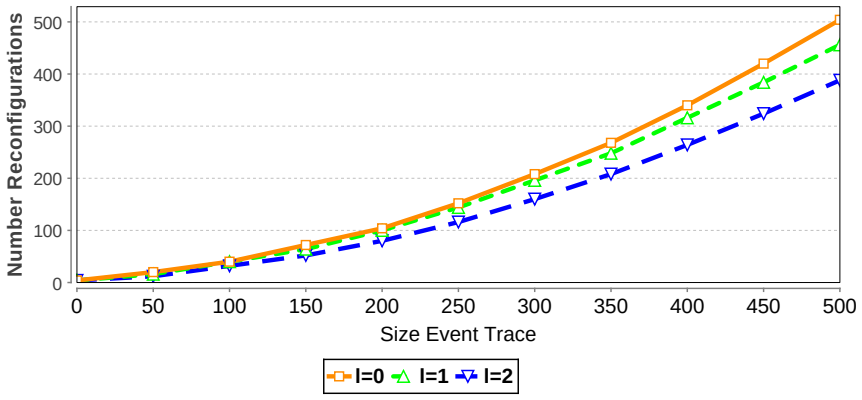


Figure 6.21: Reconfiguration Ratio at Runtime with Partial States

Summarizing, my experiments show that the size of the state space is significantly reducible by covering contextual situations. The higher the contextual coverage, the fewer configurations have to be computed at runtime. However, comparing a setting of $k = 2$ and $k = 3$, both settings achieve a similar completeness regarding the minimization of on-demand computations at runtime. The in-

vestigation of a pair-wise combination of unrestricted contexts/features indicates that the amount of reconfigurations at runtime is drastically reducible. These observations justify the costly pre-computation of unrestricted features/contexts.

6.6.3 *Identification of Unrestricted Features*

The identification of unrestricted feature variables is costly. An identification of all possible partial configurations with a maximization of unrestricted variable combinations does not scale with the number of variables. Therefore, I introduced the three different strategies genetic variable ordering, heuristic variable ordering, and randomized variable ordering to identify partial configurations with a maximized set of unrestricted feature variables in Section 6.3.2.

As a baseline for the comparison an implementation of the Quine and McCluskey algorithm (QMC) [Die78] is also evaluated. This algorithm is used to compute a minimal disjunctive normal form. During this process, the algorithm minimizes a propositional formula by removing irrelevant variables. Therefore, this algorithm is also applicable to identify unrestricted feature variables in a partial configuration, i.e., those variables, which are removed by the QMC algorithm, correspond to unrestricted feature variables.

The feature models used in the evaluation are generated using the BeTTy-Framework[3] with a cross-tree-constraint ratio (CTCR) of 20%, and an equal probability of optional, mandatory, or-group, and alternative-group constraints of 25%. The evaluation sets scale across the size of different context-feature models from 15 to 60 features. Every evaluation set consists of 30 randomly generated feature models and the results are averaged across these models.

COMPUTATIONAL TIME. Figure 6.22 depicts the time to compute a set of partial configurations with unrestricted variables for the different strategies of a genetic variable ordering, a heuristic variable ordering, a random variable ordering and QMC as a baseline for comparison. The plot scales across a set of feature models *fm*, ranging from 15 features to 60 features. The time is depicted on a logarithmic scale from 0 to 1,000 seconds.

The plot shows that the QMC approach scales worst. The time to identify unrestricted features increases exponentially from 1 second for an *fm* size of 15 features to 1,000 seconds for an *fm* size of 20 features. I abstain at this point from a more detailed evaluation of the QMC approach for a feature model size larger than 20 because it is too time consuming.

The remaining strategies perform similar and every one of them outperforms the QMC strategy. The heuristic ordering strategy performs best in average, increasing linearly from 0.03 seconds for a *fm* size of 15 to 12 seconds for a *fm* size of 60. Although the genetic ordering strategy performs best with a small *fm*, the strategy is outperformed by the heuristic ordering strategy for all generated models with a size ranging from 15 to 60. The third, randomized ordering strategy performs only a little worse than the genetic ordering strategy, especially for a small *fm*.

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

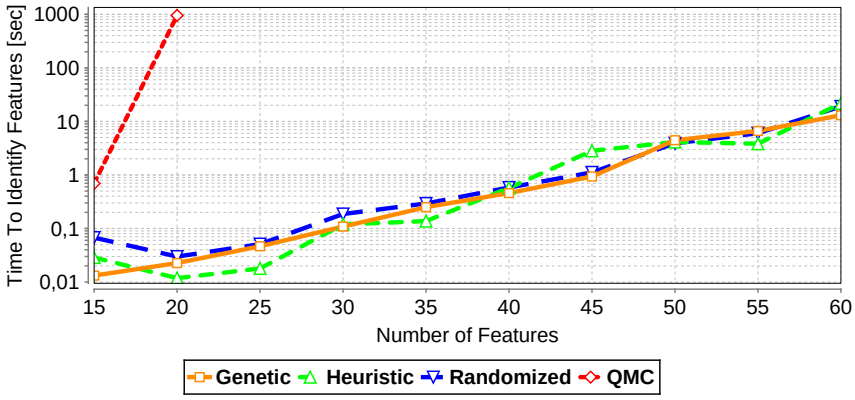


Figure 6.22: Time To Discover Unrestricted Features Scaling across Size of *fm*

EFFECTIVENESS OF IDENTIFICATION. The effectiveness of every strategy to identify unrestricted variables is measurable in the amount of unrestricted feature variables per partial state that are discovered for a feature model *fm*. Figure 6.23 depicts a comparison of the strategies genetic variable ordering, heuristic variable ordering, randomized variable ordering, and QMC in their effectiveness of identifying unrestricted features for a partial state. The plot scales across a set of feature models *fm*, ranging from 15 features to 60 features. The effectiveness is depicted across a linear scale from 0 to 18 identified unrestricted features for a partial state per *fm*.

The plot indicates that the genetic, randomized and QMC strategies perform nearly equal. The heuristic strategy discovers always the least number of unrestricted features. In every evaluation one feature is missed by this strategy in comparison to the remaining strategies. As previously stated, an evaluation of the QMC strategy is only conducted for *fm* with 15 and 20 features because it is too time consuming for larger *fm*. However, for a *fm* with 15 and 20 features, the genetic, heuristic, and QMC strategy perform nearly equally well, although QMC still performs worst of all strategies. A more detailed comparison for larger *fm* indicates that the randomized strategy performs best overall. In 6 of 10 evaluations, the genetic strategy performs minimal worse than the randomized strategy.

Note that the amount of unrestricted features does not necessarily increase with the amount of features in an *fm*. For example, the evaluation with *fm* size of 25, 35, and 55 features contain less unrestricted features than the respective previous *fm* evaluation set. This is the result of the randomized generation of every *fm*. A disadvantageous collection of constraints may increase the interdependency between features and, therefore, minimize the amount of possible unrestricted features.

Summarizing, my experiments show that the identification of unrestricted features is conductible in a time efficient manner. Although partial configurations are derived with the same amount of unrestricted features an exponential increase in the computational time is avoided by using one of the BDD-based strategies genetic, heuristic, or randomized. Each of those strategies is faster than the state of the art QMC algorithm to minimize propositional formula by achieving even better results.

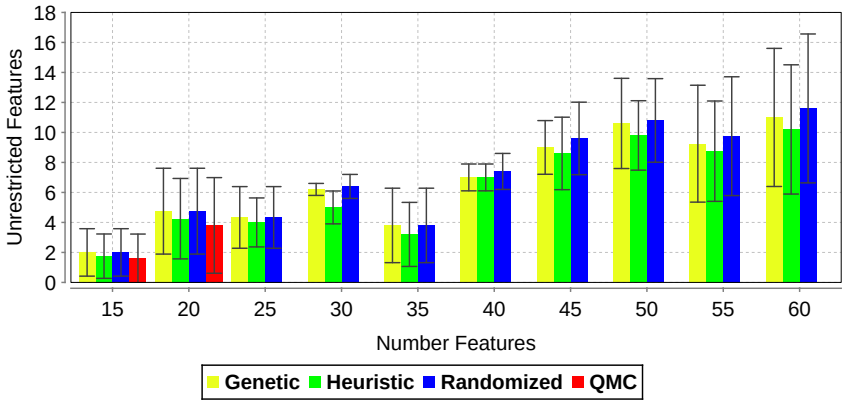


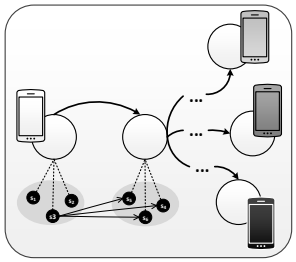
Figure 6.23: Discovery of Unrestricted Features Scaling across Size of *fm*

Comparing the effectiveness to identify partial configurations with a maximum amount of unrestricted features, the randomized BDD-ordering strategy outperformed every other approach. Although, the heuristic and genetic strategies are faster than the randomized strategy, the difference is only minor. I conclude that both, the randomized and genetic strategy, provide the overall best efficiency in a cost-effectiveness trade-off to identify unrestricted features, where the random strategy is better in maximizing the amount of unrestricted features and the genetic strategy is faster.

This Chapter introduced concepts and techniques to reduce the impact of a reconfiguration transition system. However, to further reduce the overall costs of a DSPL-based adaptation process the operational costs of a reconfiguration at runtime may be reduced, which is discussed in the next chapter.

7

RECONFIGURATION PREDICTION



The reduction approaches for a feature model and transition system are only one aspect in the overall minimization of the costs imposed by a DSPL-based adaptation process. To achieve the second goal G2 of this thesis, i.e., improving the resource utilization of an adaption of a mobile device, the reduction of the energy consumption utilized by a configuration state and by a reconfiguration are also crucial. Various methods for estimating and minimizing the energy consumption of embedded systems have been proposed for different levels of design abstractions [SPA12]. Again, approaches that rely on a model-based specification, such as on a transition system, show promising results in this regard [CGKM12]. For instance, recent methods for functional adaptation planning are extendable to reduce the energy consumption of mobile devices at runtime. Those approaches are, in general, capable to handle various kinds of *quantifiable* non-functional properties such as stability, robustness, and costs, by enriching a functional specification of an adaptive system with respective non-functional information [PS09, SHMK10, SPA12].

A further characteristic of this application domain is the fact that the average usage of mobile devices follows certain change-patterns in the contextual situation [MSRJ12, NSL+12]. For example, every working day a user leaves his *home* by entering his *car* and driving into his *office* in the morning. After finishing time, the user leaves his *office*, enters his *car* and drives back *home*. Therefore, the required reconfiguration of mobile devices are, up to a certain *probability*, predictable, e.g., by tracking contextual changes, such as time or location, at runtime [JL10]. This information may be captured in an evolving probabilistic contextual model that, in combination with a quantified reconfiguration model, builds a basis for fine-grained long-term reconfiguration planning. This way, target configurations may be chosen with a better cost behavior w.r.t. anticipated future contextual changes. However, until now, existing reconfiguration planning approaches do not take such mobility patterns into account [NSL+12].

Planning as Model Checking constitutes a research domain that uses behavioral models, such as transition systems, to specify the reconfiguration capabilities of a device and to derive appropriate reconfiguration plans [CRT98, GT00]. Such a plan is derived w.r.t. requirements that are to be satisfied by every state on such a path, e.g., `Cellular` has to be active in every configuration state on the path. Those requirements are specified by using temporal logics. A corresponding

reconfiguration model is usually specified as a transition system, in which each state represents a system configuration that satisfies certain system properties and each transition correspond to a valid reconfiguration. Based on this specification, a reconfiguration plan is obtained by means of a path leading from a source state to a target state that satisfies the given contextual requirements.

As a sample scenario, assume a smartphone user entering his `Office` from his `Car` and leaves it shortly afterwards by entering his `Car` again. This sequence of contextual changes is depicted in Figure 7.1. The device may adapt itself from a state s_3 that corresponds to a configuration, in which the feature `Cellular` is active, to a configuration state s_5, which relies on a `WLAN AP`-based connection via the access point in his `Office`. This reconfiguration utilizes costs of 5, e.g., due to the (de-)activation of several features. Shortly afterwards, the feature `WLAN AP` is deactivated and `Cellular` is reactivated, i.e., if the user leaves `Office` the device reconfigures itself from s_5 back to s_3 with costs of 7. In this example, the energy consumption of the reconfiguration process may be reduced, if the device reconfigures itself according to the requirements imposed by `Car` as well as `Office` *and* by also choosing a reconfiguration to a state that is similar to the starting configuration in the context `Car`. The reason for this opportunity to reduce the costs is that the change to `Office` is only temporary. In such a case, avoiding a local optimum by executing a reconfiguration, which is *not* the cheapest choice locally, may reduce the overall reconfiguration costs on a long term basis. Figure 7.1 depicts this opportunity for a reduction. Executing $s_3 \rightarrow s_6 \rightarrow s_3$ instead of $s_3 \rightarrow s_5 \rightarrow s_3$ is cheaper on a long-term basis. By predicting the upcoming contextual change from `Office` to `Car`, the device avoids the temporary switch from a `Cellular` to a `WLAN AP` connection and reduces the reconfiguration costs from 12 to 10.

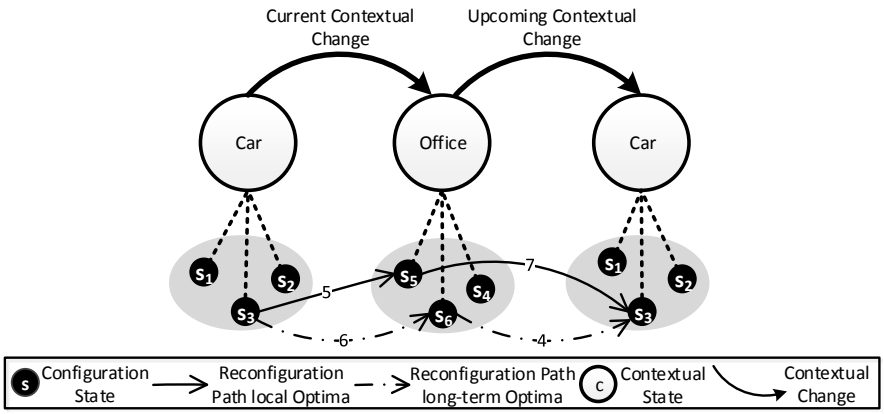


Figure 7.1: Reconfiguration based on Prediction of Contextual Changes

To achieve such a prediction, the following two aspects have to be met.

(1) The contextual changes have to be monitored by the device and those changes have to be continuously tracked and updated, and

(2) the behavior of the user has to be consistent with the previously tracked behavior. Sudden changes in the behavior result in deviations of the prediction of contextual changes.

For example, the system tracks a consistent behavior such as *"3 out of 4 contextual changes occurred in the contextual situation* `Office` *are changes to the contextual situation* `Car`*"*. In this case, a contextual change from `Office` to `Car` is *predictable* with a *probability* of $\frac{3}{4}$, i.e., 75%. Assume that one day, the car of the user is at the repair shop and the user takes a public transport to get home from work instead. At this day, a contextual change from `Office` to `Car` is still predicted with a probability of 75%, although the user does *not* enter his car after work. To provide a user-specific prediction, every change in the contextual situation monitored by the device triggers an update of the probability of that contextual change. In this case, the previously mentioned probability of a contextual change from `Office` to `Car` is updated to $\frac{3}{5}$.

In this chapter, I propose a reconfiguration planning framework for choosing reconfigurations with presumably optimal long-term non-functional properties w.r.t. predicted contextual changes. Therefore, I aggregate and compare the cost probability distribution of all reconfiguration sequences on the basis of *Planning as Model Checking* techniques. The framework consists of

- a quantified *reconfiguration model* that specifies transitions between configuration states. These transitions are annotated with the costs utilized when the transition is executed. These quantitative properties are specified by the means of a weighted automaton [CDH10], and
- a continuously updated *contextual model* to predict subsequent contextual changes. Such a prediction is derived by tracking the state of a context and the probability to change to a certain subsequent contextual state These probabilistic properties are specified by the means of a probabilistic automaton [Sto02].

The integration of both models results in a reconfiguration planning model, i.e., a *probabilistic weighted automaton* [CDH09].

This chapter is structured as follows. At first, the concept of model checking and the research domain *Planning as Model Checking* is introduced. Afterwards, I describe the formalized cost-sensitive reconfiguration concept based on the Nexus DSPL running example. This concept is used for a planning algorithm, which is introduced and discussed w.r.t. traditional model checking techniques. Finally, the planning algorithm is evaluated in a simulative manner to investigate benefits and trade-offs of my approach.

7.1 Planning via Model Checking

Model checking refers to the problem of verifying whether a given model satisfies formal properties [CGP99]. For example, in the previous chapter I specified the reconfiguration behavior of a DSPL based on a (partial) Kripke Structure as system model. Using model checking, it is possible to analyze such a model and to verify whether a property, such as that "*the feature* `VoIP` *has to be active until* `Cellular Call` *is active*". This property has to hold in every state on a path utilized by a reconfiguration from `VoIP` to `Cellular Call`. As already pointed out in the previous chapter, such paths are based on the assumption that the

configuration states of an SAS are *not* fully connected, i.e., every configuration state is *not* reachable with one reconfiguration from every other configuration. Instead, configuration states are fully path-connected, i.e., every configuration state is eventually reachable from every other configuration state via a path of consecutive reconfigurations. Such paths are necessary to model certain aspects, such as a hand-over of an active phone call. Therefore, a path of consecutive reconfigurations needs to be executed. For example, the reconfiguration from a `VoIP`-based call to a `Cellular Call` starts in a state, in which `VoIP` is active. To execute the hand-over of an active phone call, a reconfiguration is executed to an intermediate state, in which `VoIP` and `Cellular Call` are active at the same time. The final reconfiguration is executed to the target state, in which only `Cellular Call` is active.

The technique of model checking is used to pre-plan paths of reconfigurations, on which certain properties hold, which is referred to as *Planning as Model Checking* [GT00]. This section describes the concepts of model checking and how *Planning as Model Checking* is used to pre-plan reconfigurations of a DSPL.

7.1.1 *Model Checking*

Model checking is a technique for automatically verifying correctness properties of a behavioral system abstraction with a finite number of states. A *property* specifies *how* the system *should (not)* behave and a *model* specifies *how* the system *actually* behaves. In this regard, model checking is described best by the following quote from Baier et al.

> "Model checking is an automated technique that, given a finite state model of a system and a formal property, systematically checks whether this property holds for that model." [BK08]

For a model-based verification technique such as model checking, the model describing the possible system behavior as well as the properties that are to be verified have to be specified in a mathematically precise and unambiguous way. To this end, model checking properties are formulated via temporal logics [BK08]. In this regard, model checking verifies if a model satisfies a given temporal formula or not.

Model checking is a technique that relies on an exhaustive exploration of the complete state space [Sim14]. A model checker is a tool that implements a model checking algorithm. A model checker explores all possible system states and examines every possible state-transition path to verify if a given system property holds.

Properties that may be verified by a model checker are of a qualitative nature and focus on *state properties*, e.g., "*the feature* `Connectivity` *is always active*". Further, a model checker is capable to verify *properties* on a *path* of consecutive transitions, e.g., example, "*the feature* `VoIP` *has to be active until the feature* `Cellular Call` *becomes active*". These state and path properties are specified via a temporal logic such as the Linear Temporal Logic (LTL) [Roz11] and Computational Tree Logic (CTL) [CGP99]. *Planning As Model Checking* techniques exploit such a system model and temporal logics to obtain paths that satisfy the stated properties.

