Model-based Quality Assurance of Cyber-Physical Systems with Variability in Space, over Time and at Runtime

Dem Fachbereich Informatik

von

Dr. rer. nat. Malte Lochau

vorgelegte Habilitationsschrift

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Model-based Quality Assurance of Cyber-Physical Systems with Variability in Space, over Time and at Runtime

Vorgelegte Habilitationsschrift

Vorgelegt von Dr. rer. nat. Malte Lochau
Tag der Einreichung der Habilitationsschrift: 12. Juni 2017

Technische Universität Darmstadt
Fachbereich Informatik [1]
Fachbereich Elektrotechnik & Informationstechnik [2]

[1] Das Habilitationsgesuch erfolgt am Fachbereich Informatik

[2] Die Habilitationsschrift entstand während der Beschäftigung im Fachbereich Elektrotechnik & Informationstechnik. Im Vorfeld der Einreichung fand zwischen den Fachbereichen eine Abstimmung statt.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass die Habilitationsschrift und die im Verzeichnis meiner wissenschaftlichen Veröffentlichungen angegebenen Werke – abgesehen von den in ihr ausdrücklich genannten Hilfen – von mir selbständig verfasst worden sind.
Hiermit erkläre ich weiterhin, dass ich zu keinem früheren Zeitpunkt ein Habilitationsverfahren eingeleitet habe.

Darmstadt, im Juni 2017

Dr. rer. nat. Malte Lochau

Danksagung

Die Inhalte der vorliegenden Arbeit sind im Zeitraum zwischen 2012 und 2017 während meiner Tätigkeit als Postdoc am Fachgebiet Echtzeitsysteme an der TU Darmstadt entstanden. Der überwiegende Anteil der beschriebenen Forschungsarbeiten ist im Rahmen meiner Mitarbeit an dem Forschungsprojekt IMoTEP (Teilprojekt des DFG Schwerpunktprogrammes 1593) sowie dem Sonderforschungsbereich 1053 (MAKI) entstanden. Mein besonderer Dank gilt Prof. Andy Schürr für die hervorrangenden Arbeitsbedingungen am Fachgebiet Echtzeitsysteme sowie für die sehr gute inhaltliche und moralische Unterstützung bei der Erstellung dieser Arbeit. Desweiteren möchte ich stellvertretend für die vielen Forschungskollegen, die wesentlich zu den Inhalten dieser Arbeit beigetragen haben, vor allem die Doktoranden Johannes Bürdek, Géza Kulcsár, Lars Luthmann und Markus Weckesser aus meiner Gruppe erwähnen.

Meinen Eltern möchte ich für die vielen Jahre der Begleitung meines Weges danken. Schliesslich möchte ich meiner Freundin Katrin danken. Ohne ihre unermüdliche Unterstützung und Ermunterungen wäre die Fertigstellung dieser Arbeit nicht möglich gewesen.

Darmstadt, im Juni 2017

Abstract

Cyber-physical systems (CPS) are frequently characterized by three essential properties: CPS perform complex computations, CPS conduct control tasks involving continuous data- and signal-processing, and CPS are (parts of) distributed, and even mobile, communication systems. In addition, modern software systems like CPS have to cope with ever-growing extents of variability, namely variability in space by means of predefined configuration options (e.g., software product lines), variability at runtime by means of preplanned reconfigurations (e.g., runtime-adaptive systems), and variability over time by means of initially unforeseen updates to new versions (e.g., software evolution). Finally, depending on the particular application domain, CPS often constitute safety- and mission-critical parts of socio-technical systems. Thus, novel quality-assurance methodologies are required to systematically cope with the interplay between the different CPS characteristics on the one hand, and the different dimensions of variability on the other hand. This thesis gives an overview on recent research and open challenges in model-based specification and quality-assurance of CPS in the presence of variability. The main focus of this thesis is laid on computation and communication aspects of CPS, utilizing evolving dynamic software product lines as engineering methodology and model-based testing as quality-assurance technique. The research is illustrated and evaluated by means of case studies from different application domains.

Contents

1 Introduction **3**
- 1.1 Context and Motivation . . . 3
- 1.2 Objectives . . . 4
- 1.3 Outline . . . 5

2 Background **6**
- 2.1 Illustrative Examples . . . 6
- 2.2 Basic Notions and Classification of Research Challenges . . . 10

3 Model-based Quality Assurance of Variability in Space **16**
- 3.1 Test-Suite Generation for Configurable Software . . . 16
- 3.2 Test-Suite Optimization and Test Prioritization for Configurable Software 19
- 3.3 Compositional Testing Theory for Variable Software . . . 21

4 Model-based Quality Assurance of Variability at Runtime **25**
- 4.1 Complex Binding-Time Constraints in Reconfigurable Software . . . 25
- 4.2 Multi-Instantiation Constraints in Reconfigurable Software . . . 28

5 Model-based Quality Assurance of Variability over Time **33**
- 5.1 Reasoning about Problem-Space Evolution of Configurable Software . . 33
- 5.2 Reasoning about Solution-Space Evolution of Configurable Software . . 35

6 Conclusion **39**

A Reprints of Selected Publications **44**

Chapter 1

Introduction

This chapter provides an overview on the research context and objectives summarized in this thesis and briefly describes the outline of the following chapters.

1.1 Context and Motivation

The next generation of embedded software systems, frequently summarized under the term *cyber-physical systems* (CPS), are often described by three essential characteristics: CPS perform complex *computations*, CPS conduct *control* tasks in terms of feedback loops involving physical processes, and CPS devices participate in distributed, and even mobile, *communication* networks [36].

In addition, nowadays software systems have to cope with ever-growing extents of inherent *variability* which, again, may come in three different flavors: variability *in space*, variability *at runtime*, and variability *over time*.

Variability in space offers predefined configuration options to users or environmental contexts of the system, in order to tailor the software to specific stakeholders' needs, resource constraints, and platform restrictions. In this regard, *software product line engineering* is one of the most prominent development paradigms for coping with the inherent complexity of configurable software [15]. In a product line, configuration options are described in terms of *features* [1]. Every feature corresponds to a distinct product characteristic visible to customers or environmental contexts within the problem space. In addition, every feature is part of an explicit *mapping* of (combinations of) features onto composeable development artifacts (e.g., design artifacts, code snippets, test cases) within the solution space of the product line. In this way, a product line allows for automated derivation of *software variants* corresponding to a given *product configuration* (i.e., a feature selection), by applying directives for software variation to a common core software, according to a set of configuration-specific feature artifacts.

Variability may not only occur once during the initial configuration of a configurable software, but may reoccur in consecutively, and even repeatedly, executed configuration stages throughout the entire life-cycle of a product, especially *at runtime*. For instance, in the area of *(self-)adaptive systems*, software is capable of continuously monitoring and adapting itself to ever-changing environmental contexts and user behaviors [19]. In this regard, *dynamic software product lines* propose extensions to the conceptual framework of feature-oriented software product line engineering, aiming at also coping with runtime variability [13]. In a dynamic software product line, the binding times of configuration options may be deferred to later stages in the life-cycle of the product, where (self-)adaptations are conducted in terms of repetitive *reconfigurations*.

To summarize, (dynamic) software product lines enable *preplanned* software variability by means of a-priori *anticipated* (re-)configuration options. However, software systems in general, and CPS in particular, tend to become more and more long-living thus being in operation for years or even decades. As a consequence, anticipating all future customers' needs, contextual requirements, new legal restriction etc. in advance already during the initial development of a CPS is impossible. Hence, the third kind of variability in modern CPS, variability *over time*, is continuously caused by initially unforeseen (and often even ad-hoc) updates to new *versions*, usually summarized under the terms *software aging* and *software evolution* [30]. Here, one of the major challenges arises in comprehending and properly documenting what *changes* have been actually applied to an evolving software system (or, an entire software product line) and what are the possible (intended as well as erroneous) *impacts* of those changes.

Depending on the particular application domain, CPS often constitute safety- and mission-critical parts of socio-technical systems. Software testing in general and *model-based testing* in particular are one of the most established quality-assurance techniques today, constituting a practicable and semi-automated, yet theoretically founded research discipline [37]. However, novel testing methodologies are required for efficient, yet effective *quality-assurance* (QA) of CPS to systematically cope with subtle interplays between the different CPS characteristics on the one hand, and the additional complexity introduced by the different dimensions of variability on the other hand.