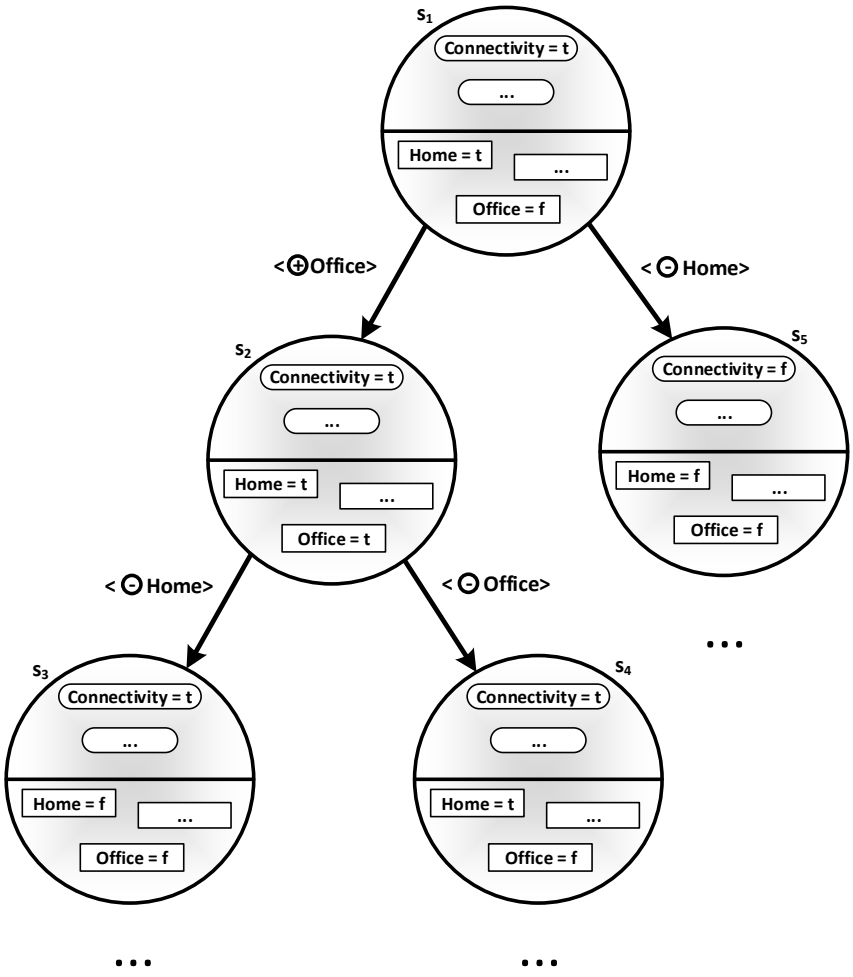


Figure 7.2: Reconfiguration Paths Explored by a Model Checker

Let us revisit the previous example on a hand-over of a phone call. With a fully path-connected transition system, there may be multiple paths from a configuration state, in which `VoIP` is active, to a configuration state, in which `Cellular Call` is active. By applying *Planning As Model Checking*, those paths are derivable, which ensure the required hand-over state-transition sequence between `VoIP` and `Cellular Call`.

Example 7.1 (*Model Checking*).

For the Nexus DSPL a $\mathcal{PKS}$ is used as a system model to execute a model checking algorithm. A model checker is capable to verify whether a property such as "*The feature* `Connectivity` *is active in every configuration state*" holds. If a state exists that does not satisfy the property, the model checker returns a path of state-transitions which lead to the state that violates the system property.

Figure 7.2 depicts an extract of possible reconfiguration paths, starting from a configuration state in which the feature `Connectivity` is active. The two paths depicted on the left-hand side of the tree correspond to $s_1 \xrightarrow{\oplus Office} s_2 \xrightarrow{\ominus Home} s_3$

and $s_1 \xrightarrow{\oplus Office} s_2 \xrightarrow{\ominus Office} s_4$, respectively. Every state on these two paths satisfies the property that `Connectivity` has to be active, i.e., `Connectivity=t`. However, the path on the right-hand side of the tree $s_1 \xrightarrow{\ominus Home} s_5$ leads to a state s_5 that does not satisfy the stated property since `Connectivity` is inactive. A model checker discovers the state that does not satisfy this property by examining every possible reconfiguration path.

The depicted reconfiguration paths on the right-hand side of the tree further satisfy a path property such as "*the feature* `Home` *has to be active until the feature* `Office` *becomes active*", whereas the right-hand side path of the tree does not satisfy this property because `Home` becomes inactive before `Office` is active.

The next section explains how the research domain *Planning As Model Checking* uses model checking to derive a reconfiguration path that satisfies a given property hold and how such a concept may be leveraged to reduce the reconfiguration costs on a long-term basis.

7.1.2 *Reconfiguration Planning*

For a contextual change a reconfiguration is usually to be chosen among several appropriate reconfigurations, in which each of those reconfigurations result in a state that satisfies the current contextual situation. The goal of the domain of *Planning as Model Checking* is to derive a *path* from a source state to a target state, on which certain properties hold, such as requirements imposed by a contextual situation. Therefore, if multiple reconfigurations are executed for a contextual change, with *Planning as Model Checking*, a path is derived, which ensures that stated properties are not violated during the reconfiguration. For example, for a contextual change such as $s \xrightarrow{\oplus \texttt{Office}} s' \xrightarrow{\ominus \texttt{Home}} s''$, it has to hold that the feature `Connectivity` remains active. In this regard, a model checker derives a path of consecutive reconfigurations on which "`Connectivity=t`" holds in every state.

In addition to functional properties, such as that a certain feature has to be (in-)active, *non-functional* properties are used to identify suitable target states for a reconfiguration. Thus, depending on the current contextual situation, a particular system configuration may be preferred to others, e.g., regarding energy costs. For example, an Internet connection via WLAN is cheap and fast, but only available in a contextual situation providing a WLAN access point in order to activate `WLAN AP`, e.g., in the contexts `Office` or `Home`. In those contexts, there is a choice between a configuration that uses a (cheap) connection via `WLAN AP` and a configuration that uses a `Cellular`-based connection. Furthermore, costs are also utilized by a reconfiguration (path) from a source state to a target state, e.g., due to the (de-)activation of features during the process of reconfiguration. Therefore, the overall costs utilized at runtime correspond not only to the costs utilized during the reconfiguration but also to the costs utilized by the target configuration state.

The *Planning as Model Checking* techniques [CRT98, GT00] incorporate transition systems to specify not only the system behavior but further also include non-functional properties such as costs. However, recent approaches do, to the best of my knowledge, not incorporate capabilities to trace and predict behavioral patterns of

changes that trigger a reconfiguration [NSL+12]. Such a prediction offers the possibility to improve the costs utilized by a reconfiguration process on a long-term basis w.r.t. the specified non-functional properties.

Example 7.2 (*Costs of a Reconfiguration*).

Figure 7.3 depicts a simplified reconfiguration automaton denoting reconfiguration costs for the Nexus DSPL example. Every transition is annotated with costs utilized by the reconfiguration from source state to target state. The costs utilized by a state for being active are denoted by a self-transition. Every configuration state is equipped with such a self-transition.

The configuration states s_1 and s_2 satisfy the requirements of the contextual state incorporating the contextual situation {`Slow Network`}. Let us assume that the contextual situation {`Slow Network`} is currently active and the device moves to the contextual situation {`Slow Network`, `Home`}. In this case, a reconfiguration to one of the configuration states s_3, s_4, s_5, or s_6 is to be executed, in which each reconfiguration choice results in different overall costs, e.g., $s_1 \rightarrow s_3$ utilizes costs of 9 whereas $s_1 \rightarrow s_5$ utilizes costs of 7. The costs in this example may correspond to the energy consumed by the (de-)activation of features during the reconfiguration. Furthermore, each of the possible target states utilizes different costs while being active. For example, every time the devices remains in the state s_5 costs of 6 are utilized. The costs in this example may correspond to the energy consumed by the set of active features in s_5.

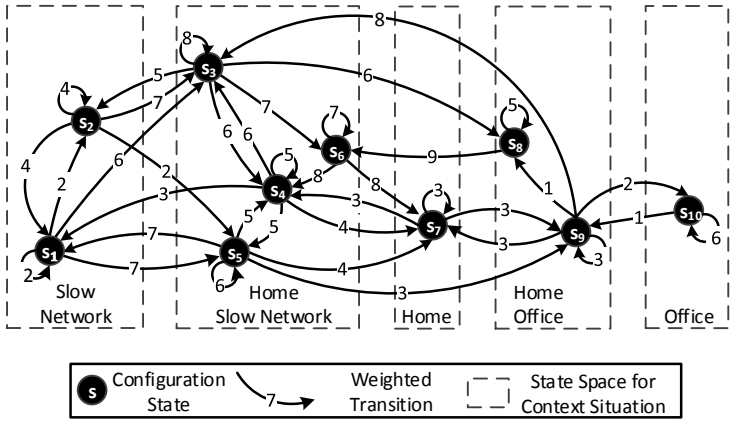


Figure 7.3: Weighted Reconfiguration Automaton

For choosing among those reconfiguration options and reducing the overall reconfiguration costs on a long-term basis, two transition systems are introduced in the next section. A *reconfiguration automaton* as a model describing the costs utilized at runtime and a *contextual automaton* as a model to track the changes in the contextual situations.

7.2 Combining Costs and Probabilities of Reconfigurations

A change in the contextual situation of a device triggers a reconfiguration transition between configuration states. Therefore, the reconfiguration behavior of a

DSPL may be represented as a transition system whose set of states refer to the set of feature configurations of the feature model. In the previous chapter, I specified such reconfiguration behavior on the basis of a (partial) Kripke Structure, c.f., Definitions 6.1 and 6.4, to describe abstraction concepts such as an *incomplete state space* and *partial states*. This chapter focuses on the reduction of costs utilized at runtime. Therefore, non-function properties of configuration states and reconfiguration transitions are required. Furthermore, a probabilistic model is required to track the contextual behavior. To tackle this, and in accordance with existing approaches [CDH10, Sto02], I use

- a *weighted automaton* to specify costs utilized at runtime and
- a *probabilistic automaton* to track the probability of contextual changes.

7.2.1 *Cost Model for Reconfigurations*

Every reconfiguration executed at runtime utilizes *costs*, e.g., the energy consumption for deactivating and activating affected software and hardware features. Hence, when choosing among multiple possible reconfigurations, those with lower costs should be preferred. Therefore, transitions in a transition system, which is specifically intended to model the costs of a reconfiguration, carry additional *weights* by means of discrete numerical values denoting their estimated costs as depicted in Figure 7.3. For instance, let us assume that the configuration state s_1 is currently active and the device changes into the contextual situation {`Slow Network`, `Home`}. In this case, the system reconfigures itself either to state s_3 or to s_5. A reconfiguration to s_5 utilizes costs of 7, whereas a reconfiguration to s_3 causes costs of 6. However, s_3 utilizes more costs while being active. Thus, a reconfiguration to s_3 utilizes costs of 6+8=14, whereas a reconfiguration to s_3 utilizes costs of 7+6=13.

A *Weighted Automaton* ($\mathcal{WA}$) is used to specify the reconfiguration costs between states and the costs of states for being active. Such a weighted automaton consists of a set of configuration states $\mathcal{S}$, in which each state corresponds to a valid configuration $\gamma\big|_{\mathcal{F}} \in \hat{\Gamma}_{cfm}$ of a context-feature model *cfm*, restricted to the set of features $\mathcal{F}$. Thus, each state $s \in \mathcal{S}$ does *not* further interpret contexts $c \in \mathcal{C}$, as I do in the state space of a (partial) Kripke Structure. The contextual behavior is specified in an additional *probabilistic automaton*, which is discussed in the next section. I further assume the weight of a transition $s \longrightarrow s'$ to denote the costs for a reconfiguration between two states. A self-transition $s \longrightarrow s$ denotes the costs utilized by state for being active.

A weight function $\omega_{\mathcal{S}} : \mathcal{S} \times \mathcal{S} \to \mathbb{R} \cup \{\infty\}$ evaluates the costs of a transition, e.g., the costs utilized by (de-)activating a set of features during a reconfiguration. For example, a reconfiguration from a configuration state in which `WLAN` is inactive to a state in which `WLAN` is active may utilize 4 costs for the activation of the feature. If the device is restricted to execute a reconfiguration from s to s', the transition has an infinite weight.

Definition 7.1 (*Weighted Automaton (WA) [CDH10]*)**.** A *weighted automaton* is a 3-tuple $(\mathcal{S}, \longrightarrow, \omega_{\mathcal{S}})$ where

- $\mathcal{S} = \{\gamma\big|_{\mathcal{F}} \ : \ \gamma \in \hat{\Gamma}_{cfm}\}$ is a finite set of *feature configuration states*,
- $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a *transition relation*, and
- $\omega_{\mathcal{S}} : \mathcal{S} \times \mathcal{S} \to \mathbb{Q} \cup \{\infty\}$ is a *weight function* where $\omega_{\mathcal{S}}(s, s') \in \mathbb{Q}$ if $s \to s'$ and $\omega_{\mathcal{S}}(s, s') = \infty$ if $s \not\to s'$.

A finite sequence of reconfiguration transitions $s_0 \to s_1 \to s_2 \cdots \to s_n$ corresponds to a path π. The $\mathcal{WA}$ specifies a set $\Pi_{\mathcal{WA}}$ of such reconfiguration paths.

Every path starts in some initial configuration state $s_0 \in \mathcal{S}$ and ends in some target state $s_n \in \mathcal{S}$. As introduced in the previous chapter, I denote possible path(s) from a source state to a target state with $s_0 \Rightarrow s_n$ for short.

Notation 7.1 (Reconfiguration Path)**.** A path $\pi \in \Pi_{\mathcal{WA}}$ denotes a sequence of consecutive transitions $s_0 \Rightarrow s_n$ from a source state s_0 to a target state s_n, i.e.,

$$\pi = s_0 \to s_1 \to s_2 \cdots \to s_n,$$

with $n \in \mathbb{N}$. By $\Pi_{\mathcal{WA}}$ I denote the set of all possible finite paths for a weighted automaton $\mathcal{WA}$.

Chatterjee et al. [CDH10] proposed to aggregate the weights of a path $\pi \in \Pi_{\mathcal{WA}}$ with a generalized value function. Such an aggregation may correspond to the average, the maximum, or the minimum of weights.

In the following, I use a summation function $\mathbb{V} : \Pi \to \mathbb{Q}$ to compute the sum of a sequence of weights $w_1 w_2 \cdots w_n$, with $w_i = \omega_{\mathcal{S}}(s_{i-1}, s_i)$. For a sequence of weights a summation of the weights on a path π is computed by $\mathbb{V}(\pi) \in \mathbb{Q} = \sum_{i=1}^{n} \omega_{\mathcal{S}}(s_{i-1}, s_i)$.

Definition 7.2 (*Summation Function [CDH10]*)**.** Given a finite sequence of transition weights $w_1 w_2 \cdots w_n$ utilized by a path $\pi \in \Pi_{\mathcal{WA}}$, the summation function $\mathbb{V} : \Pi_{\mathcal{WA}} \to \mathbb{Q}$ computes the sum of the weights as follows

$$\mathbb{V}(\pi) \in \mathbb{Q} \ = \ \sum_{i=1}^{n} \omega_{\mathcal{S}}(s_{i-1}, s_i),$$

with $n \in \mathbb{N}$ denoting the length of the path.

By intuition, choosing the reconfiguration option for a state that utilizes the fewest costs should be preferred to reduce the costs of a continuous reconfiguration process. The following example illustrates a reconfiguration process based on this assumption.

Example 7.3 (*Reconfiguration Costs by Choosing Local Optimum for a State*).

Source State	Source Context	Target Context	Target State	Costs
s_1	{Slow Network}	{Slow Network, Home}	s_5	7 + 6
s_5	{Slow Network, Home}	{Home}	s_9	3 + 3
s_9	{Home}	{Slow Network, Home}	s_4	3 + 3 + 5
			Overall Cost:	30

Table 7.1: Adaptation Sequence Without Prediction

Table 7.1 lists an example for a reconfiguration path utilized by the following sequence of three contextual changes from {Slow Network} to {Slow Network, Home} to {Home} to {Slow Network, Home}. This path utilizes a sum of

$$\begin{aligned}\mathbb{V}(\pi) &= \textstyle\sum_{i=1}^{4} \omega_S(s_{i-1}, s_i) \\ &= \omega_S(s_1, s_5) + \omega_S(s_5, s_5) + \omega_S(s_5, s_9) + \omega_S(s_9, s_9) \\ &\quad + \omega_S(s_9, s_7) + \omega_S(s_7, s_4) + \omega_S(s_4, s_4) \\ &= 7 + 6 + 3 + 3 + 3 + 3 + 5 \\ &= 30\end{aligned}$$

costs in total. Note that in this example every target state for a contextual change is only active for a short period of time. Therefore, only one self-transition is executed for every target state.

For example, the contextual change {Home} → {Slow Network, Home} results in a path of two consecutive reconfigurations $s_9 \rightarrow s_7 \rightarrow s_4$. This path utilizes costs of 3+3=6 and the target state s_4 utilizes additional costs of 5. This is an illustrative example, how costs for a reconfiguration are computed. The sum of the costs utilized by *each transition* on the reconfiguration path and the costs utilized by *the target state*.

On a long-term view, constantly choosing the locally cheapest reconfiguration option may not always result in the minimal overall reconfiguration costs, as already pointed out by the example depicted in Figure 7.1. To reduce the overall costs at runtime, subsequent contextual situations potentially emerging at runtime are to be taken into account. Therefore, a *probabilistic automaton* is used in order to track changes in the contextual situation of a device emerging at runtime.

This probabilistic automaton tracks the frequency of contextual changes, e.g., $\frac{3}{4}$ of all contextual changes emerging in Office are executed to Car and $\frac{1}{4}$ contextual changes are executed to Slow Network. Thus, with a probability of 75% a contextual change occurs from Office to Car and with a probability of 25% a contextual change occurs from Office to Slow Netowrk. Based on this information, future upcoming changes may be estimated with a certain probability. Such a *prediction* of contextual changes provides the possibility to choose the cheapest reconfiguration option on a long-term basis.

7.2.2 *Probabilistic Behavioral Model of Contextual Changes*

To anticipate user-specific long-term changes in the contextual situation, a continuous tracking of previous contextual changes are to be captured in an evolving contextual model. Therefore, a contextual automaton is used with a transition probability distribution resulting from previously observed sequences of contextual changes emerging at runtime. Figure 7.4 depicts an extract of a *Probabilistic*

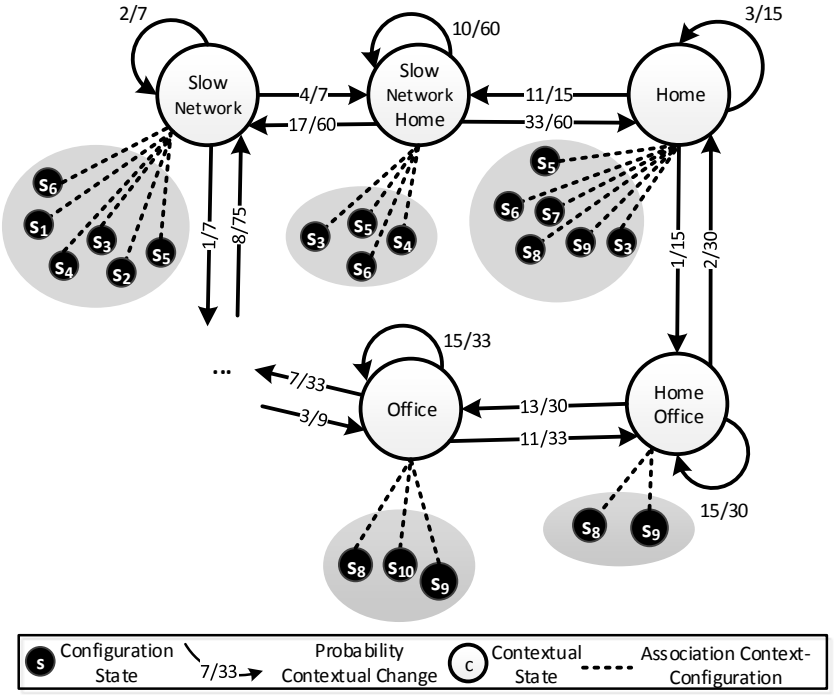


Figure 7.4: Probabilistic Contextual Automaton

Automaton $\mathcal{PA}$ for the Nexus DSPL running example. Every contextual state $q \in \mathcal{Q}$ (depicted as a white circle) of $\mathcal{PA}$ corresponds to a valid configuration $\gamma\big|_{\mathcal{C}} \in \hat{\Gamma}_{cfm}$ of a context-feature model, restricted to the set of contexts $\mathcal{C}$. Thus, every contextual state q resembles a state of the contextual situation and specifies requirements. For example, the contextual situation {`Home`} requires `WLAN AP` to be active and `GPS` to be inactive, according to the context-feature model depicted in Figure 4.2.

As pointed out in the previous chapter, every contextual situation may be satisfiable by several configuration states $s \in \mathcal{S}$ (depicted as black circle), e.g., {`Home`} is satisfiable by 6 states. Further, a configuration state $s \in \mathcal{S}$ may satisfy several contextual situations, e.g., s_6 satisfies {`Home`}, {`Home`, `Slow Network`}, and {`Slow Network`}.

Every transition denotes a change from the source contextual state q to the target contextual state q'. A labeling function $\Sigma_{\mathcal{Q}}$ assigns a label to every transition, i.e. $\sigma \in \Sigma_{\mathcal{Q}} :=$(*source context* q, *target context* q').

The execution of every contextual transition is continuously monitored and updated w.r.t. its frequency of execution. For example, Figure 7.4 depicts the estimated probability of changing from the contextual state {`Slow Network`} to the state {`Home`, `Slow Network`} as $\frac{4}{7}$. This probability is the result of previously moni-

tored frequency of contextual changes. If this contextual change occurs one more time the probability of this change is updated to $\frac{5}{8}$. Additionally, the remaining two specified contextual changes in the source state {`Slow Network`} are updated to $\frac{2}{8}$ and $\frac{1}{8}$, respectively. Self-transitions represent autonomous reconfigurations of the device without contextual changes, e.g., $s_1 \rightarrow s_2$ in {`Slow Network`} with a probability of $\frac{2}{7}$.

The $\mathcal{PA}$ is used as a model to track contextual changes. With $\mathcal{D}(\mathcal{Q})$, I refer to the set of all probability distributions over a set of contextual states $\mathcal{Q}$. A probability distribution $\rho \in \mathcal{D}(\mathcal{Q})$ is used as a function

$$\rho : \mathcal{Q} \rightarrow P$$

to map a state $q \in \mathcal{Q}$ to probability $p \in P$, with $0 < p < 1$. Note that the sum of all probabilities $p \in P$ for a probability distribution ρ is always 1, i.e., $\sum_i^n p_i = 1$, with $n \in \mathbb{N}$.

A probabilistic transition function $\delta_{\mathcal{Q}} : \mathcal{Q} \rightarrow \mathcal{D}(\mathcal{Q})$ assigns a probability distribution $\rho \in \mathcal{D}(\mathcal{Q})$ to a state q to denote the probability of every possible contextual transition from that source state q to some target state q'. For example, the contextual state {`Home`} has three transitions ({`Home`}, {`Home`}), ({`Home`}, {`Slow Network, Home`}), ({`Home`}, {`Home, Office`}). The probabilities of those transitions to be executed are $\delta_{\mathcal{Q}}(\{\texttt{Home}\})(\{\texttt{Home}\}) = \frac{3}{15}$, $\delta_{\mathcal{Q}}(\{\texttt{Home}\})(\{\texttt{Slow Network, Home}\}) = \frac{11}{15}$, and $\delta_{\mathcal{Q}}(\{\texttt{Home}\})(\{\texttt{Home, Office}\}) = \frac{1}{15}$, respectively.

After the automaton is initialized, I assume an initial probability distribution $\rho_I \in \mathcal{D}(\mathcal{Q})$ to be given to denote whether a state is initially active, e.g., as an equal distribution. For example, assuming an equal distribution each contextual state $q \in \mathcal{Q}$ depicted in Figure 7.4 has an initial probability of $\rho_I(q) = \frac{1}{5}$ to be active when the system is initialized.

Definition 7.3 (*Probabilistic Automaton ($\mathcal{PA}$) [Sto02]*)**.** A *probabilistic automaton* is a 4-tuple $(\mathcal{Q}, \rho_I, \Sigma_{\mathcal{Q}}, \delta_{\mathcal{Q}})$ where

- $\mathcal{Q} = \{\gamma\big|_{\mathcal{C}} \ : \ \gamma \in \hat{\Gamma}_{cfm}\}$ is a finite set of *contextual configuration states*,
- $\Sigma_{\mathcal{Q}}$ is a finite set of *transition labels*,
- $\delta_{\mathcal{Q}} : \mathcal{Q} \rightarrow \mathcal{D}(\mathcal{Q})$ is a *probabilistic transition function*, and
- $\rho_I \in \mathcal{D}(\mathcal{Q})$ is an *initial state probability distribution*.

A *run* of $\mathcal{PA}$ corresponds to a sequence of contextual changes $r = \sigma_1, \sigma_2, \cdots \sigma_n \in \Sigma_{\mathcal{Q}}^*$. Such a run corresponds a path $\pi \in \Pi_{\mathcal{PA}} = q_0\sigma_1 q_1 \sigma_2 q_2 \cdots \sigma_n q_n$ of contextual changes.

Notation 7.2 (Contextual Run)**.** A run $r \in \Sigma_{\mathcal{Q}}^*$ denotes a sequence of consecutive contextual changes $\sigma \in \Sigma_{\mathcal{Q}}$

$$r = \sigma_1, \sigma_2, \cdots \sigma_n \in \Sigma_{\mathcal{Q}}^*,$$

in which Σ_Q^* denotes the set of all possible finite runs for $\mathcal{PA}$ and $n \in \mathbb{N}$ denotes the length of the run.

The *probability* $\mathbb{P}(\pi)$ of a finite path $\pi \in \Pi_{\mathcal{PA}}$ corresponds to the probability $\rho_I(q_0)$ of a state $q_0 \in Q$ to be initially active multiplied with every probability $\delta_Q(q)(q')$ of a contextual change (q,q') occurring on a path π.

Definition 7.4 (*Probability of a Path for a contextual Run*)**.** The probability $\mathbb{P} : \Pi_{\mathcal{PA}} \to \mathbb{Q}$ of a path $\pi \in \Pi_{\mathcal{PA}}$ is computed by the following multiplication

$$\mathbb{P}(\pi) \in \mathbb{Q} = \rho_I \cdot (q_0) \prod_{i=1}^{n} \delta_Q(q_{i-1})(q_i).$$

The $\mathcal{PA}$ describes changes in the contextual situation of a device and every state imposes requirements that have to be satisfied by the configuration state $s \in \mathcal{S}$ of the device. To analyze both the probability of contextual changes and the costs of a reconfiguration, I use a *probabilistic weighted automaton* as defined in [CDH09] in the next section. This type of automaton represents the fundamental basis to estimate upcoming costs w.r.t. predicted contextual changes.

7.2.3 *Probabilistic Weighted Automaton*

In order to reason about the costs utilized by a reconfiguration for an estimation of future contextual change, the weighted automaton *and* the probabilistic automaton are combined to one *probabilistic weighted automaton* $\mathcal{PWA}$. This automaton allows for a cost-sensitive reconfiguration planning that is, up to a certain degree, aware of the predicted long-term costs of each possible reconfiguration choice. In a $\mathcal{PWA}$ every state corresponds to a combination of a configuration state and a contextual state. Every transition corresponds either to

- a contextual change without a reconfiguration,
- a reconfiguration without a contextual change,
- a reconfiguration and a contextual change, or
- a self-transition denoting no contextual change or reconfiguration occurred.

Furthermore, every transition is annotated with a weight denoting the utilized costs and a probability denoting the frequency of occurrence.

With such an automaton, an estimation of the upcoming reconfiguration costs w.r.t. future contextual changes is derivable. Therefore, every path leaving a source state is investigated w.r.t. its overall probability to occur. This probability is correlated with the overall costs utilized by a path to derive an cost estimation for that path. The overall estimated costs of all investigated paths correspond to a cost estimation of all possible future reconfigurations for the investigated state.

Such a cost estimation is usable to compare states w.r.t. the costs they probably utilize in the future.

Based on this information, a reconfiguration path is chosen that is the cheapest w.r.t. future emerging contextual changes, instead of choosing the reconfiguration, which is locally the cheapest choice for the currently active configuration. If one compares the listings in Table 7.1 and Table 7.2, it becomes apparent that an alternative reconfiguration path utilizes fewer costs for the sample probability distribution depicted in Figure 7.4. With the derivation of a prediction from the probability distribution of contextual changes, different configuration states are chosen for a reconfiguration.

Example 7.4 (*Reconfiguration Costs by Following Predicted Changes in the Context*). — Following the sequence of contextual changes listed in Table 7.1, a different sequence of reconfigurations emerges if one estimates future contextual changes w.r.t. previously monitored frequency of contextual changes. This frequency of contextual changes is given by a probability measure for each contextual change. Based on these probabilities, an estimation of future contextual changes is derivable. Table 7.2 lists the most probable subsequent contextual changes for each contextual state. Note that an example, which investigates all possible paths, is to extensive for an exemplary elaboration. However, the concept remains the same and alternative paths with a lower probability are derivable analogously by using the probability distributions depicted in Figure 7.4 and cost annotations depicted in Figure 7.3.

Source State	Source Context	Target Context	Probability	Target State	Costs
s_1	{Slow Network}	{Home, Slow Network}	$\frac{4}{6}$	s_5	7 + 6
s_5	{Home, Slow Network}	{Home}	$\frac{33}{60}$	s_7	4 + 3
s_7	{Home}	{Home, Slow Network}	$\frac{11}{15}$	s_4	3 + 5

Overall Cost: 28

Table 7.2: Adaptation Path With Prediction

The overall probability $\mathbb{P}$ of the path π utilized by a run r =({Slow Network}, {Home, Slow Network}), ({Home, Slow Network}, {Home}), ({Home}, {Home, Slow Network}) corresponds to

$$\begin{aligned}\mathbb{P}(\pi) &= \rho_I(q_0)\prod_{i=1}^{3}\delta_Q(q_{i-1})(q_i)\\ &= \rho_I(\{\texttt{Slow Network}\}) \cdot \delta_Q(\{\texttt{Slow Network}\})(\{\texttt{Home, Slow Network}\})\\ &\quad \cdot \delta_Q(\{\texttt{Home, Slow Network}\})(\{\texttt{Home}\})\\ &\quad \cdot \delta_Q(\{\texttt{Home}\})(\{\texttt{Home, Slow Network}\})\\ &= \tfrac{1}{5}\cdot\tfrac{4}{6}\cdot\tfrac{33}{60}\cdot\tfrac{11}{15}\\ &= \tfrac{121}{225}\end{aligned}$$

if we assume an equally distributed initial probability distribution of $\frac{1}{5}$ for each state depicted in Figure 7.4.

For example, the configuration s_7 is chosen instead of s_9 for the run of contextual changes ({`Slow Network`}, {`Home`, `Slow Network`}), ({`Home`, `Slow Network`}, {`Home`}) based on the assumption that the most probable upcoming contextual change will be ({`Home`}, {`Home`, `Slow Network`}).

Although it is more expensive to choose s_7 instead of s_9 in the first place, the configuration s_7 presumably constitutes a cheaper choice on a long-term basis depending on the significance of mobility patterns observed until now. In this example, the overall costs are reduced from 30 to 28.

In a $\mathcal{PWA}$, every state $u = (s, q) \in \mathcal{U}$ corresponds to a configuration state $s \in \mathcal{S}$ *and* a contextual state $q \in \mathcal{Q}$. Every transition is labeled with a contextual change $\sigma \in \Sigma_{\mathcal{U}} := (q, q')$. Further, a probability distribution function $\delta_{\mathcal{U}}$ assigns a certain probability to each transition and a weight function $\omega_{\mathcal{U}}$ assigns a weight to every transition. An initial probability distribution $\rho_I \in \mathcal{D}(\mathcal{U})$ assigns a probability to every state to be initially active.

Definition 7.5 (*Probabilistic Weighted Automaton ($\mathcal{PWA}$) [CDH09]*)**.** A *probabilistic weighted automaton* is a 5-tuple $(\mathcal{U}, \rho_I', \Sigma_{\mathcal{U}}, \delta_{\mathcal{U}}, \omega_{\mathcal{U}})$ where

- $\mathcal{U}$ is a finite set of *contextual configuration states*,
- $\rho_I \in \mathcal{D}(\mathcal{U})$ is an *initial state probability distribution*,
- $\Sigma_{\mathcal{U}}$ is a finite set of *transition labels*,
- $\delta_{\mathcal{U}} : \mathcal{U} \to \mathcal{D}(\mathcal{U})$ is a *probabilistic transition function*, and
- $\omega_{\mathcal{U}} : \mathcal{U} \times \Sigma_{\mathcal{U}} \times \mathcal{U} \to \mathbb{Q} \cup \{\infty\}$ is a *weight function*.

Both types of states, the configurations states of $\mathcal{WA}$ and the contextual states of $\mathcal{PA}$, are subject to the configuration semantics of the context-feature model *cfm*, from which they are derived. Therefore, composition operation is required for

(1) combining configuration states $\mathcal{S}$ with those contextual states $\mathcal{Q}$, whose requirements are satisfied by that configurations, and

(2) obtaining the probability distributions of contextual changes and the costs for potential reconfigurations,

thus resulting in a *Probabilistic-Weighted Automaton*. This composition is specific for the constraints imposed by a context-feature model *cfm*, i.e., the contextual states and their respective sets of suitable configuration states. In this regard, the composition $\mathcal{WA} \oplus \mathcal{PA}$ ensures (i) valid combination of contextual states and configuration states as well as (ii) valid transitions between the states.

For (1) the composition combines contextual states $q \in \mathcal{Q}$ with configuration states $s \in \mathcal{S}$ that satisfy the requirements imposed by q (c.f. Definition 7.6 (i)). Further, for those contextual states q requiring a configuration that is unsatisfiable

w.r.t. the context-feature model constraints specified in *cfm*, a special error state u_ϵ is introduced, resembling an unsatisfiable contextual situation.

In order to deal with a valid combination of contextual states and reconfiguration states, the initial state probability distribution of $\mathcal{PA}$ is adapted for the resulting $\mathcal{PWA}$. The *initial state probability distribution* $\rho_I \in \mathcal{D}(\mathcal{U})$ is defined as the probability distribution $\rho_I(q)$ for a contextual state $q \in \mathcal{Q}$ as usual. However, the probability of a state q to be active is equally distributed over the set $\{s' \in \mathcal{S} \mid (s', q) \in \mathcal{U}\}$ of suitable configuration states, i.e.,

$$\rho(u) = \frac{\rho_I(q)}{|\{s' \in \mathcal{S} \mid (s', q) \in \mathcal{U}\}|}$$

to derive a initial probability for a composition u=(s,q) (c.f. Definition 7.6 (ii)). For example, if the initial probability for a contextual state {`Home`} is $\frac{1}{5}$ and {`Home`} is satisfiable by 6 configuration states, the composition results in 6 states and each of these states has the same probability of $\frac{1}{5} \cdot \frac{1}{6}$ to be initially active. Further, the probability of an initially erroneous state corresponds to the remaining probability that a state corresponds to a contextual situation that is *not* satisfiable.

For (2) it is necessary to combine probabilities *and* costs for the transitions in the resulting $\mathcal{PWA}$ to derive an estimation of upcoming costs w.r.t. their probability to occur. Therefore, the *probabilistic transition function* $\delta_\mathcal{Q}$ of the $\mathcal{PA}$ is adapted (c.f. Definition 7.6 (iii)) as follows. Similar to $\mathcal{PA}$, every state (s, q) has a probability distribution $\rho \in \mathcal{D}(\mathcal{U})$ that assigns a probability to every possible target state $(s', q') \in \mathcal{U}$ to denote the probability of a transition $(s, q) \to (s', q')$. However, only the frequency of contextual changes (q,q') are of relevance. Furthermore, the $\mathcal{WA}$ does not provide any information about the frequency of reconfigurations. Therefore, every single transition that is executable for a contextual state $q \in \mathcal{Q}$ is split into a set of transitions that lead to a set of composed states $(s, q) \in \mathcal{U}$. Each transition probability for (q, q') is normalized by the number k of states satisfying the requirements imposed by the contextual state q', with k=$|\{s'' \in \mathcal{S} \mid (s'', q') \in \mathcal{U}\}|$. Thus, the probability of a transition (s,q)→($s'q'$) is normalized as follows

$$\delta_\mathcal{U}(s, q)(s', q') = \frac{\delta_\mathcal{Q}(q)(q')}{|\{s'' \in \mathcal{S} \mid (s'', q') \in \mathcal{U}\}|}.$$

For example, the contextual change ({`Home`},{`Home, Slow Network`}) has a probability of $\frac{11}{15}$. Since the target contextual situation is satisfiable by 4 configuration states, this transition is split up into four transitions and each of these transitions has the same probability of $\frac{11}{15} \cdot \frac{1}{4}$ to be executed. However, if the target contextual state $q' \in \mathcal{Q}$ is not satisfiable w.r.t. *cfm* the probability distribution is *not* split up since such transitions always lead to the error state u_ϵ.

The weight function $\omega_\mathcal{U}$ provides the costs for a transition from a source state u=(s,q) to a target state u'=(s',q') for a contextual change $\sigma = (q, q')$ (c.f. Definition 7.6 (iv)). The corresponding weights of a transition are inherited from the reconfiguration transitions in *WA*, i.e.,

$$\omega_\mathcal{U}((s, q), \sigma, (s', q')) = \omega_\mathcal{S}(s, s').$$

The *WA* provides the required weights to denote the costs of every reconfiguration $s \to s'$. For example, let us assume that s_7 and the contextual state {Home} are currently active. In the $\mathcal{PWA}$, there is only one valid transition executable to the state s_4 utilizing costs of 3 for the contextual change ({Home, Slow Network}).

The finite set of transition labels $\sigma \in \Sigma_{\mathcal{U}}$ denotes possible contextual changes $\sigma = (q, q')$ (c.f. Definition 7.6 (v)). Such a contextual change may possibly lead to multiple transitions from a single source state (*s*,*q*) to multiple target states $\{s'' \in \mathcal{S} \mid (s'', q') \in \mathcal{U}\}$. Furthermore, a contextual change $\sigma = (q, q')$ may yield multiple paths $(s, q) \Rightarrow (s', q')$ *of reconfiguration transitions* to reach a suitable target state (s', q'). For example, let us assume that the state s_3 is currently active and the contextual change ({Home, Slow Network}, {Home}) occurs. This contextual change yields the path (s_3, {Home, Slow Network})→(s_4, {Home, Slow Network})→(s_7, {Home}). In such a case, I write

$$(s, q) \Rightarrow (s'q')$$

to denote paths from a source state *u*=(*s*,*q*) to a target state *u'*=(*s'*,*q'*) for short.

A *run* $r \in \Sigma_{\mathcal{U}}^*$ of contextual changes yields a set of paths. In such a case, I write

$$\Pi_{\mathcal{PWA}}(r)$$

to denote the set of paths utilized by a contextual run $r \in \Sigma_{\mathcal{U}}^*$.

Summarizing, the composition operation of a $\mathcal{WA}$ and a $\mathcal{PA}$ that preserves the configuration semantics imposed by the respective context-feature model *cfm* is defined as follows.

Definition 7.6 (*Composition Operation for* cfm*-based Automata*)**.** Given a weighted automaton $\mathcal{WA}$, a probabilistic automaton $\mathcal{PA}$, and a corresponding context-feature model *cfm*. The composition of both automata $\mathcal{PWA} := \mathcal{WA} \oplus \mathcal{PA}$ is derived w.r.t. the following five rules

(i) $\mathcal{U} = \{(s, q) \in \mathcal{S} \times \mathcal{Q} \mid s \sqsubseteq q\} \cup \{u_\epsilon\}$,

(ii) $\rho_I \in \mathcal{D}(\mathcal{U}) = \begin{cases} \rho_I(u) = \frac{\rho_I(q)}{|\{s' \in \mathcal{S} \mid (s', q) \in \mathcal{U}\}|} & \text{for a valid state } u=(s,q) \text{ or} \\ \rho_I(u) = 1 - \sum_{(s,q) \in \mathcal{U}} \rho'_I(s, q) & \text{for an error state } u_\epsilon, \end{cases}$

(iii) $\delta_{\mathcal{U}} = \begin{cases} \delta_{\mathcal{U}}(s, q)(s', q') = \frac{\delta(q)(q')}{|\{s'' \in \mathcal{S} \mid (s'', q') \in \mathcal{U}\}|} & \text{if } q' \models \mathit{cfm}, \text{ or} \\ \delta_{\mathcal{U}}(s, q)(u_\epsilon) \quad = \delta(q)(q') & \text{if } q' \not\models \mathit{cfm}, \end{cases}$

(iv) $\omega_{\mathcal{U}}((s, q), \sigma, (s', q')) = \omega(s, s')$, and

(v) $\Sigma_{\mathcal{U}} = \mathcal{Q} \times \mathcal{Q}$, transition labels are inherited from $\mathcal{PA}$, with $\sigma = (q, q')$.

As previously explained it is safe to assume that the configuration states in $\mathcal{PWA}$ are fully path-connected, e.g., due to hand-over protocols between features. Thus, a transition between two configuration states *s* and *s'* may be restricted

from being executed. In such a case it holds that $\omega_{\mathcal{U}}((s,q),(q,q'))(s',q') = \infty$. Furthermore, I assume that contextual states q' with requirements that are not satisfiable by a context-feature model *cfm* constantly lead to the special error state u_ϵ. For convenience, the transition $u_\epsilon \to u_\epsilon$ is considered to have the probability 1, i.e., once the system enters the error state u_ϵ, the system remains in that state. As previously explained, it seems reasonable for an SAS to assume that $\mathcal{WA}$ is *fully path-connected*. Note that the error state s_ϵ that has only a self-transition as an outgoing transition, represents an exception to that assumption. In such a case error-handling strategies may be applied such as a manual restart of the system.