1.2 Objectives

This thesis surveys selected contributions of the author as well as open challenges in the specification and quality-assurance of complex software systems such as CPS in the presence of variability. The main focus of this thesis is laid on the computation and communication aspects of CPS, utilizing *evolving dynamic software product lines* and *model-driven software development* as software-development methodology and *model-based testing* as quality-assurance technique. Concerning variability in space, an automated test-generation approach based on symbolic model-checking is presented for efficiently covering highly-configurable source code. A subsequent test-suite optimization and test prioritization approach aims at further improving efficiency of configurable software systems testing. In order to cope with variability in software systems consisting of multiple communicating, and even potentially distributed or mobile components, a theoretical framework is presented for reasoning about compositionality and decomposition properties in the context of input/output conformance testing.

Extensions of recent concepts of (dynamic) software product line engineering for handling variability at runtime include novel modeling and reasoning techniques for dealing with constraints on binding times of configuration options as well as reconfiguration behaviors. In addition, an extension to existing feature-modeling formalisms is presented for handling multiple feature instances, as being crucial in (self-)adaptive component-based or mobile communication systems.

Finally, to also handle unforeseen evolution, the presented techniques comprise methodologies for reasoning about software-artifact changes and their potential impact on crucial system properties, including model differencing on problem-space artifacts as well as formal regression analysis on solution-space artifacts of configurable software. The research challenges are illustrated using a collection of case studies from different recent CPS application domains: medical-device software in health care, control software in automation engineering, as well as mobile communication networks.

1.3 Outline

The remainder of this thesis is structured as follows.

Chapter 2. This chapter provides a short overview on general notions and concepts referred to throughout this thesis, namely foundations of CPS, quality assurance using model-based testing, software variability, and software evolution. In addition, several case studies from different application domains of CPS are used to illustrate and motivate open research challenges addressed in this thesis. To this end, a classification of research challenges is proposed which is used to structure the main part of this thesis and to classify the different contributions.

Chapter 3. This chapter presents model-based testing techniques coping with variability in space. The selected contributions comprise an automated test-suite generation methodology for efficient test coverage of configurable software, a conceptual framework for lifting the notions of test-suite optimization and test prioritization to configurable software, and a theoretical framework for reasoning about compositional testing in the presence of variability in space.

Chapter 4. This chapter presents modeling and quality-assurance techniques coping with variability at runtime. The main focus is laid on extending model-based specification techniques of product lines to also deal with dynamic software product lines. The selected contributions comprise approaches for specifying and analyzing complex binding-time constraints and reconfiguration behaviors in (self-)adaptive systems, as well as multi-instantiation configuration based on cardinality-based feature models.

Chapter 5. This chapter presents modeling and quality-assurance techniques coping with variability over time in terms of unforeseen software evolution. The main focus is laid on techniques for reasoning about the potential impact of syntactic as well as semantic changes applied to artifacts within the problem space as well as the solution space of evolving configurable software.

Chapter 6. This chapter concludes the thesis by giving a short summary of the results achieved so far and a brief outlook on open challenges and possible future research directions.

Appendix A. The appendix of this thesis contains reprints of full versions of the selected papers summarized in this thesis.

Chapter 2

Background

This chapter gives an overview on basic notions and open challenges in the fields of research addressed in this thesis. To this end, this chapter presents case studies to motivate open research challenges and to illustrate solution approaches and techniques presented in the remainder of this thesis. The selected case studies constitute representatives from different application domains of cyber-physical systems, namely *medical devices*, *automation engineering* and *mobile communications*. In particular, we consider challenges apparent in quality assurance of software systems being subject to different kinds of inherent *variability*. As quality-assurance methodology, this thesis mainly focuses on *model-based testing*.