Whether a state $u \in \mathcal{U}$ corresponds to a valid composition of a contextual state $q \in \mathcal{Q}$ and a configuration state $s \in \mathcal{S}$ depends on the constraints imposed by the respective context-feature model *cfm*. For the composition $\mathcal{WA} \oplus \mathcal{PA}$ to be correct, every path for a contextual change $q \to q'$ has to result in a target state s', which corresponds to a refinement $s' \sqsubseteq q'$ of q'. Such a valid transition always has a probability $\delta_{\mathcal{U}}((s,q),(q,q'))(s',q')$ greater than 0. Summarizing, I introduce the following theorem for the *correct* construction of $\mathcal{PWA}$.

Theorem 7.1. Let $\mathcal{PWA} = \mathcal{WA} \oplus \mathcal{PA}$ be a probabilistic weighted automaton. Then for each $(q,q') \in \Sigma_{\mathcal{U}}$ and $(s,q) \in \mathcal{U}$ the following property holds

$$\delta_{\mathcal{U}}(s,q)(s',q') > 0$$

iff $(s,q) \neq u_\epsilon$ and $s' \sqsubseteq q'$ hold.

Proof. Theorem 7.1 is ensured by rules (i) and (iii) of Definition 7.6. The first rule (i) of Definition 7.6 allows only states $u = (s,q) \in \mathcal{U}$ to be composed by a configuration state $s \in \mathcal{S}$ and contextual state $q \in \mathcal{Q}$ iff this composition does not contradict the constraints imposed by a context-feature model *cfm*. Thus, every possible composition of states and context that do contradict *cfm*, i.e., $(s,q) \in \mathcal{S} \times \mathcal{Q} \mid s \not\sqsubseteq q$, yield an error state $u_\epsilon \in \mathcal{U}$.

Considering that both automata $\mathcal{WA}$ and $\mathcal{PA}$ are fully-path connected the third rule (iii) of Definition 7.6 ensures that a path exists from a source state (s,q) to a target state (s',q'). More precisely, for every contextual change (q,q') occurring in a source state $(s,q) \neq u_\epsilon$ there exists a path of consecutive transitions to a valid target state (s',q'), which has an overall probability larger than 0. More precisely, the transition probability from the contextual change $q \to q'$ is equally distributed over every valid composition of configuration states $S \subseteq \mathcal{S}$ validly refining the target contextual state q', i.e., $\forall s \in S : s \sqsubseteq q'$ holds. Thus, every contextual change that is restricted from occurring results in an error state $u_\epsilon \in \mathcal{U}$. □

The next section introduces a planning algorithm that uses a $\mathcal{PWA}$ as a basis to identify a target state for a contextual state, which is estimated to be the cheapest choice on a long-term basis. Therefore, the algorithm analyzes the actual costs of the current reconfiguration choices and the estimated costs of future contextual changes.

7.3 Reconfiguration Planning Algorithm

This section discusses a planning algorithm for choosing a presumably cost-optimal reconfiguration from a set of candidate configurations on the basis of the automata $\mathcal{WA}$, $\mathcal{PA}$, and $\mathcal{PWA}$, as introduced in the previous section. The algorithm is inspired by the model-based technique of *Planning as Model Checking*. By using a probabilistic weighted automaton $\mathcal{PWA}$ as a planning model, a reconfiguration (path) $s \Rightarrow s'$ is computed for satisfying a contextual change (q, q') that constitutes an appropriate trade-off between

(1) minimal actual reconfiguration costs for the current reconfiguration between configuration states $s \Rightarrow s'$, and

(2) minimal estimated reconfiguration costs of future reconfigurations for predicted subsequent contextual changes (q', q'') in a contextual run $r \in \Sigma_Q^*$.

Overview. The embedding of the planning algorithm in the overall reconfiguration life cycle of an SAS is depicted in Figure 7.5. Whenever the mobile device detects a contextual change (01) the *reconfiguration planning algorithm* is triggered w.r.t. the monitored contextual change (*q*,*q'*). The algorithm derives a suitable target state s' for the target contextual situation q' that is estimated to be the cheapest target configuration on a long-term basis. To derive this reconfiguration choice $s \rightarrow s'$ the algorithm relies on a $\mathcal{PWA}$ composed of a $\mathcal{WA}$ and $\mathcal{PA}$. This choice of reconfiguration $s \rightarrow s'$ is used to reconfigure the device (02) as well as to update $\mathcal{WA}$ to the currently activated configuration state s'. The contextual change (*q*,*q'*) is used to update the probability distribution $\delta_Q(q) \in \mathcal{D}(Q)$ of the source contextual state q in $\mathcal{PA}$ (03) w.r.t. the executed change to the target contextual state q'. Finally, $\mathcal{PWA} = \mathcal{WA} \oplus \mathcal{PA}$ is updated by re-composing the updated $\mathcal{WA}$ and $\mathcal{PA}$ (04), before another contextual change is being processed.

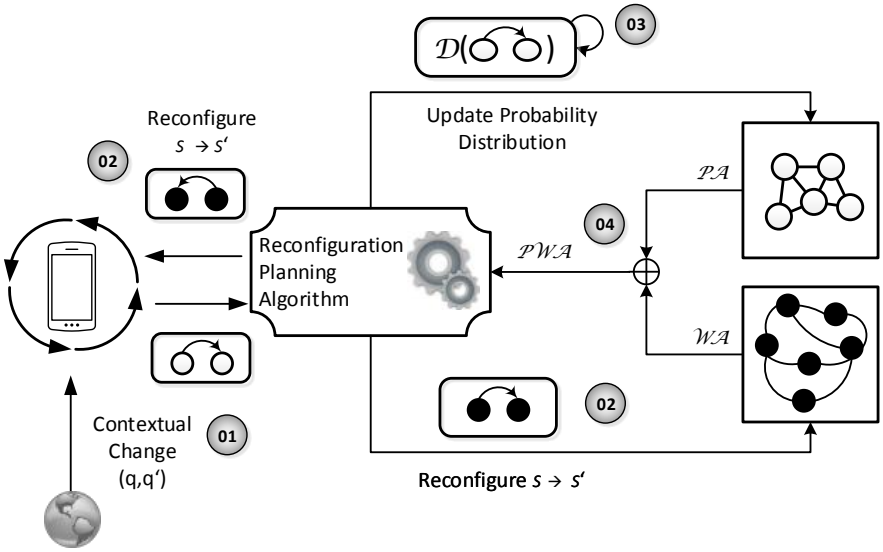


Figure 7.5: Conceptual Life Cycle of the Planning Algorithm

Algorithm 4 Reconfiguration Planning with Cost Prediction

Input: $(\mathcal{WA}, s)$; $(\mathcal{PA}, q)$; $\sigma = (q, q')$; m; m'
Output: $(\mathcal{WA}, s')$; $(\mathcal{PA}', q')$
Init: $\mathcal{PWA} := \mathcal{WA} \oplus \mathcal{PA}$; $u' = u_\epsilon$;
    $acost := ecost := cost := 0$; $cost_{min} := \infty$

for all $\pi := (s, q) \Rightarrow (s', q') \in \Pi_{\mathcal{PWA}}$ where $|\pi| \leqslant m$ **do**
    if $(\mathbb{V}(\pi) < \infty)$ **then**
        $acost := \mathbb{V}(\pi)$
        $\rho'_I(s', q') := 1$
        $ecost := 0$
        for all $(r \in \Sigma_Q^*$ where $|r| \leqslant m')$ **do**
            for all $\pi' := (s', q') \Rightarrow (s'', q'') \in \Pi_{\mathcal{PWA}}(r)$ **do**
                $ecost := ecost \ + \ \mathbb{V}(\pi') \cdot \mathbb{P}(\pi')$
            end for
        end for
        $cost := acost \ + \ ecost$
        if $(cost < cost_{min})$ **then**
            $u' := (s', q')$
            $cost_{min} := cost$
        end if
    end if
end for
$\mathcal{PA}' :=$ **evolve**$(\mathcal{PA}, \sigma)$
if $u' \neq u_\epsilon$ **then**
    return $((\mathcal{WA}, s'), (\mathcal{PA}', q'))$
else
    return *error*
end if

Since the algorithm is designed according to model checking techniques, the algorithm explores every suitable reconfiguration path for the immanent contextual change (q,q') and every path $\pi' \in \Pi_{\mathcal{PWA}}(r)$ for future contextual runs $r \in \Sigma_Q^*$. In order to control the computational utilization of the algorithm, the maximum length of reconfiguration paths $s \Rightarrow s'$ is limited by the bound m. Additionally, the prediction of a run of future contextual changes $r \in \Sigma_Q^*$ is limited by the bound m'. Note that with such an restriction on the length of an investigated path, the results become *incomplete*. If a reconfiguration path $\pi = (s \Rightarrow s')$ for a contextual change (q,q') is longer than the bound m, i.e., $|\pi| > m$, then the algorithm ignores that path π as a possible reconfiguration choice. Therefore, the algorithm evaluates its choice for the cheapest reconfiguration path on the basis of an incomplete set of reconfiguration paths.

Analogously, this incompleteness holds for the investigation of future runs of contextual changes $r \in \Sigma_Q^* = ((q',q''), \cdots, (q_{n-1}, q_n))$. The bound m' provides an estimation over all upcoming contextual changes up to a limit of m', i.e., $|r| \leqslant m'$. Therefore, the cost expectation becomes incomplete.

Algorithm 4 summarizes the computation of a reconfiguration step for a contextual change (q,q'). The algorithm requires the recent configuration state s of the weighted automaton $\mathcal{WA}$ and the current contextual state q of the probabilistic automaton $\mathcal{PA}$ as input. The algorithm considers the contextual change $\sigma = (q, q')$ (line 1) as basis to derive a cost-optimal configuration state for q'. The Parameter $m > 0$ defines the bound for the reconfiguration path to reach a target configuration state. The parameter $m' \geqslant 0$ defines the bound for the length of predicted contextual runs. As output, the target configuration state s' computed for reconfiguring the device to the new contextual requirements of state q' is returned. Further, the probabilistic automaton is evolved to $\mathcal{PA}'$ by updating the probability distribution, i.e., increasing the probability distribution for the contextual change (q, q').

First, the probabilistic weighted automaton $\mathcal{PWA}$ is composed according to Definition 7.6 and variables for intermediate cost values are initialized (line 3). The computation of the presumably cost-optimal reconfiguration $s \Rightarrow s'$ for (q, q') consists of two nested loops.

(i) The outer loop (line 5–21) investigates all reconfiguration paths $s \Rightarrow s'$ of up to a length of m, with s' satisfying the new contextual requirements imposed by q'.

(ii) The nested inner loop (line 10–14) investigates all possible reconfiguration paths utilized by the possible contextual runs $r \in \Sigma_Q^*$ up to an overall number of m' subsequent contextual changes. To derive an estimation of future emerging reconfiguration costs *for each* reconfiguration candidate s' investigated in (i), the algorithm quantifies the costs utilized by each path with the probability of the respective run of contextual changes.

Both steps are automatable by applying a model checker that is able to process quantified and probabilistic properties, such as PRISM [GPS13]. However, using a model checker to derive a probabilistic cost estimation does not resemble the standard use-case of a model checker, i.e., the *verification* of system properties. Therefore, the respective technique to tackle specific segments of the algorithm via model checker queries is provided in addition to the discussion of the algorithm.

In a final step, the cheapest target state s' is identified w.r.t. the computed cost estimations of future contextual changes and the probability distributions of $\mathcal{PA}$ are updated accordingly.

RECONFIGURATION COSTS FOR A CONTEXTUAL CHANGE. For step (i), *Planning as Model Checking* is applied as usual to find all reconfiguration paths satisfying the contextual requirements and to aggregate their reconfiguration costs into the variable *acost*. In this step, transition probabilities are neglected as the reconfiguration is determined by the contextual change $q \to q'$ (line 7). Thus, the costs utilized by a suitable reconfiguration path $\pi = s \Rightarrow s' \in \Pi_{\mathcal{PWA}}$ are computed by the summation function $\mathbb{V}$, i.e, the sum of all weights occurring on π. The investigated reconfiguration paths $\pi \in \Pi_{\mathcal{PWA}}$ are limited by the parameter m, i.e., the sequence of reconfigurations $s \Rightarrow s'$ for the contextual change (q, q') has to be equals to or smaller than m.

The respective model checker query would be composed of two aspects. The first aspect would compute all suitable states $\Pi_{\mathcal{PWA}}(r)$, i.e. "*compute all paths that eventually lead to a state in which* $s' \sqsubseteq q'$ *holds with an upper bound of* m". The second aspect would compute a summation of the weights occurring computed on paths $\pi \in \Pi_{\mathcal{PWA}}(r)$ derived by the previous query.

PREDICTED RECONFIGURATION COSTS OF FUTURE CONTEXTUAL CHANGES. In contrast to step (i), in step (ii), the costs for predicted future contextual changes are derived by computing an estimation of future emerging costs utilized by reconfigurations and states. Therefore, the costs $\mathbb{V}(\pi')$ of every path $\pi' \in \Pi_{\mathcal{PWA}}(r)$ utilized by every possible run of contextual changes $r \in \Sigma_Q^*$ originating from the current initial state (s',q') is multiplied by the probability of the path $\mathbb{P}(\pi)$ (line 11–13) and stored in the variable *ecost*. This computation of the estimated costs in *ecost* reflect the fact that costs utilized by reconfigurations for contextual changes, which have a higher probability to occur, are more important (line 12). The estimated costs *ecost* are computed for *every* suitable target state $s' \in S \subseteq \mathcal{S}$ for the contextual change (q, q') investigated in step (i).

Furthermore, the sum of estimated costs *ecost* over every $\pi' \in \Pi_{\mathcal{PWA}}(r)$ and every possible run $r \in \Sigma_Q^*$ is derived to compute the cost estimation for suitable target states (s',q') for the investigated contextual change $\sigma = (q, q')$ (line 15).

Existing model checkers, such as PRISM[1] [GPS13], are currently not capable to support a query that combines probabilistic and weighted properties of transitions, as required by my approach. One of the resulting drawbacks is that PRISM supports only weights of a state and not of a transition. However, PRISM is applicable to derive an estimation of upcoming reconfiguration costs by adding an additional variable *acost* to the system model that tracks the costs of an executed transition. The range of possible values for *acost* directly influences resource utilization of PRISM. Since all possible states of the system model are to be explored by PRISM every existing state has to be combined with every possible value of *acost*. This increases the executed state exploration and, therefore, the overall processing time and resource utilization. If such an additional variable is used, a query may be used such as "*compute the expected costs of acost for all possible paths with m' as an upper bound*". Note that PRISM is capable to compute an expected cost value for a path by multiplying the probability of a transition with the utilized costs denoted in *acost*.

COST AGGREGATION AND AUTOMATON EVOLUTION. Both, the actual reconfiguration costs computed in step (i) and the estimated future reconfiguration costs computed in step (ii) are, again, aggregated into an overall cost *cost* value (line 15). Finally, this cost value allows to determine the presumably cost-optimal reconfiguration choice for a suitable target configuration state s' (line 16–19).

The probabilistic automaton $\mathcal{PA}$ is evolved by increasing the probability of the contextual change (q, q'), accordingly (line 22). This way, although a self-transition $s \to s$ (idling) may arise as one possible reconfiguration option with low reconfiguration costs, another reconfiguration $s \to s'$ with $s \neq s'$ may be preferred.

[1] Examined version of PRISM: `4.1.beta2`

Such a reconfiguration $s \rightarrow s'$ may be preferred to a self-transition if the target configuration state s' is the cheaper reconfiguration choice for future contextual changes.

Note that on system start-up, the weighted automaton $\mathcal{WA}$ as well as the probabilistic automaton $\mathcal{PA}$ reside in some initial state s_0 and q_0, respectively, i.e., $\rho_I(q_0) = 1$, $\rho_I(q_{>0}) = 0$, and as initial transition probability distribution $\mathcal{D}(\mathcal{U})$ of $\mathcal{PA}$, e.g., an uniform distribution may be considered.

At runtime, two possible failure cases might arise

- the contextual requirements of q' may not be satisfiable by a context-feature model *cfm* and
- no satisfying configuration s' is reachable from state s within bound m.

Both cases lead to an error state u_ϵ (line 23–27). In such a case, additional error handling strategies may be required, such as resetting the system or reverting the current configuration sate to a non-erroneous state. Note that in case of $\mathcal{WA}$ has a fully-connected configuration state-transition graph and setting $m' = 1$, the algorithm just selects the cheapest immediate reconfiguration as usual.

In the next section, an evaluation of the planning approach on the basis of a sample implementation of the described algorithm is discussed.

7.4 EVALUATION

In the following, I provide a proof-of-concept evaluation of the introduced algorithm regarding its ability to reduce costs on a long-term basis. Therefore, I analyze

(1) the cost-probability distribution for a run of contextual changes,

(2) effectiveness of increasing the number of the investigated future contextual changes m', and

(3) the overall benefit of applying the reconfiguration planning algorithm over several contextual changes,

before I discuss the limitations of the simulative evaluation. The random probabilistic automaton $\mathcal{PA}$ depicted in Figure 7.6 is used as a basis for the evaluation. The automaton is similar to the automaton used as a running example in this chapter. However, the transitions, probabilities and compatibility to configuration states differ.

Furthermore, a random weighted automaton $\mathcal{WA}$ is given by the state-transition matrix listed in Table 7.3. The first column of this table denotes the source configuration states and the upper row denotes the target configuration states. A number in a cell denotes the reconfiguration costs between the two respective configuration states. For example, the reconfiguration $c_1 \rightarrow c_0$ utilizes costs of 6. The infinity ∞ symbol denotes that a transition is restricted from being executed. For example, the reconfiguration $c_0 \rightarrow c_3$ is not allowed to be executed.

The evaluation starts in the source state (s_3,c_0), which is denoted with a gray state in Figure 7.6 and a gray row in Table 7.3. If the reconfiguration planning

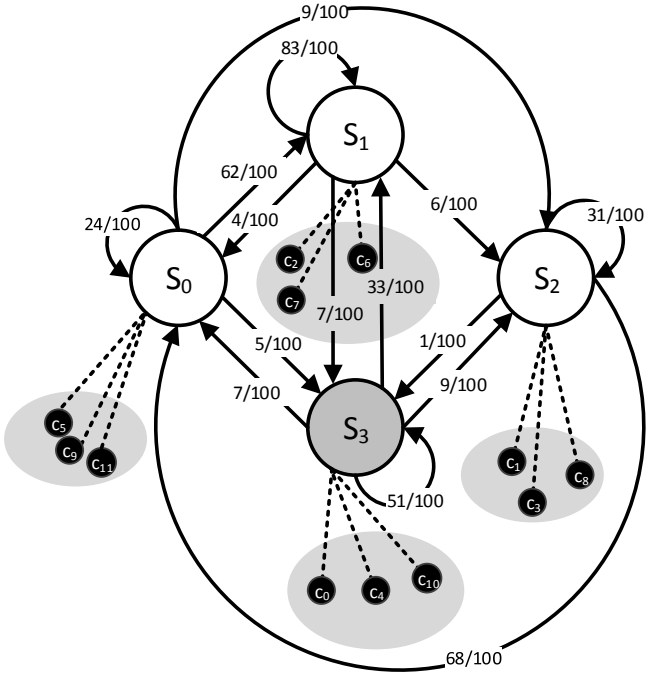


Figure 7.6: $\mathcal{PA}$ Simulation Model

target → source ↓	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
c_0	∞	7	6	∞	9	∞	∞	6	∞	∞	10	2
c_1	6	∞	8	8	∞	10	∞	3	∞	∞	8	6
c_2	∞	7	∞	∞	3	4	6	∞	∞	∞	∞	∞
c_3	10	∞	∞	∞	∞	7	∞	6	6	6	8	∞
c_4	∞	7	∞	∞	∞	∞	10	9	∞	3	10	∞
c_5	10	∞	1	∞	∞	∞	∞	1	10	10	∞	4
c_6	∞	7	2	∞	∞	∞	∞	∞	∞	∞	9	7
c_7	∞	9	3	∞	∞	6	5	∞	∞	∞	6	7
c_8	7	7	8	4	10	∞	∞	∞	∞	10	∞	10
c_9	∞	∞	7	6	4	2	∞	1	∞	∞	∞	10
c_{10}	∞	6	10	∞	7	8	∞	∞	8	∞	∞	∞
c_{11}	∞	∞	∞	∞	∞	2	3	∞	2	∞	6	∞

Table 7.3: Transition Matrix for $\mathcal{WA}$ Simulation Model

algorithm is parameterized with m=1, the two the configuration states c_2 and c_7 are the only valid target configuration states for the contextual change from s_3 to s_1. Therefore, those two target states are highlighted in a dark gray in the state-transition matrix.

This simulation setup provides insights into the process, which of those two states c_2 and c_7 is to be chosen for the contextual change s_3 to s_1 based on the Reconfiguration Planning Algorithm 4 discussed in the previous chapter. The algorithm is initialized with the $\mathcal{PA}$ depicted in Figure 7.6 and the $\mathcal{WA}$ listed in Table 7.3. Furthermore, the possible paths for a reconfiguration are limited to 1 with m=1, whereas the length of the investigated future contextual changes m' is evaluated with different parameter settings.

7.4.1 *Comparison of Cost Prediction*

In the following evaluation, I assume $\mathcal{WA}$ to reside in the initial configuration state c_0 and $\mathcal{PA}$ to reside in s_3. A contextual change from s_3 to s_1 occurs and the device has to reconfigure itself accordingly.

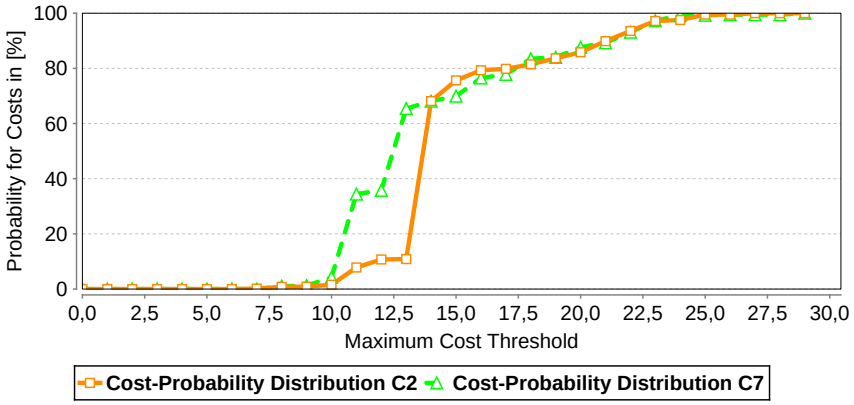


Figure 7.7: Cost-Probability Distribution

To choose a suitable configuration for s_1 the Algorithm 4 is used with a parameter setting of m=1 and m'=3. Figure 7.7 depicts the probabilities that a certain cost limit is kept by each configuration state. This probability distribution of utilized costs allows for a comparison of the expected cost-development after choosing one of the possible configuration states c_2 and c_7. For example, 10% of the estimated costs for predicted subsequent reconfigurations remain below a cost limit of 12.3 if the reconfiguration is executed to c_2. In contrast to that 37% of the predicted subsequent reconfigurations remain below a cost limit of 11 if c_7 is chosen as a target state. Thus, although c_2 and c_7 utilize equal costs of 6 for a reconfiguration from c_0, the reconfiguration planning algorithm chooses c_7 as the next configuration state for the contextual change from s_3 to s_1. The reason for this is that c_7 utilizes fewer costs and on a long-term basis.

This plot illustrates the possibilities how an overall value of a cost estimation may be derived. For example, the costs may be multiplied with their probability to occur. The sum of every probabilistic cost value corresponds to the overall cost estimation for a state. A similar method is also applied in Algorithm 4 (c.f. line

12). Based on the cost-probability distribution of both states depicted in Figure 7.8, an overall estimation of 15.36 for c_2 and 14.34 for c_7 is derivable.

The next section discusses the difference in the estimated costs between the two target states c_2 and c_7 across different settings for m'.

7.4.2 *Length of a Contextual Run*

The parameter m' denotes the upper bound in the length of future upcoming contextual changes. The limitation of the length of an investigated path has a direct influence on the computational efforts of the path exploration as well as on the significance and accuracy of the results. Thus, to be efficient an appropriate m' must be chosen. Figure 7.8 depicts the difference in the cost estimation between c_2 and c_7. An evaluation of m'=1 is skipped because in this case the state with the cheapest reconfiguration is chosen and, therefore, no prediction is used. The plot indicates that with m'=3 the estimated costs between c_2 and c_7 differ by 1.02. Thus, the configuration state c_7 probably causes 1.02 less costs than c_2 across the next 3 contextual changes. With an increase of m' the difference in the estimated costs seem to converge to 0.6 instead of the difference of 1 as estimated with m'=3.

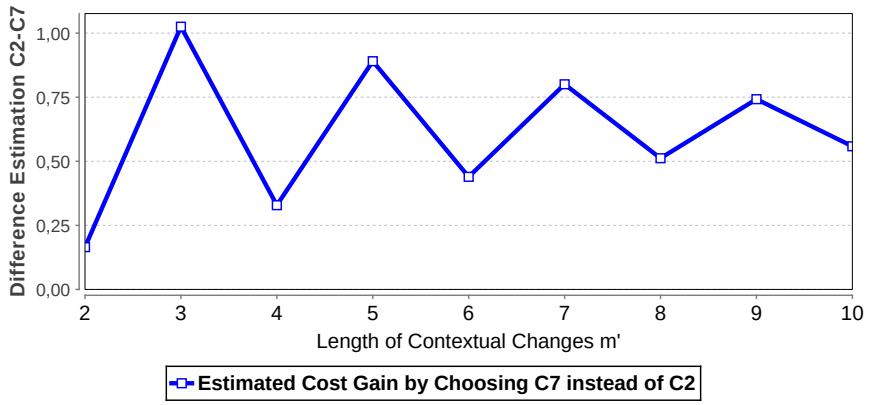


Figure 7.8: Delta in Estimated Costs between c_2 and c_7 across m'

This plot provides insights in the impact of the investigated length of contextual changes m'. As previously, explained the setting of m' implies an *incompleteness* of the investigated paths. The plot indicates, that a greater setting of m' increases the accuracy and at the end the estimation has the tendency to converge to a single value.

For this evaluation, two things have to be considered.

- The investigated $\mathcal{PWA}$ is a small example (11 configuration states and 4 contextual states). If the $\mathcal{PWA}$ increases in its number of states and transitions more paths are to be investigated by the algorithm. Therefore, a greater setting of m' is necessary to recognize a convergence in the computed cost estimation.

- In the investigated example c_7 is *always* the better choice in comparison to c_2. However, the difference may (i) be larger than 1 and (ii) be negative. If

the difference becomes negative in one specific setting of m' c_2 would be chosen instead of c_7.

Thus, a wrong setting of m' may lead to a decision that impacts the overall results negatively on a long-term basis. Furthermore, the setting of m' depends on the characteristics of the investigated probabilistic weighted automaton.

The next section discusses the possible savings in costs by applying the reconfiguration planning algorithm with a fixed m' of 4.

7.4.3 *Possible Costs Savings across Contextual Changes*

Figure 7.9 depicts a comparison of possible reductions in the costs utilized by a sequence of reconfigurations triggered by 100 contextual changes. The figure depicts one plot, in which the contextual change events occurred according to the probability distribution depicted in Figure 7.6, and one plot, in which the contextual changes occur randomly. The horizontal line at 0 represents the baseline of comparison, i.e., the costs, which are utilized if the Reconfiguration Planning Algorithm 4 is *not* used.

The plot *Changes as Predicted* illustrates that it is possible to reduce the sum of the reconfigurations costs utilized after 100 contextual changes by 87 in comparison to an approach that chooses constantly the cheapest reconfiguration. However, the plot *Random Changes* indicates that such an extent of reduction is only achievable iff the contextual changes occur within the probability distribution specified in the respective probabilistic automaton $\mathcal{PA}$. Random contextual changes diminish the possibility to reduce the reconfiguration costs because the predicted contextual behavior does not meet the contextual behavior, which actually occurs.

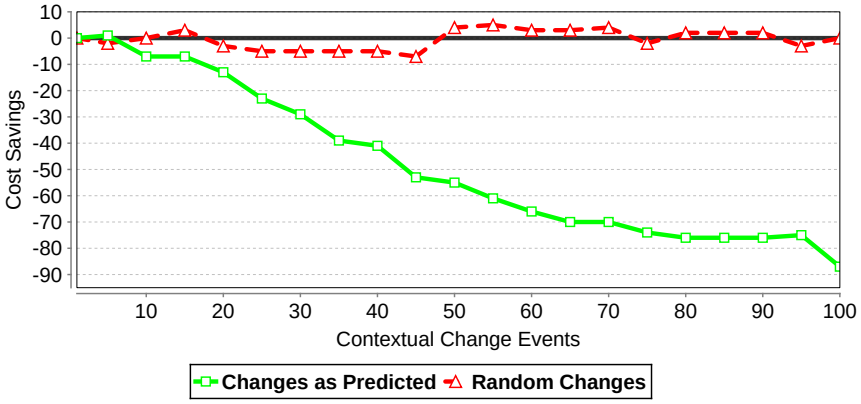


Figure 7.9: Possible Cost Reduction Across Contextual Changes

This evaluation illustrates the importance that a user behaves according to the previously tracked contextual changes. Sudden changes in the behavior lead to configurations that are less likely to occur w.r.t. the probabilistic history of contextual changes. Thus, if a user is taking a vacation the approach may possibly lead to an overhead in the reconfiguration costs instead of savings. Additional overhead in the reconfiguration costs may be generated until either (i) the user follows

again the pattern of previously tracked contextual changes or (ii) the probability distributions $\mathcal{PA}$ are finally adapted to the new behavior of contextual changes.

This evaluation of possible cost savings as well as the previously discussed evaluations rely on a single simulation model. The next section discusses such limitations of the investigated proof-of-concept evaluation.

7.4.4 *Limitations*

The reconfiguration planning algorithm is able to reduce the overall costs utilized at runtime on a long-term basis (c.f. Section 7.4.3). Furthermore, although the investigated path of contextual changes is always limited by the parameter m', the computed cost prediction has the tendency to converge to a single estimation value (c.f. Section 7.4.2).

A quantitative evaluation may never be complete and it is only natural that the presented evaluation has a number of limitations. The measurement setups of my proof-of-concept evaluation have been limited to (i) a single small-scale combination of a $\mathcal{PA}$ and a $\mathcal{WA}$, (ii) a random assignment of costs, (iii) a random initial probability distribution of contextual changes, and (iv) a random topology in the state-transition automata $\mathcal{PA}$ and $\mathcal{WA}$.

The most important and obvious restriction is that the quantitative assessment of my approach is based on a small-scale example consisting of 4 contextual states and 11 configuration states. The evaluation is restricted to such small automata due to scalability issues in the exhaustive exploration of *all* possible paths leaving from *every* possible target state for a contextual change.

Due to the restriction to such small automata it remains questionable how representative the results are for a large scale use-case scenario. For example, my first investigation of a dataset containing the usage behavior of smartphones for multiple users [KJD+10, LGPA+12] resulted in a feature model containing at least 16 features and 10 contexts. Based on this context-feature model 111 contextual states and 270 configuration states are derivable.

In order to provide representative results for the extent in the cost reduction a representative cost model is required. For example, the (de-)activation of a WLAN-chip consumes a certain amount of energy. In addition to that, the WLAN-chip and related features consume energy while being active. However, the evaluation is based on a random assignment of costs, which range from 1 to 10 (c.f. Table 7.3).

The evaluation shows that the investigated length of future contextual changes m' is important for choosing the cheapest target configuration state on a long-term basis. Although it has been shown that the algorithm is capable to identify the cheapest configuration state, neither an overall best parameterization of m' nor guidance criteria to derive such a parameterization can be determined due to the limitations discussed above.

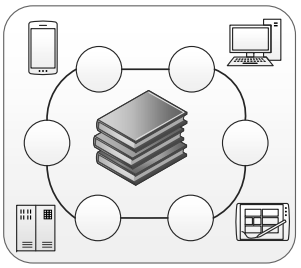
The reduction of reconfiguration costs on a long-term basis w.r.t. predicted upcoming contextual changes is the last conceptual contribution of this thesis. The next chapter discusses related concepts and approaches to the contributions provided in this thesis.

Part IV

CONCLUSIONS

8

DISCUSSION OF RELATED WORK



The concepts and techniques provided in the previous chapters of this thesis tackle well-known research challenges in the domain of self-adaptive software systems (SAS) and dynamic software product lines (DSPLs). Thus, it is only natural that approaches exist, which address the principal topic of this thesis. In the following, an overview of relevant and important related approaches from the different areas of (i) *model-based runtime adaptation*, (ii) *context modelling*, (iii) *refinement of feature models*, (iv) *(re-)configuration state spaces*, as well as (v) *prediction and planning of adaptations* are discussed.

The approaches are, if possible, evaluated w.r.t. the research challenges (RCs) established in Chapter 2 of this thesis, i.e., the following four RCs for SAS

RC1. Systematic Engineering Approach,

RC2. Flexibility at Runtime,

RC3. Autonomous Adaptation, and

RC4. Non-Intrusive Adaptation,

as well as the following two RCs for DSPLs

RC5. Autonomous Reconfiguration and

RC6. Improvement of Resource Consumption.

8.1 Model-Based Runtime Adaptation

The domain of adaptive software systems is well established and provides a wide spectrum of approaches and solutions. Existing approaches, which tackle *RC1* formally describe the possible adaptation behavior of a system to establish a model-based adaptation process. With such a description, a transition system is derived to coordinate or verify [CGKM12, Shi07] the adaptation process. Specific approaches also include non-functional properties for an adaptation [PS09, SHMK10, SPA12]. For example, the MUSIC framework [FFF+13, RBD09] is a representative implementation of model-based adaptation approaches to reconfigure service oriented architectures at runtime. Similar to the first goal of my thesis, they also aim to establish an autonomous, model-based adaptation process.

More specifically, a component-based approach that deals with adaptivity in decentralized ad hoc systems is proposed by Pepper et al. [PS09]. The authors address the problem of resource limitation of mobile devices in self-adaptive ad hoc networks, i.e., *RC4* and *RC5*. They provide an approach to uphold a certain quality of services. Services are composed by linking service units. Each service unit is characterized by its functionality and quality. Each service corresponds to a composition of features and each feature is characterized by its provided functionality and its attributes. The attributes are used to describe the non-functional qualitative nature of a feature, such as its stability or its performance. Dealing with quality and functional properties, the resulting matching problem to find an optimal configuration suitable for the stated requirements, i.e., best service vs. a minimal utilization of resources, is considered to be a major challenge. Pepper et al. present several challenges and possibilities how to deal with qualitative characteristics and how the derivation of a reconfiguration may be improved but they do not provide a solution.

FUSION is a framework for engineering SAS [EEM10] tackling *RC3* by implementing a feedback loop similar to MAPE-K. The framework aims to continuously learn and optimize the reconfiguration process by taking non-functional system behavior into account, such as costs or responsiveness. Similar to my approach, FUSION also relies on a feature-based adaptation process and specifies the adaptation knowledge on the basis of logic formulae. By continuously reasoning about the current system state and the current contextual situation, FUSION is even capable to adapt to contextual changes unforeseen at design time and, thereby, addressing *RC2*. However, due to the continuous reasoning at runtime, the approach is costly, which is contradictory to *RC4*. In contrast to FUSION, my approach does not incorporate learning techniques but evaluates monitored behavior to estimate cost-effective reconfiguration choices. My approach adapts to unforeseen contextual changes in one of two ways, either

- the contextual change triggers a reconfiguration of an unrestricted feature or
- a new (arbitrary) configuration state is derived on demand.

Although I aim for similar goal as the authors of [EEM10], I (i) do not try to find an optimal solution and (ii) shift the computational efforts for deriving a configuration at runtime to design-time.

Although FUSION relies on the concepts of features to describe the characteristics of a system, FUSION does not use any DSPL specific techniques. The next paragraph discusses in detail approaches that use DSPL techniques to realize an adaptive software system.

DSPL-Based Runtime Adaptation. Recent approaches propose to handle runtime adaptations on the basis of DSPL techniques [BHS12, Lee06, HSSF06]. Those approaches either derive a suitable configuration at runtime based on a feature model specification [BSBG08] or fully specify a reconfiguration automaton based on a complete configuration state space to model valid reconfigurations at runtime [WDSB09, BSBG08].

Bencomo et al. study component-based technologies for runtime adaptivity using a component model called OpenCOM [BSBG08]. OpenCOM offers a component framework for developers, to implement applications as well as middleware platforms. The adaptive behavior is specified by reconfiguration policies in the form of event-condition-action rules. An action triggers architectural changes in the component model, e.g., by re-wiring components to other components during runtime. Every possible variation of active components and their relations is called a configuration. Using context-aware monitors, OpenCOM tackles *RC5* by capturing relevant information from the environment to trigger the event-do-action rules. The authors apply their approach to wireless sensor nodes within a case study for a flood warning system. Regarding resource limited devices, the runtime behavior remains still to be unknown and, therefore, *RC6* is unaddressed. Possibilities for an optimization, like a pre-configuration step similar to my approach to reduce a feature model, are proposed but not realized.

A comprehensive formalization of DSPL (re-)configuration semantics similar to my formalization is not yet provided, to the best of my knowledge. Instead, existing approaches provide tools and concepts to

- specify DSPL feature models and transition systems individually [PRC14] or
- derive a reconfiguration transition system from a feature model specification and an additional realization-model [CGF08].

In both cases, the authors investigate either component-based systems or service-oriented architectures as an application scenario for a DSPL. Further, they do not apply their approaches to resource constraint systems such as mobile devices.

Recent approaches that focus on a DSPL (re-)configuration process analyze feature models *and* transition systems. In this regard, approaches, such as [RSAS11, BLL+14], investigate how and when a feature has to be configured to be active or inactive. Therefore, a step-wise (re-)configuration process is introduced that (de-)selects a feature at different stages in the life-cycle of a DSPL, e.g., design-time or runtime. Based on such a model, a sequence is derived in which order a feature has to be reconfigured. Furthermore, the model is analyzed in order to identify static features, i.e. non-reconfigurable features, or dead features.

Recent verification approaches for SPLs analyze feature model specifications by incrementally applying CSP, BDD, and SAT solvers in several stages [BTRCRC05, HCH09, WDSB09]. In each stage, a configuration is further refined. After a configuration-stage is finished, the feature model constraints are validated. However, those approaches aim at step-wise configuration semantics of feature models assuming interactions with stakeholders. In contrast to that, I established a continuous reconfiguration process that implements an autonomous configuration refinement w.r.t. contextual changes. Thus, such an incremental approach is intended to derive one specific configuration that remains static. Therefore, the approaches [BTRCRC05, HCH09, WDSB09] are not applicable for a dynamic reconfiguration of an SAS at runtime. Furthermore, the continuous usage of a solver in each stage consumes a lot of resources, which is not beneficial for *RC6*.

None of the approaches discussed above provide such a seamless integration of contexts into the reconfiguration process of a DSPL as context-feature models do (c.f. Section 4.2.2). However, such an integration of contexts into feature models is, to the best of my knowledge, a novel concept. Therefore, I discuss state of the art approaches for context-modelling, which are comparable to context-feature models, in the next section.

8.2 Context Modelling

There are several approaches, which integrate contextual aspects into variability modeling. Note that these approaches solely focus on the domain of context modelling and, therefore, neither of the research challenges from *RC1* to *RC6* are addressed in these approaches.

Ali et al. [ACG09] investigate how contexts influence variability of a software system. Similar to my approach of a context-aware DSPL, they identified different variation points in which the context has to be considered, e.g., at design time and at runtime. Further, Hartmann et al. [HT08] propose to combine the concept of a context and a variability model. Based on contexts, they derive multiple context specific product lines. Both approaches use goal models [VL01] in combination with feature models to reason about the functional and non-functional requirements of a specific context. In my approach, I propose to enrich a feature model with contexts to define functional requirements of contexts, denoted by require or exclude relations between the features and contexts. Thus, I separate the modeling of the variability and contextual requirements and, thereby, establish a reconfiguration approach that is primarily intended to satisfy functional requirements imposed by a context.

With a context-feature model, I organize contexts in a similar way as features are arranged in a feature model. Context modeling techniques, such as context-goal models or ontology models [ACG09, BBH+10], use similar modelling techniques that rely on a hierarchy and dependency relations.

Goal models are used to specify functional *and* non-functional requirements of a stakeholder [LM09, ACG09, VL01]. Such goals are specified in a hierarchical tree-like diagram. Every parent-goal is decomposable into logical or-sub-goals or logical and-sub-goals to introduce variability in goals. Thus, the structure of such goal models is comparable to a feature model diagram. However, in addition to feature model diagrams, goal models are capable to specify qualitative soft-goals, e.g., to minimize the energy consumption. Soft-goals do not have a clear-cut criterion for their fulfillment. Instead, soft-goals positively or negatively influence the overall goal.

Summarizing, goal models are comparable to my concept of a context-feature model, although the introduced concept of context-feature models is intended to express only functional goals. To this end, non-functional properties are neglected in my concept of a context. However, my approach may be extended to express non-functional properties by adding attributes to features that specify the non-functional properties of a feature. For example, feature `WLAN` may be extended with the attribute *cost=40* to express a resource utilization of

40. Kang et al. propose such feature attributes in their concept of extended feature models [KCH+90]. To derive a configuration w.r.t. non-functional requirements, first-order logic constraints [SMD+12] are required, e.g., $((\texttt{costs} < 50) \wedge (\texttt{stability} > 90))$. With such a model, not only a suitable configuration is derivable that satisfies the functional requirements of a context. Instead, a configuration is derivable that is *optimal* for a contextual situation w.r.t. the non-functional requirements.

Razzaque et al. [RDN06] stresses the fact that it is necessary to specify

- contextual requirements and
- dependencies amongst contexts

for the modelling of contexts. Therefore, the authors investigated existing approaches and concluded that the specification of requirements *and* dependencies are used rarely in combination. Instead, existing approaches focus on either the modelling of contextual requirements or on the modeling of dependencies amongst contexts. However, they point out that every context modelling approach fits the needs of the individual application domain. Furthermore, they derive a categorization of contexts, e.g., state of a user, state of the physical environment, or whether a context is static or not. The authors use first order logic for the specification of a context model. In contrast to that, I used only propositional logic in order to specify a context-feature model. Therefore, the context modelling approach of Razzaque et al. is more expressive in its specification possibilities of contextual characteristics and requirements.

With Proteus Toninelli et al. [TMKL07] propose a rule-based policy model that relies on an ontology model [BBH+10] to specify the characteristics of a context. The ontology model consists of a hierarchy of abstract contexts, which results in a concrete context, describing a contextual situation. Thus, in contrast to my approach, such an ontology-based modelling of contexts further allows reasoning about contextual situations, i.e., the derivation of a contextual situation based on the information captured by the system. I do not investigate techniques to reason about a contextual situation based on sensor data since context reasoning is considered to be not within the scope of this thesis. However, similar to the ontology-based approach, my approach is able to

- model context hierarchically in a context-feature model and
- derive a contextual situation consisting of several atomic contexts.

To reduce the computational effort for the derivation of a configuration w.r.t. the requirements imposed by a contextual situation, I propose a DSPL specific reduction approach of feature models that depends on a pre-configuration of features with the possibility to refine a configuration. The next section discusses related approaches on feature model slicing and a stage-based derivation of configurations.

8.3 Refinement of Feature Models

Existing approaches that aim at reducing the size of a feature model exploit the dependency relations in a feature model diagram specification. They apply so called *slicing* techniques to remove certain features w.r.t. a slicing criterion [ACLF11, RSA11]. Such *slicing* approaches are applicable to tackle *RC6*.

The authors specify a slicing criterion as a set of features that is to be removed from the feature model diagram. Furthermore, the authors also stress the importance of preserving the feature constraints of the original specification, i.e., to derive a *correct* slice of a feature model. However, they do not provide a proof of correctness, as I do. The slicing approaches provided in [ACLF11, RSA11] are *not* designed to maximize variability at runtime. In this regard, I used a three--valued logic to provide a developer with the possibility to interpret features as reconfigurable. My approach specifically reduces the propositional formula representation of a feature model, whereas other approaches conceptually argue on the level of a dependency graph, i.e., a feature model diagram.

Czarnecki et al. were the first to propose the concept of a staged configuration process based on feature models [CHE04, CHE05]. The authors argue that the derivation of a product configuration using feature models may be done in stages. In each stage, they restrict possible configuration choices by further refining the feature model configuration. In this regard, each stage results in a customized, partially configured feature model.

This process is also referred to as specialization of a feature model. The process of a staged configuration describes successive specialization steps followed by configuration steps using the most specialized feature model. A staged configuration process corresponds to incrementally applying a partial configuration until there is no more variability left and a complete configuration is derived. In contrast to a staged configuration process, my approach neither specializes nor generalizes the constraints of a feature model. My technique applies a similar concept of a specialization, i.e., a *refinement*, with the intention that *not* every feature is interpreted as selected or deselected. Such a partial configuration is transitively completed in order to provide a set of features, which are reconfigurable at runtime. This results in a smaller feature model, which is always consistent with the feature constraints of the original feature model.

With my feature model reduction approach, I aim at reducing the resource utilization of a continuous reconfiguration at runtime. Therefore, the next section discusses how related approaches handle reconfigurations at runtime.

8.4 (Re-)Configuration State Space

Recent approaches investigating the (re-)configuration of SPLs [WDSB09] and DSPLs [Hel12, DPS12] use transition systems for modelling the configuration process. Just like in this thesis, the authors interpret a product configuration as a state and transitions as reconfigurations. The transitions may be labeled and/or weighted to guide reconfiguration choices, e.g., by choosing the least expensive transition. Despite those similarities, existing transition systems for (D)SPLs are

neither intended to be tailored for resource-constrained devices nor are they dynamically evolvable at runtime. Thus, these approaches do not explicitly tackle *RC6*, since they do not scale with the complexity of a feature model and are, therefore, unqualified for resource constrained devices. Although the issue of an autonomous reconfiguration, i.e., *RC5*, is never directly addressed, these approaches are easily extendable to execute reconfigurations autonomously, e.g., by annotating a transition with a contextual-trigger.

In this section, I focus on the discussion of related approaches and how they

- handle dynamic reconfigurations or adaptations at runtime,
- step-wise refine a configuration until a valid product configuration is derived, and
- identify unrestricted features.

RECONFIGURATIONS BASED ON TRANSITION SYSTEMS. Damiani et al. [DS11, DPS12] propose an approach that relies on a transition system to define the semantics of a reconfiguration. The authors apply a technique called *delta oriented programming* to handle dynamic reconfigurations at runtime. A delta describes a change in the code basis of the currently active configuration, e.g., a class is added to the code basis or a method is modified. In this regard, the authors specify a transition system, in which each transition applies such a delta to the code basis. Further, they ensure a correct device configuration after a transition with a safety check. In this regard, every reconfiguration results in a valid target configuration. The authors do not rely on a *complete* configuration state space. Instead, they are able to derive new configurations and remove existing configurations dynamically at runtime. Similar to my approach, the authors contribute to *RC6* by avoiding a state explosion. To avoid such a state explosion, the authors dynamically derive a configuration state on-demand at runtime or remove existing states. A fundamental difference to my approaches is the focus on *delta oriented programming*. The authors encode the reconfiguration semantics in the transitions of a system, whereas I describe a device configuration via state predicates. This allows me to abstract from states by using unrestricted features, which further reduces the computational utilization at runtime.

Similarly to Damiani et al., Helvensteijn [Hel12] introduces a technique for the reconfiguration of DSPLs at runtime using Mealy Machines as a transition system. Based on delta modeling techniques the author proposes a delta oriented approach to specify transitions between two configurations. By labeling the transitions with a weight, it is possible to reduce the cost of a reconfiguration. In contrast to my approach, the author assumes a *complete* state space consisting of every possible valid configuration of the DSPL. This results in a *fully connected* graph that specifies the reconfiguration behavior of the DSPL, which is not in favor of *RC6*. One problem the paper does not address is the state explosion w.r.t. all possible combinations of deltas and product configurations.

Surveys, such as [BHS12], propose the usage of transition systems to handle the reconfiguration of a DSPL at runtime. However, to the best of my knowledge, there are no approaches that provide

- concepts for a context-aware state space derivation as a transition system as well as
- the possibility to reduce the size of a state space without restricting the adaptation capabilities at runtime of a DSPL.

Lee et al. [LK10] present an approach to derive a product configuration based on contextual information, which is similar to the concepts provided in this thesis. In contrast to my concept of a context-aware DSPL, Lee et al. aim to support the developers to select features suitable for a contextual situation by providing certain recommendation criteria. Thus, it is not intended for a continuous autonomous reconfiguration of features as required by *RC5*.

For the reconfiguration of a DSPL, I propose a reconfiguration semantics that relies on a transition system and refines a partial configuration until a complete valid configuration is derived. Similar approaches propose a process model for the state configuration of an SPL that refines a configuration in a step-wise manner. Such an approach is discussed in the next paragraph.

MULTI-STEP REFINEMENT OF A CONFIGURATION. White et al. proposes to use SPLs as a reconfigurable software architecture [WDSB09]. This definition is close to the basic principle of a DSPL with the difference that the SPL is reconfigured across multiple stages instead of a reconfiguration at runtime. In contrast to the previously discussed stages-configuration concepts of Czarnecki et al., each stage corresponds to a fully configured product configuration.

The authors propose a technique to derive product configurations via multiple steps by mapping the problem to a constraint satisfaction problem. This results in a path, where a product configuration is derived automatically via multiple steps. Similar to a DSPL reconfiguration, the authors propose a multi-step configuration based on a graph of configuration nodes and reconfiguration links. Start node and end node represent the start configuration and target configuration, respectively. The authors describe a technique how to choose a path that is optimized w.r.t. a certain criterion. For example, they investigated criteria such as minimal costs of a reconfiguration path or global requirements for configurations, e.g., each configuration utilizes costs below a certain threshold. Although the problem domain is similar, the approach of White et al. is not designed to handle dynamic reconfigurations of a device at runtime. Further, the specification process is assumed to be static, i.e., no configurations may be added once the (re-)configuration process has started.

At runtime, I dynamically refine unrestricted contexts and features of partial configuration states to derive a complete valid configuration. Therefore, I identify unrestricted contexts and features in a pre-processing step. The next paragraph discusses approaches that similarly are applicable to identify unrestricted contexts and features.

IDENTIFICATION OF UNRESTRICTED FEATURES. My techniques to identify unrestricted context and feature variables in a partial state are comparable to approaches that try to minimize a propositional formula. Every variable that is

removable corresponds to an unrestricted variable in my concept of a partial state. However, I do not aim at minimizing the amount of possible configurations as related approaches do [Qui52, Cou95, HCO74, SW79]. Instead, it is my goal to maximize unrestricted variables for a partial configuration and a partial state.

In the evaluation of my approach, I compare my BDD-based techniques with the Quine and McCluskey (QMC) algorithm [McC56, Qui52, Rud89] to minimize a logic formula. Instead of a BDD representation of the formula, QMC relies on a propositional formula in the conjunctive normal form, i.e., variables are concatenated disjunctively to a formula and every formula is concatenated conjunctively. QMC identifies unrestricted variables by comparing a set of formulae Φ, in which each formula $\varphi \in \Phi$ consists of variables that are concatenated in a disjunctive manner. Afterwards, the formulae $\varphi \in \Phi$ are compared in a pair-wise manner to identify removable, unrestricted variables. My proposed BDD-based techniques rely on a BDD representation of a formula. Therefore, I do not need to compare every conjunctive formula in a pair-wise manner. Instead, I identify unrestricted variables based on the structure of a BDD, which is much faster, as my evaluation in Section 6.6.3 indicates.

O. Coudert [Cou95] proposed an approach that also uses a BDD to minimize a propositional formula as I do. The main difference to my approach is that the author uses a fix-point algorithm to re-order the variables of a BDD, whereas I rely on either a genetic, a heuristic, or a random re-ordering of variables. The fix-point algorithm aims to reduce the amount of possible configuration interpretations of a formula, i.e., the author tries to minimize the disjunctive normal form representation of a formula. Instead, it is my goal to derive as much unrestricted variables as possible. For this goal, the fix-point algorithm of O. Coudert probably leads to better results since *every* possible permutation in the variable ordering is investigated. However, to achieve this completeness, the approach consumes much more resources and time to identify unrestricted variables.

The next section discusses related approaches that may be used to plan reconfigurations w.r.t. future contextual changes that also rely on transition systems as a model.

8.5 Prediction and Planning of Adaptations

The *planning as model checking* domain uses model checkers to derive a path that satisfies certain goals [CRT98, GT00, SG01]. Similarly to my approach, the problem is formalized using propositional formulae to specify a transition system. Each state is assigned a set of Boolean variable interpretations. Further, the transition system is capable to express non-deterministic system behavior, i.e., for a transition there exist multiple possible target states. Functional goal requirements are specified via temporal logics [PT01], e.g., which properties have to hold in a certain state. A model checker derives a path from a start state to a target state, on which none of the formulated state properties are violated. In terms of my formalization, such a path corresponds to a series of reconfigurations. However, none of these approaches provides the possibility to include non-functional properties,

such as costs, to tackle *RC4* and *RC6* by *predicting* the future cost development of reconfigurations.

In the following, I categorize related approaches into

- planning and verification at design time and
- planning and verification at runtime.

PLANNING AND VERIFICATION AT DESIGN TIME. The domain of adaptive systems is well established and provides a broad spectrum of approaches and solutions. Approaches that deal with model-based adaptations formally describe the possible adaptation behavior of a system. With such a description, a transition system is derived to coordinate or verify [CGKM12] the overall adaptation process. Specific approaches also include non-functional properties and probabilities of an adaptation [PS09, SHMK10, SPA12] in their system specification. Although such approaches rely on a model specification similar to a probabilistic weighted automaton introduced in this thesis, to the best of my knowledge, I am the first to use such a model to reduce reconfiguration costs based on a probabilistic look-ahead of changes in the context.

In addition to probabilistic weighted automata, stochastic models, such as continuous Markov chains, are also used to handle the runtime adaptivity of a device [QP99, PBBDM98]. These approaches rely on a system that reacts on incoming events. The transition behavior of the system depends

- on the system state at a specific point of time and the incoming event and
- on the stated optimization problem, e.g., minimize energy consumption.

Similar to my approach, a state automaton is used to specify the adaptation capabilities. A solution is derived by establishing and continuously updating a probability matrix. Linear programming algorithms are used to achieve certain optimization goals.

Although my approach is executed at design time, I discuss similar approaches that are applicable at runtime in the next paragraph.

PLANNING AND VERIFICATION AT RUNTIME. Recent approaches stress the necessity of a verification of runtime behavior [CGKM12, Daw05] to verify, to which extent, non-functional requirements, such as costs, efficiency or robustness, are satisfied by a system specification. Such a verification may be done at runtime, before a system is deployed on a device to verify if the system may end up in a deadlock. However, the system specification may change at runtime, e.g., the probability distribution of a contextual change has to be adapted with every contextual change. Therefore, such the verification may also be executed dynamically at runtime. Filieri et al. [FGT11] propose an approach for dynamically executing a verification at runtime. They execute verifications dynamically on a system specification that is being altered at runtime to execute an adaptation with costs that are below a certain threshold. In order to reduce the computational efforts for executing a verification, the authors assume certain aspects of the specification to be static, e.g., the probability of an error occurring during sending a message is

assumed to remain statically at 1%. Based on this assumption, they pre-process the original specification. They pre-compute the static aspects and introduce variables that are updated at runtime. Every time such a variable is changed, they just re-evaluate the verification results that are affected by such a change. With this concept, the authors tackle *RC4* and reduce the computational utilization of a verification process at runtime.

A similar approach focuses on contextual changes that were not considered at design time [EGMT09]. In this regard, they consider a system specification to be altered with every contextual change, e.g., new configuration states are added, old states are removed, or new transitions are added or removed. In such a case, each verification of a certain requirement may return different results after every change.

Although, I similarly assume that a system specification changes w.r.t. the contextual situation of a system, my approach has a different goal. Instead of permanently ensuring the correctness of a system at runtime and, thereby, preventing erroneous reconfigurations from occurring, I use verification techniques to precompute cost-effective configuration states w.r.t. predicted upcoming contextual changes. However, those approaches [EGMT09, FGT11] are not contradictory to my approach. Instead, they are combinable, since I use a similar model for a system specification and, therefore, those approaches may be used at runtime, to fine tune my prediction and ensure correctness.

Dubslaff et al. [DKB14] analyze the energy consumption of (D)SPLs based on a probabilistic weighted automaton and a model checker. Therefore, their automaton implements the configuration semantics of a feature model in a similar manner as I do. The authors track the choice of a reconfiguration with a probability distribution. For example, if several configurations are suitable for a reconfiguration, they investigate which configuration is chosen with a probability measure. In contrast to that, I measure the reason for a reconfiguration, i.e. contextual changes, with a probability distribution. Thus, I do not track, which specific configuration state is chosen for a certain contextual change. Based on their probabilistic weighted automaton, the authors analyze the frequency of a configuration state to be chosen for a reconfiguration. The probability of various configuration states to become active is used to determine when a reconfiguration has to be executed in order to (i) save energy costs and (ii) to finish the reconfiguration process in time. In this regard, the authors tackle *RC6* by providing a technique to reduce the energy consumed at runtime. However, the authors apply their approach for the reconfiguration of the network interfaces of a server. Therefore, they are able to use a model checker that utilizes a considerable amount of memory and processing capabilities.

The next chapter concludes this thesis by summarizing the contributions given in this thesis, discussing observations and open problems, as well as introducing possible future research topics to tackle identified limitations of this thesis

9

CONCLUSIONS

The objective of this thesis was to establish a model-based adaptation process for mobile devices. In this regard, I established concepts and techniques to satisfy the following goals

- providing an autonomous, model-based adaptation process (G1) and
- reducing the resource consumption of an adaptation at runtime (G2).

To conclude my thesis, I summarize the contributions provided in this thesis in order to tackle both goals G1 and G2. Afterwards, I critically discuss the results of the provided contributions and point out issues that remain open. Furthermore, I provide an overview of future research topics that tackle the identified limitations of this thesis.

9.1 Summary

To manage the inherent adaptivity of mobile devices at runtime, I combined the two existing concepts of a MAPE-K feedback loop with the model-based paradigm of Dynamic Software Product Lines (DSPLs) in Chapter 3. Since vendors, such as Google, HTC, Motorola, or Samsung, provide a product line of mobile devices, my thesis has shown that DSPL models and techniques as a well-known extension of Software Product Lines (SPLs) are an appropriate choice to manage the runtime adaptivity of mobile devices. Note that, although I focus on the domain of mobile devices, my concepts and techniques are generally applicable to other domains such as peer-to-peer systems or service-oriented architectures. To tackle the first goal G1 of this thesis, I

(1) established the concept of a context-aware DSPL to autonomously handle changes in the context of a device (see Chapter 4) and

(2) introduced DSPL specific, three-valued configuration and reconfiguration semantics (see Chapters 4 and 6, respectively).

To provide context-awareness in DSPLs (1) I enriched traditional (D)SPL feature models with contextual elements to denote dependency relations, such as exclude and require, amongst contexts and features. Therefore, contexts constitute atomic elements, such as *Home* or *Office*. These context elements are modeled as traditional features. However, in contrast to traditional features, contexts are not part of a software system. Instead, contexts are used to specify contextual

requirements, such as *"WLAN is required to be active at home"*. In this regard, a context-feature model offers the possibility to

- intuitively model contextual requirements for a DSPL and
- autonomously reconfigure a DSPL by denoting, which contexts are active and, therefore, which features have to be activated or deactivated to satisfy the contextual requirements.

For (2) I proposed a formal framework by establishing (re-)configuration semantics for DSPLs. The configuration semantics of a DSPL is defined on the basis of a three-valued propositional formula, in which features and contexts correspond to variables. A configuration of a DSPL is valid if the assignment of variables satisfies the propositional formula. A context denotes a partial assignment of variables, i.e., which features have to be active or inactive.

In addition to a propositional formula, I used transition systems to specify the reconfiguration semantics of a DSPL. Such transition systems are automatically derivable from a feature model by computing the valid configurations of a context-feature model. Every valid configuration of a feature model corresponds to a state in the transition system. A reconfiguration is denoted by a transition between two states. Such a transition system is applicable to

- specify the reconfiguration behavior of a DSPL and
- analyze and reason about formal system properties.

Thus, for a DSPL-based adaptation, a feature model specified as propositional formula as well as a reconfiguration transition system have to reside on the device as adaptation knowledge.

To further tackle the second goal G2 of this thesis, I

(3) reduced a feature model specification w.r.t. the individual characteristics of a device (see Chapter 5),

(4) introduced two approaches to reduce a configuration state space and, thereby, minimize the resource utilization of a DSPL, i.e.,

  (4.1) the concept of a state space reduction to an *incomplete* state space w.r.t. a contextual coverage criterion (see Chapter 6) as well as

  (4.2) an abstraction technique for deriving *partial* configuration states, to enhance the flexibility of a DSPL and to reduce the size of a state space (see Chapter 6), and, finally, I

(5) reduced the operational costs of an adaptation by predicting changes in the context (see Chapter 7).

To reduce the complexity of a feature model (3), I removed non-reconfigurable features w.r.t. a device-specific reduction criterion. In this regard, contexts and features are removed if they depend on features that are incompatible to a device. For instance, if a device has no integrated WLAN-chip, every context and feature that depends on a WLAN-based connection may never be activated at runtime

and, therefore, are removed from the context-feature model. Similarly, features that have to be permanently active at runtime are removed. In this regard, I removed variables from the context-feature model formula that have to be active or are incompatible to the device.

To further reduce the complexity of a DSPL-based adaptation, I reduced the configuration state space to an incomplete state space (4.1). The complete configuration state space of a DSPL is reduced based on the assumption that a user of a mobile device behaves according to repetitive contextual patterns. Therefore, I proposed a metric to cover certain individual contexts or combination of contexts, for which a configuration state is derivable. Configuration states that are not required at runtime w.r.t. the context-coverage metric are discarded. Hence, this concept imposes a trade-off between the complexity of a complete configuration state space and the possibility to compute a configuration on-demand at runtime.

A further reduction of the size of a state space is achieved with the abstraction concept of a partial state (4.2). In a partial state, context or features are identified, which are *unrestricted* w.r.t. the remaining configuration of contexts and features. This allows an unrestricted context or feature to be arbitrarily reconfigurable, i.e., no costly solver call is necessary if an unrestricted context or feature needs to be reconfigured. In this regard, partial states subsume multiple states, which are fully configured. I identified unrestricted contexts and features by transforming the context-feature model into a binary decision diagram (BDD). A BDD strongly relies on the ordering of context and feature variables. I re-ordered these variables based on three different techniques, i.e., a randomized ordering, a heuristical ordering, and a genetic ordering, to identify unrestricted contexts and features. In addition to further reduce the size of a state space due to the subsumption of configuration states, a partial state improves also the flexibility of a state space. With a partial state, the state space is able to handle contextual changes, which were not pre-planned at design time. This flexibility is based on the amount of unrestricted contexts and features in a pre-planned partial state. For example, every unrestricted context in a partial state may be arbitrarily entered or left without the need to execute a reconfiguration.

To reduce the operational costs of a DSPL at runtime (5) I introduced an approach to predict contextual changes based on a probabilistic transition system and model checking techniques. At runtime, the frequency of contextual changes are continuously tracked by the device itself. With every contextual change the respective probability distribution of a contextual change is updated accordingly. In addition to this probabilistic transition system, I used a weighted transition system to model the reconfigurability of a device as well as non-functional properties of a reconfiguration, such as the cost of a reconfiguration. Based on these two transition systems I reduced the reconfiguration cost of a DSPL on a long-term basis by choosing a configuration state as a target state for a reconfiguration that is the cheapest w.r.t. future upcoming contextual changes. To identify such a target state, I exploited model checking techniques to calculate an estimation of future upcoming reconfiguration costs for each possible target configuration state.

9.2 Observations and Open Problems

The contributions provided in this thesis are not limited to the development of concepts. I also implemented and evaluated my concepts and techniques w.r.t. trade-offs and potential gains at runtime. In the following I discuss the advantages and disadvantages of my concepts and techniques provided in this thesis.

Context-Aware DSPLs. The extension of the traditional DSPL concept to a context-aware DSPL has proven to be beneficial for all my techniques to improve the resource utilization of a DSPL-based adaptation. The seamless integration of contexts into a feature model eases the specification of contextual requirements and, at the same time, provides the means to autonomously trigger a reconfiguration w.r.t. changes in the contextual situation of a system. This simplicity of modelling contextual requirements comes with the cost of a reduced expressiveness in comparison with existing context-modelling approaches. For example, the concept of a context-feature model may be used to derive a configuration for a contextual situation. However, since this model does not cover non-functional properties, such as stability or efficiency, it is not possible to derive the *ideal* configuration for a contextual situation w.r.t. non-functional properties.

Reduction of a Feature Model. Existing approaches for the reduction of a feature model specification propose slicing techniques on feature model diagrams. However, since it is sufficient to use a formula representation of the feature model to derive a configuration at runtime, my technique is intended for a reduction of a formula representation instead of a diagram representation. It is also easier to develop optimization and reduction techniques, which are based on a logic formula instead of a diagram specification.

Based on the investigated simulation models, my experiments show that the derivation of a configuration based on a reduced feature model is up to 60% faster. Furthermore, the computational efforts for deriving a configuration are decreased by 66%. Nonetheless, the speed-up and reduction of computational efforts are costly to compute and, therefore, a feature model reduction cannot be executed at runtime due to the inherent restriction of resources on mobile devices. Instead, such a pre-processing step has to be executed prior to runtime on a PC or deployment server. Finally, I have to point out that the results of my experiments strongly depend on the feature model, which is to be reduced, i.e., the number of features and feature constraints, as well as the reduction criterion.

Incomplete State Space. As a first technique to minimize the impact of a DSPL state space, a rigorous pre-planning of a state space is executed prior to runtime. The state space is pre-planned w.r.t. a metric in order to cover configurations, which are required by contextual situations emerging at runtime, has shown to be an efficient improvement to alternative reconfiguration concepts. My stimulative experiments indicate that a state space with a size of 105.000 may be reduced by 99.998% to a set of 210 representative configuration states that cover all contextual situations consisting of up to 6 atomic contexts. At the same time,

the demand to execute a costly solver call at runtime to compute a missing configuration remains below 1%. However, the demand to compute a configuration on-demand at runtime significantly increases if the state space is further reduced. Thus, the concept of incomplete state spaces that cover certain contextual situations shows great potential to significantly reduce

- the memory consumption of DSPLs and
- the search efforts for suitable states

without noteworthy additional computational efforts at runtime.

PARTIAL STATES. As a second technique to further reduce the size of a state space, completely configured states are abstracted to partial states. This technique has shown to be a promising concept. The benefit of a state space with partial states is twofold. Firstly, the amount of states is further reducible since a partial state may subsume multiple configuration states. My experiments show that every third state is abstractable to a partial state by subsuming up to 4 configuration states. Secondly, the state space is able to cover contextual situations, which were not pre-planned at design time. The coverage of additional contextual situations is the result of unrestricted contexts and features identified in a partial state. Such unrestricted contexts and features are arbitrarily reconfigurable w.r.t. unplanned contextual situations that may spontaneously emerge at runtime. After 500 contextual changes, 20% of the resulting reconfigurations affected unrestricted features. However, I have to point out that the derivation of a partial state is computationally expensive. Although I was able to mitigate the processing time of identifying unrestricted features, the computational efforts remain considerably high. Another disadvantage of partial states is the increased complexity of the overall approach since with a partial state not only several configuration states may be suitable for a contextual situation. Instead, further partial states may be suitable to cover a contextual situation in addition to complete configuration states. Thus, the right partial or complete configuration state needs to be identified. Another problem may emerge from the fact that the identification of unrestricted features heavily relies on the constraints specified in the context-feature model. According to my experiments it is safe to assume that a more restrictive specification results in less unrestricted features.

PREDICTION OF CONTEXTUAL CHANGES Using a model checker with a probabilistic weighted transition system to predict future reconfiguration costs is directly compatible to my reconfiguration semantics based on a transition system. Model checking and transition systems are both well researched techniques, which constitutes a major benefit for my concept of reducing reconfiguration costs. Combining both techniques of these to reduce the operational costs of an adaptive software system on a long-term basis is an innovative contribution. The results of my experiments are promising. In a simulation of 100 contextual changes the overall utilized reconfiguration costs were reduced significantly iff the user moves according to previously monitored contextual change patterns. Nonetheless, in conclusion I have to point out that the proof-of-concept implementation of

my current prediction approach is not directly applicable for the usage on mobile device. The executed exploration of configuration states on a path of future contextual changes consumes to much computational resources and does not scale with the amount of configuration states and contextual states. A model checker, such as PRISM [GPS13], has shown to be oversized for the purpose of a cost prediction of future reconfigurations. Existing approaches already intend to reduce the model checking efforts in order to execute model checking queries dynamically at runtime [FGT11]. Similarly, a customized solution with the sole intention to compute a cost-prediction has to be implemented to execute my approach on mobile devices.

The development of a customized solution to predict costs of a reconfiguration instead of using a model checker is only one possible step to further investigate the research topics discussed in this thesis. The next section discusses unaddressed research questions and future work.

9.3 Future Work

The concepts and techniques presented in this thesis revealed some deficiencies that have to be tackled in future contributions. These shortcomings concern

- further concepts to reduce the resource utilization of a context-aware DSPL reconfiguration process,
- new concepts to apply a DSPL-based adaptation to heavily resource constrained devices such as sensor nodes,
- extending a DSPL-based adaptation to coordinate an adaptation of *multiple* distributed devices, and
- quality assurance of self-adaptive software systems.

Runtime Prediction in Real-Time. First of all, an alternative to use a model checker for the computation of a cost prediction is required. In this regard, Discrete Time Markov Chains [FGT11] may be used to compute a probabilistic cost expectation for future upcoming reconfigurations. Alternatively, dynamic programming techniques may be applied. Based on such a technique, a representative completeness of the estimated costs utilized by the predicted contextual changes is traded for a reduction in the computational efforts. Thus, it has to be investigated how an estimation of upcoming reconfiguration costs is to be implemented to be (i) non-intrusive to the remaining system and (ii) derives a result in real-time. Furthermore, the approach needs to be evaluated in a large scale scenario and based on realistic data. For example, a dataset containing the usage behavior of smartphones [KJD+10, LGPA+12] may be used to validate my approach.

Applicability for Heavily Resource Constrained Devices. My concepts are intended to be applicable on resource constrained devices, such as smartphones, tablets, or notebooks. Wireless sensor nodes have even stronger resource

constraints, i.e., less computational power and less energy. Such devices have between 4kB and 512kB RAM, in contrast to 2GB of RAM provided by a Nexus 4. Up to now, there is no constraint solver available that may be executable on such heavily resource constrained devices. Therefore, solutions are required (i) to further reduce the computational efforts and (ii) that remotely compute a suitable configuration for a sensor node if the context of a node changes.

ATTRIBUTED CONTEXT-DSPL. My concept of a context-feature model neither supports non-functional properties of features nor non-functional requirements of contexts. However, Kang et al. [KCH+90] already propose to enrich features with attributes. Such attributes are applicable to specify non-functional properties and non-functional requirements. In this regard, it has to be investigated, how an *optimal* feature configuration is to be derived for the non-functional requirements of a context. Further, if a context imposes multiple non-functional requirements, an approach for a *multi-criteria* optimization is required. Such a multi-criteria optimization is expensive to compute on-demand at runtime. Therefore, it has to be investigated how the complexity of a multi-criteria optimization may be pre-processed in order to execute such an optimization at runtime.

OFFLINE EVOLUTION. Up to now, little support exists for an offline-evolution of a DSPL. A successful evolution requires approaches that profile the reconfiguration behavior at runtime. This behavior has to be analyzed, e.g., by data-mining approaches, and based on learning algorithms, e.g., reinforcement-learning, a feedback may be derived, which may be used to continuously optimize the DSPL. In this regard, the selection of configuration states, pre-computed prior to runtime, may be improved w.r.t. the individual behavior of a user. States may be removed, which never become active at runtime, and new states may be added, which are computed on-demand at runtime.

DISTRIBUTED DSPL. The concepts provided in this thesis are applicable to adapt *one* device *locally*. However, mobile devices become more and more interconnected via the Internet. In this regard, mobile devices may not only depend on services that are dynamically reconfigurable locally, but depend on services that are only accessible via the Internet. The Internet itself changes dynamically, e.g., services such as Skype may become unavailable. Thus, if one considers a service as a feature, features are dynamically activated and deactivated, which may trigger a reconfiguration of the device. To tackle such scenarios, approaches are required that coordinate a reconfiguration of a *distributed* DSPL. Distributed constraint solvers [DBL11] show promising advances in this regard. Thus, it has to be investigated, how my concepts are extendable to a distributed DSPL by using distributed constraint solvers instead of standard SAT or BDD constraint solvers.

QUALITY ASSURANCE. Traditional concepts for testing of SPLs focus on testing features and dependency relations between individual features. Testing a DSPL further requires concepts to test a dynamic (re-configuration) of the entire system. Thus, the quality assurance of a DSPL requires the testing of all possible

reconfigurations and all contextual changes. However, the inherent complexity of self-adaptive software systems renders the derivation of a complete DSPL test suite, i.e., the testing of all possible system behavior, infeasible. In order to tackle this issue the reduction approaches introduced in Chapter 5 and in Chapter 6 may be used. This provides the possibility to reduce the system under test itself due to the reduction of configuration states to a representative set of state that cover certain contextual situations. In this regard not all possible reconfiguration behavior of a DSPL needs to be tested. Furthermore, the transition systems used in this thesis are applicable to define testing coverage criteria for reconfiguration transitions or contextual transitions.

9.4 Final Remarks

Context-aware DSPLs provide promising contributions to cope with the inherent dynamics of self-adaptive software systems used on mobile devices. However, DSPLs will only succeed on a long-term basis if their implementation is fast, non-intrusive, and the resource utilization is kept to a minimum. The approaches developed and evaluated in this thesis, therefore, provide a substantial contribution to establish a context-aware DSPL-based adaptation process of mobile devices.

BIBLIOGRAPHY

[ACG09] R. Ali, R. Chitchyan, and P. Giogini. Context for Goal-Level Product Line Derivation. *3rd International Workshop on Dynamic Software Product Lines*, pages 24–28, 2009. (Cited on page 9, 74, and 180.)

[ACLF11] M. Acher, P. Collet, P. Lahire, and R. B. France. Slicing Feature Models. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 424–427. IEEE, 2011. (Cited on page 9, 53, 82, and 182.)

[ADB+99] 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 *Proceedings of the 1st International Symposium on Handhelds and Ubiquitous Computing*, pages 304–307, 1999. (Cited on page 74.)

[AK03] C. Atkinson and T. Kühne. Model-Driven Development: a Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. (Cited on page 5.)

[AK09] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8:49–84, 2009. (Cited on page 32.)

[Ake78] S. B. Akers. Binary decision diagrams. *Transactions on Computers*, 100(6):509–516, 1978. (Cited on page 64.)

[ALMW09] J. Andersson, R. Lemos, S. Malek, and D. Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 27–47. Springer, 2009. (Cited on page 17 and 26.)

[Bat05] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Software Product Line Conference*, volume 2005, pages 7–20, 2005. (Cited on page 9, 33, 58, and 61.)

[BBF09] G. Blair, N. Bencomo, and R.B. France. Models@run.time. *Computer*, 42(10):22–27, 2009. (Cited on page 5.)

[BBH+10] 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. (Cited on page 9, 74, 180, and 181.)

[BE01] V. Bellotti and K. Edwards. Intelligibility and Accountability: Human Considerations in Context-aware Systems. *Human Computer Interaction*, 16(2):193–212, 2001. (Cited on page 3.)

[BG99] G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-valued Temporal Logics. In *Computer Aided Verification*, pages 274–287. Springer, 1999. (Cited on page 10, 102, 103, 123, and 124.)

[BG01] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th International Conference on Automated Software Engineering*, volume 872565 of *ASE 2001*, pages 273–280. IEEE, 2001. (Cited on page 5.)

[BHS12] N. Bencomo, S. Hallsteinsen, and E. Santana de Almeida. A View of the Dynamic Software Product Line Landscape. *Computer*, 45(10):36–41, 2012. (Cited on page xvi, 6, 8, 41, 46, 47, 73, 178, and 183.)

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. (Cited on page 150.)

[BLL+14] J. Bürdek, S. Lity, M. Lochau, M. Berens, U. Goltz, and A. Schürr. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2014. (Cited on page 179.)

[BMSG+09] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, 2009. (Cited on page 21, 22, and 26.)

[Bos01] J. Bosch. Software Product Lines: Organizational Alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 91–100. IEEE, 2001. (Cited on page 28 and 29.)

[BP10] D. Le Berre and A. Parrain. The SAT4J Library, Release 2.2. *JSAT*, 7(2-3):59–66, 2010. (Cited on page 88, 94, and 133.)

[Bro10] M. Broy, editor. *Cyber-Physical Systems*. Springer, 2010. (Cited on page 3 and 14.)

[BSBG08] 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 *Proceedings of the 12th International Software Product Line Conference*, pages 23–32, 2008. (Cited on page 6, 54, 102, 178, and 179.)

[BSRC10] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Advanced Information Systems Engineering*, 35:615–636, 2010. (Cited on page 27 and 34.)

[BTRCRC05] D. Benavides, P. Trinidad, A. Ruiz-Cortés, and A. Ruiz-Cort. Automated Reasoning on Feature Models. In *17th International Conference on Advanced Information Systems Engineering*. Springer, 2005. (Cited on page 179.)

[BW96] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs is NP-Complete. *Transactions on Computers*, 45(9):993–1002, 1996. (Cited on page 116.)

[CDH09] K. Chatterjee, L. Doyen, and T.A. Henzinger. Probabilistic Weighted Automata. In *Proceedings of the 20th International Conference on Concurrency Theory*, pages 1–15, 2009. (Cited on page 149, 159, and 161.)

[CDH10] K. Chatterjee, L. Doyen, and T. A. Henzinger. Quantitative Languages. *ACM Transactions on Computational Logic*, 11(4):1–38, 2010. (Cited on page 149, 154, and 155.)

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000. (Cited on page 30 and 33.)

[CE10] C. Cetina Englada. *Achieving Autonomic Computing through the use of Variability Models at run-time*. PhD thesis, Universitat Politècnica de València. Departamento de Sistemas Informáticos y Computación – Departament de Sistemes Informàtics i Computació, 2010. (Cited on page 3 and 4.)

[CGF08] C. Cetina, P. Giner, and J. Fons. A Model-Driven Approach for Developing Self-Adaptive Pervasive Systems. In *Models@run.time*, 2008. (Cited on page 179.)

[CGKM12] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-Adaptive Software needs Quantitative Verification at Runtime. *Communications of the ACM*, 55(9):69, 2012. (Cited on page 147, 177, and 186.)

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. (Cited on page 149 and 150.)

[CHE04] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. In R. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 162 – 164. Springer, 2004. (Cited on page 182.)

[CHE05] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature

Models. In *Software Process Improvement and Practice*, 2005. (Cited on page 52 and 182.)

[Cle99] P. Clements. Software Product Lines: A New Paradigm for the New Century. *The Journal of Defense Software Engineering*, pages 21–23, 1999. (Cited on page 27.)

[CLG+09] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer, 2009. (Cited on page 3, 4, 13, 14, 17, 22, 25, 26, and 73.)

[CN01] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001. (Cited on page 28.)

[Coo71] S. A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the 3rd Annual Symposium on Theory of Computing*, pages 151–158. ACM, 1971. (Cited on page 41.)

[Cou95] O. Coudert. Doing Two-Level Logic Minimization 100 Times Faster. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 112–121. Society for Industrial and Applied Mathematics, 1995. (Cited on page 185.)

[CRT98] A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains Via Model Checking. In *Proceedings of the Conference on Artificial Intelligence Planning Systems*, pages 36–43, 1998. (Cited on page 10, 147, 152, and 185.)

[CW07] K. Czarnecki and A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Software Product Line Conference*, pages 23–34. IEEE, 2007. (Cited on page 5, 9, and 61.)

[Daw05] C. Daws. Symbolic and Parametric Model Checking of Discrete-time Markov Chains. In *roceedings of the 1st International Conference on Theoretical Aspects of Computing*, pages 280–294. Springer, 2005. (Cited on page 186.)

[DBL11] K. R. Duffy, C. Bordenave, and D. J. Leith. Decentralized Constraint Satisfaction. *CoRR*, 1103(3240), 2011. (Cited on page 195.)

[Die78] D. L. Dietmeyer. *Logic Design of Digital Systems*. Allyn & Bacon, Inc., 2nd edition, 1978. (Cited on page 144.)

[DKB14] C. Dubslaff, S. Klüppelholz, and C. Baier. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *Proceedings of the 13th International Conference on Modularity*, pages 169–180. ACM, 2014. (Cited on page 187.)

[DPS12] F. Damiani, L. Padovani, and I. Schaefer. A Formal Foundation for Dynamic Delta-oriented Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. ACM, 2012. (Cited on page 7, 54, 97, 182, and 183.)

[DS11] F. Damiani and I. Schaefer. Dynamic Delta-oriented Programming. In *Proceedings of the 15th International Software Product Line Conference*, volume 2. ACM, 2011. (Cited on page 183.)

[EEM10] A. Elkhodary, N. Esfahani, and S. Malek. FUSION: a Framework for Engineering Self-tuning Self-adaptive Software Systems. In *Proceedings of the 8th International Symposium on Foundations of Software Engineering*, 2010. (Cited on page 178.)

[EFT94] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer, 1994. (Cited on page 58, 59, and 60.)

[EGMT09] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model Evolution by Run-time Parameter Adaptation. In *Proceedings of the 31st International Conference on Software Engineering*, pages 111–121. IEEE, 2009. (Cited on page 187.)

[Elk10] A. Elkhodary. A Learning-Based Approach for Engineering Feature-Oriented Self-Adaptive Software Systems. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, pages 345–348. ACM SIGSOFT, 2010. (Cited on page 108.)

[ES04] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004. (Cited on page 64.)

[FFF+13] J. Floch, C. Frá, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis, H. Rahnama, et al. Playing music - building context-aware and self-adaptive mobile applications. *Software: Practice and Experience*, 43(3):359–388, 2013. (Cited on page 8, 98, 100, 108, and 177.)

[FGT11] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time Efficient Probabilistic Model Checking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 341–350, 2011. (Cited on page 186, 187, and 194.)

[FSSA06] M. Fahrmair, B. Spanfelner, W. Sitou, and K. D. Althoff. Unwanted Behavior and its Impact on Adaptive Systems in Ubiquitous Com-

puting. In *Proceedings of the Conference on Learning, Knowledge, and Adaptation*, pages 36–41, 2006. (Cited on page 3.)

[FYBSV93] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic Variable Reordering for BDD Minimization. In *Proceedings of the Design Automation Conference*, pages 130–135, 1993. (Cited on page 116 and 117.)

[Gar14] Gartner. Number of smartphones sold to end users worldwide from 2007 to 2013 (in million units). `http://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/`, 2014. [Online; accessed 23-May-2014], ©statista. (Cited on page xvi, 3, and 4.)

[GKSS08] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability Solvers. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008. (Cited on page 64.)

[GPS13] N. Gethin, David Parker, and J. Sproston. Model Checking for Probabilistic Timed Automata. *Formal Methods in System Design*, 43(2):164–190, 2013. (Cited on page 167, 168, and 194.)

[GT00] F. Giunchiglia and P. Traverso. Planning as Model Checking. In *Recent Advances in AI Planning*, pages 1–20, 2000. (Cited on page 10, 147, 150, 152, and 185.)

[Har06] B. Hardian. Middleware Support for Transparency and User Control in Context-Aware Systems. In *Proceedings of the 3rd International Middleware Doctoral Symposium*. ACM, 2006. (Cited on page 26.)

[HCH09] A. Hubaux, A. Classen, and P. Heymans. Formal Modelling of Feature Configuration Workflows. In *Proceedings of the 13th Software Product Line Conference*, 2009. (Cited on page 179.)

[HCO74] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A Heuristic Approach for Logic Minimization. *IBM journal of Research and Development*, 18(5):443–458, 1974. (Cited on page 185.)

[Hel12] M. Helvensteijn. Dynamic Delta Modeling. In *Proceedings of the 6th International Workshop on Dynamic Software Product Lines*, 2012. (Cited on page 7, 8, 54, 57, 97, 98, 100, 103, 108, 182, and 183.)

[HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. W. Bakker, C. Huizing, W. P. Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer, 1992. (Cited on page 102.)

[HMS10] S. Hölldobler, N. Manthey, and A. Saptawijaya. Improving Resource-Unaware SAT Solvers. In C. Fermüller and A. Voronkov,

editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 519–534. Springer, 2010. (Cited on page 41.)

[Hor01] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM, 2001. (Cited on page 4.)

[HPS12] M. Hinchey, S. Park, and K. Schmid. Building Dynamic Software Product Lines. *Computer*, 45(10):22–26, 2012. (Cited on page 6, 8, 37, 41, and 97.)

[HR85] F. Hayes-Roth. Rule-Based Systems. *Communications of the ACM*, 28(9):921–932, 1985. (Cited on page 57.)

[HSSF06] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using Product Line Techniques to Build Adaptive Systems. In *Proceedings of the 10th International Software Product Line Conference*, pages 141–150, 2006. (Cited on page 37, 46, 57, and 178.)

[HST+08] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating Formal Properties of Feature Diagram Languages. *Institution of Engineering and Software*, (3):281–302, 2008. (Cited on page 32.)

[HT08] H. Hartmann and T. Trew. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *Proceedings of the 12th International Software Product Line Conference*, pages 12–21. IEEE, 2008. (Cited on page 74 and 180.)

[IBM06] IBM. An Architectural Blueprint for Autonomic Computing. 2006. (Cited on page xvi, 5, 23, 24, and 25.)

[ICP+99] A. Iwata, C. C. Chiang, G. Pei, M. Gerla, and T. W. Chen. Scalable Routing Strategies for Ad Hoc Wireless Networks. *Journal on Selected Areas in Communications*, 17(8):1369–1379, 1999. (Cited on page 6.)

[JL10] B. S. Jensen and J. E. Larsen. Estimating Human Predictability from Mobile Sensor Data. In *Machine Learning for Signal Processing*, pages 196–201, 2010. (Cited on page 10, 19, 55, 107, and 147.)

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature Oriented Domain Analysis (FODA). Technical report, Carnegie-Mellon University, 1990. (Cited on page 6, 30, 33, 37, 38, 73, 181, and 195.)

[Kel76] R. M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, 1976. (Cited on page 102.)

[KJD+10] N. Kiukkonen, Blom J., O. Dousse, D. Gatica-Perez, and Laurila J. Towards rich Mobile Phone Datasets: Lausanne Data Collection

Campaign. In *Proceedings of the 7th International Conference on Pervasive Services*. ACM, 7 2010. (Cited on page 174 and 194.)

[Knu74] D. E. Knuth. Computer Programming As an Art. *Communications of the ACM*, 17(12):667–673, 1974. (Cited on page v.)

[KV00] Y.-B. Ko and N. H. Vaidya. Location-Aided Routing (LAR) in Mobile Ad-Hoc Networks. *Wireless Networks*, 6(4):307 – 321, 2000. (Cited on page 7 and 40.)

[Lee06] J. Lee. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *10th International Software Product Line Conference*, pages 131–140, 2006. (Cited on page 46, 57, and 178.)

[LGPA+12] J. K. Laurila, D. Gatica-Perez, I. Aad, Blom J., O. Bornet, Trinh-Minh-Tri Do, O. Dousse, J. Eberle, and M. Miettinen. The Mobile Data Challenge: Big Data for Mobile Computing Research. In *Pervasive Computing*, 2012. (Cited on page 174 and 194.)

[LK06] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Proceedings of the 10th International Software Product Line Conference*, pages 130–140, 2006. (Cited on page 37.)

[LK10] K. Lee and K. Kang. Usage Context as Key Driver for Feature Selection. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2010. (Cited on page 184.)

[LKR10] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2010. (Cited on page 81.)

[LLYM06] S. Liaskos, A. Lapouchnian, E. Yu, and J. Mylopoulos. On Goal-Based Variability Acquisition and Analysis. In *Proceedings of the 14th International Requirements Engineering Conference*, pages 79–88. IEEE, 2006. (Cited on page 17.)

[LM09] A. Lapouchnian and J. Mylopoulos. Modeling Domain Variability in Requirements Engineering with Contexts. In *Proceedings of the 28th International Conference on Conceptual Modeling*, 2009. (Cited on page 180.)

[Loc12] M. Lochau. *Model-Based Conformance Testing of Software Product Lines*. PhD Thesis, TU Braunschweig, 2012. (Cited on page 30.)

[LY04] L. Liu and E. Yu. Designing Information Systems in Social Context: A Goal and Scenario Modelling Approach. *Information Systems*, 29(2):187–203, 2004. (Cited on page 9 and 17.)

[MBJ09] B. Morin, O. Barais, and J. M. Jézéquel. Models@run.time to Support Dynamic Adaptation. *Computer*, 42(10), 2009. (Cited on page 41.)

[McC56] E. J. McCluskey. Minimization of Boolean Functions. *Bell System Technical Journal*, 35(6):1417–1444, 1956. (Cited on page 185.)

[MCP13] R. Muschevici, D. Clarke, and J. Proença. Executable modelling of dynamic software product lines in the ABS language. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 17–24, 2013. (Cited on page 40.)

[MRD08] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th International Conference on World Wide Web*, pages 815–824. ACM, 2008. (Cited on page 21.)

[MSRJ12] J. McInerney, S. Stein, A. Rogers, and N. R. Jennings. Exploring Periods of Low Predictability in Daily Life Mobility. In *Mobile Data Challenge by Nokia*. Nokia, 2012. (Cited on page 10, 27, 55, 107, and 147.)

[MWC09] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240, 2009. (Cited on page 41.)

[Nie14] Nielsen. Share of smartphone subscribers by operating system and device manufacturer in the U.S. in 2nd quarter of 2012. `http://www.statista.com/statistics/271994/smartphone-users-by-operating-system-and-device-in-the-us/`, 2014. [Online; accessed 23-May-2014], ©statista. (Cited on page xvi, 5, and 6.)

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*, volume 1211. Tioga Publishing Company, 1980. (Cited on page 23.)

[NSL+12] A. Noulas, S. Scellato, R. Lambiotte, M. Pontil, and C. Mascolo. A Tale of Many Cities: Universal Patterns in Human Urban Mobility. *PloS one*, 7(5), 2012. (Cited on page 147 and 153.)

[OGC12] Ó. Ortiz, A. B. García, and R. Capilla. Runtime Variability for Dynamic Reconfiguration in Wireless Sensor Network Product Lines. In *Proceedings of the 16th International Software Product Line Conference*, volume 2, pages 143–150, 2012. (Cited on page 8 and 97.)

[Par76] D. L. Parnas. On the Design and Development of Program Families. *Transactions on Software Engineering*, SE-2(1):1–9, 1976. (Cited on page 27.)

[PBBDM98] G. A. Paleologo, L. Benini, A. Bogliolo, and G. De Micheli. Policy optimization for Dynamic Power Management. In *Proceedings of the International Design and Automation Conference*, 1998. (Cited on page 186.)

[PBv05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. (Cited on page xvi, 5, 27, 28, 29, 47, and 48.)

[PG96] R. Puri and J. Gu. A BDD SAT Solver for Satisfiability Testing: An Industrial Case Study. *Annuals of Mathematics and Artificial Intelligence*, 17(2):315–337, 1996. (Cited on page 64.)

[PNS+00] D. Petrelli, E. Not, C. Strapparava, O. Stock, and M. Zancanaro. Modeling Context is Like Taking Pictures. In *Conference on Human Factors in Computers, Workshop "The What, Who, Where, When, Why and How of Context-Awareness"*, 2000. (Cited on page 14.)

[PRC14] Universitat Politécnica de Valéncia ProS Research Center. MOSKitt4SPL. `http://www.pros.upv.es/m4spl/`, 2014. [Online; accessed 08-May-2014]. (Cited on page 179.)

[PS09] C. Peper and D. Schneider. On Runtime Service Quality Models in Adaptive Ad-Hoc Systems. In *Proceedings of the International Workshop on Software Integration and Evolution @Runtime*, pages 11–18, 2009. (Cited on page 147, 177, 178, and 186.)

[PT01] M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-Deterministic Domains. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 479–484, 2001. (Cited on page 10 and 185.)

[QP99] Q. Qiu and M. Pedram. Dynamic Power Management Based on Continuous-Time Markov Decision Processes. In *DAC*, 1999. (Cited on page 186.)

[Qui52] W. V Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952. (Cited on page 185.)

[RBD09] R. Rouvoy, P. Barone, and Y. Ding. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. *Software Engineering for Self-Adaptive Systems*, 5525:164–182, 2009. (Cited on page 177.)

[RDN06] M. A. Razzaque, S. Dobson, and P. Nixon. Categorization and Modeling of Quality in Context Information. In *Proceedings of the International Joint Conference on Artificial Intelligence Workshop on AI and Autonomic Communications*, 2006. (Cited on page 14 and 181.)

[RLH06] Y. Rekhter, T. Li, and S. Hares. RFC 4271: A Border Gateway Protocol 4 (BGP-4). Technical report, IETF, 2006. (Cited on page 7 and 40.)

[RN04] S. Russell and P. Norvig. *Künstliche Intelligenz*, volume 2. Pearson Studium, 2004. (Cited on page xvii, 120, and 121.)

[Roz11] K. Y. Rozier. Survey: Linear Temporal Logic Symbolic Model Checking. *Computer Science Revision*, 5(2):163–203, 2011. (Cited on page 150.)

[RSA11] M. Rosenmüller, N. Siegmund, and S. Apel. Tailoring Dynamic Software Product Lines. *Proceedings of the 10th International Conference on Generative Programming and Component Engineering*, 2011. (Cited on page 9, 53, 82, and 182.)

[RSAS11] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible Feature Binding in Software Product Lines. *Automated Software Engineering*, 18(2):163–197, 2011. (Cited on page 179.)

[Rud89] R. Rudell. *Logic synthesis for VLSI design*. PhD thesis, University of California, Berkeley, 1989. (Cited on page 185.)

[SG01] S. K. Shukla and R. K. Gupta. A Model Checking Approach to Evaluating System Level Dynamic Power Management Policies for Embedded Systems. In *Proceedings of the 6th International High-Level Design Validation and Test Workshop*, pages 53–57, 2001. (Cited on page 185.)

[SGS13] D. Stingl, C. Gross, and K. Saller. *Decentralized Monitoring in Peer-to-Peer Systems*, volume 7847 of *Lecture Notes in Computer Science, Computer Communication Networks and Telecommunications*, pages 81–111. Springer, 2013. Published in "Benchmarking Peer-to-Peer Systems - Understanding the Quality of Service in Large-Scale Distributed Systems". (Cited on page 23.)

[Shi07] B. Shishkov. Model-driven Design of Context-Aware Applications. In *9th International Conference on Enterprise Information Systems*, 2007. (Cited on page 177.)

[SHMK10] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Exploiting Non-Functional Preferences in Architectural Adaptation for Self-Managed Systems. In *Symposium on Applied Computing*, page 431. ACM, 2010. (Cited on page 147, 177, and 186.)

[SHT06] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th International Conference on Requirements Engineering*, pages 139 –148. IEEE, 2006. (Cited on page 34.)

[Sim14] A. Simaitis. *Automatic Verification of Competitive Stochastic Systems*. PhD thesis, Department of Computer Science, University of Oxford, 2014. (Cited on page 150.)

[SLP04] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. In *Proceedings of the International Workshop on Advanced Context Modelling, Reasoning and Management*, 2004. (Cited on page 74.)

[SLR13] K. Saller, M. Lochau, and I. Reimund. Context-Aware DSPLs: Model-Based Runtime Adaptation for Resource-Constrained Systems. In *Proceedings of the 17th International Software Product Line Conference co-located Workshops*, pages 106–113, 2013. (Cited on page 39 and 99.)

[SMD+12] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes. Using Contraint Programming to Manage Configurations in Self-Adaptive Systems. *Comuter*, 45(10):56–63, 2012. (Cited on page 181.)

[SOS+12] K. Saller, S. Oster, A. Schürr, J. Schroeter, and M. Lochau. Reducing Feature Models to Improve Runtime Adaptivity on Resource Limited Devices. In *Proceedings of the 6th International Workshop on Dynamic Software Product Lines*, pages 135–142, 2012. (Cited on page 39 and 83.)

[SPA12] H. W. Schmidt, I. Peake, and H. A. Aysan. Towards Probabilistic Mode Automata for Adaptable, Resource-Aware Component-Based Systems Design. In *Proceedings of the International Improoving Systems and Software Engineering Conference*, 2012. (Cited on page 147, 177, and 186.)

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP - a New Search Algorithm for Satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227. IEEE/ACM, 1996. (Cited on page 64.)

[SSH06] S. Schmid, M. Sifalakis, and D. Hutchison. Towards autonomic networks. In D. Gaíti, G. Pujolle, E. Al-Shaer, K. Calvert, S. Dobson, G. Leduc, and O. Martikainen, editors, *Autonomic Networking*, volume 4195 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2006. (Cited on page 20 and 26.)

[Sto02] M. Stoelinga. An Introduction to Probabilistic Automata. *Bulletin of the European Association for Theoretical Computer Science*, 78:176–198, 2002. (Cited on page 149, 154, and 158.)

[SvGB05] M. Svahnberg, J. van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques: Research Articles. *Software Practice & Experience*, 35(8):705–754, 2005. (Cited on page 38.)

[SW79] A. Svoboda and Donnamaie E White. *Advanced Logical Circuit Design Techniques*. Garland STPM Press, 1979. (Cited on page 185.)

[TMKL07] Alessandra Toninelli, Rebecca Montanari, Lalana Kagal, and Ora Lassila. Proteus: A Semantic Context-Aware Adaptive Policy Model. In *Proceedings of the 8th International Workshop on Policies for Distributed Systems and Networks*, pages 129–140. IEEE, 2007. (Cited on page 181.)

[TYH+13] S. Talwar, S. Yeh, N. Himayat, K. Johnsson, G. Wu, and R. Q. Hu. *Capacity and Coverage Enhancement in Heterogeneous Networks*, pages 51–65. John Wiley & Sons, 2013. (Cited on page 13.)

[Ver09] H. Verkasalo. Contextual Patterns in Mobile Service Usage. *Personal and Ubiquitous Computing*, 13:331–342, 2009. (Cited on page 10 and 107.)

[VL01] A. Van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the 5th International Symposium on Requirements Engineering*, pages 249–262. IEEE, 2001. (Cited on page 180.)

[Voe11] M. Voelter. MD*/DSL Best Practices. 2011. (Cited on page 5.)

[vSR07] F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007. (Cited on page 28 and 29.)

[WDSB09] J. White, B. Dougherty, D. C. Schmidt, and D. Benavides. Automated Reasoning for Multi-Step Feature Model Configuration Problems. In *Proceedings of the 13th International Software Product Line Conference*, pages 11–20, 2009. (Cited on page 7, 57, 97, 178, 179, 182, and 184.)

[Wei81] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981. (Cited on page 81 and 82.)

[WL99] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: a Family-Based Software Development Process*. Addison-Wesley Longman Publishing, 1999. (Cited on page 28.)

[Yan12] Yan, T. and Chu, D. and Ganesan, D. and Kansal, A. and Liu, J. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 113–126. ACM, 2012. (Cited on page 15 and 55.)

[ZBC96] N. Zhuang, M. S. T. Benten, and P. Y. K. Cheung. Improved Variable Ordering of BDDS with Novel Genetic Algorithm. In *Proceedings of the International Symposium on Circuits and Systems*, volume 3, pages 414–417. IEEE, 1996. (Cited on page xvii, 116, 117, and 122.)

CURRICULUM VITAE

Personal Details

Date of Birth	30th of May, 1981
Place of Birth	Darmstadt, Germany
Nationality	German

Work Experience

2010 - 2014	Research Associate and Doctoral Candidate at the Real Time Systems Lab, TU Darmstadt
2008 - 2010	Quality Engineer at Matrix42 (Part Time)
2007 - 2009	Freelancer with Focus on Software Development, Database Engineering, Quality Assurance
2006	Internship Software Developer / System Engineer at Daimler Chrysler

Education

2010	Degree: Master of Science, Computer Science. Topic of Thesis: “Automatic Test Case Generation using Model Checking Techniques".
2007 - 2010	Student at Computer Science Department, TU Darmstadt
2007	Degree: Bachelor of Science, Software and Internet Technologies. Topic of Thesis: “Three-Dimensional Class Diagrams".
2002 - 2007	Student at the Computer Science Department, University of Mannheim
2001	Degree: A-Levels
1999 - 2001	Berufliches Gymnasium Groß-Gerau

Activities

2013	Team Lead Development *MAKI Demonstrator*
2012 - 2013	Lab Tutor *Software Systems*
2011 - 2012	Active Participant Research Proposal *MAKI – Multi-Mechanism Adaptation for the Future Internet*
2010 - 2013	Teaching Assistant *Software Engineering – Quality Assurance and Maintenance*