2.1 Illustrative Examples

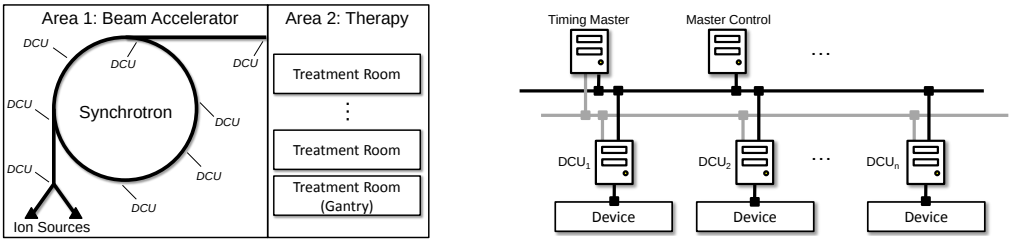
This section presents selected examples of real-world software systems from different modern application domains to illustrate the characteristics of cyber-physical systems and to motivate the research challenges considered in this thesis.

2.1.1 Medical-Device Control Software

The first sample software system comprises several components of the *control software*, being part of a complex medical device at the Heidelberg Ion-Beam Therapy Centre (HIT) [24]. The HIT provides a novel kind of radiotherapy with very precise irradiation beam positioning. With this technique, a remarkable reduction of damages to unaffected human tissues located nearby the tumor is achievable, as compared to conventional radiotherapy, especially in case of deeply positioned tumors. As illustrated in Figure 2.1a, the overall system consists of two parts.

- The beam-accelerator area consists of the ion sources and the synchrotron, being responsible for creating and regulating the ion beam during treatment procedures, as performed during a therapy program.
- The therapy area consists of a variable number of different types of treatment rooms including treatment rooms for conventional radiotherapy, as well as a gantry that allows for a very precise 360 degree positioning of ion beams.

For creating and regulating the ion beam, different physical devices are located around the synchrotron, such as magnets and E/E devices. As shown in Figure 2.1b, the software components being responsible for controlling those devices are *distributed* over



(a) System Architecture

(b) Software Architecture

Figure 2.1: HIT System and Software Architecture

different types of Device Control Units (DCUs) (e.g., one DCU is dedicated to control the angle position and the intensity of the magnets adjusting the beam cycle). Each DCU is an instance (or, variant) of the same core DCU and being *preconfigured* to become a concrete DCU type prior to its installation. The different DCU instances are integrated via a real-time bus (RTB) based on a hierarchical master-slave architecture. The master control initiates the beam creation, sets up the beam control with type-specific DCU parameters, continuously supervises the DCU status, and controls the operation mode of each DCU at runtime. A timing master further schedules and synchronizes time-critical tasks. To this end, the positioning of a DCU within the overall hierarchy determines the priority and, therefore, the timing delay of its tasks within the control cycle.

When activated, each DCU receives type-specific parameter values from the master-control device, which may change throughout the DCU control cycle. In the next step of a radiation procedure, beam creation is initialized and beam-control parameters are set up by the master control for each DCU component. Those dynamically adaptable runtime parameters adjust the control tasks to be performed by the different devices, depending on the therapy program. At run time, the behavior of each DCU further depends on its current *operation mode* as follows.

- In therapy mode, each DCU strictly follows a predefined therapy procedure by continuously setting and updating control parameters of the different radiation devices, depending on the state of the therapy program as well as the parametrization and the current status.
- During idle mode, the whole system is suspended.
- The experiment mode permits reconfiguration of type-specific DCU parameters.
- The adjustment mode permits a fine-grained adjustment of treatment data for different therapies.
- In case of erroneous behaviors (e.g., unexpected timeouts), a fail-safe mode releases predefined error-handling procedures to ensure a safe shutdown of the active procedure.

Hence, *reconfiguration* of DCU parameters is restricted by the currently active operation mode. To this end, each DCU periodically reports its current status to the master control and synchronizes itself with the timing master.

Apparently, the DCU software is part of a *safety-critical* system that may potentially harm patients in case of erroneous behaviors (e.g., switches from therapy mode into experiment mode, accidentally allowing any possible parameter setting during radiation).

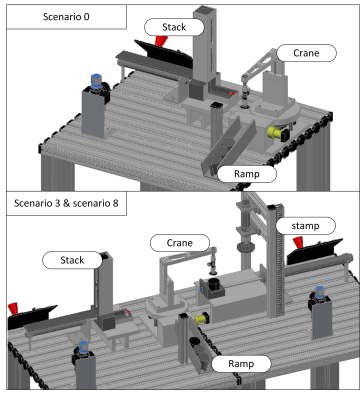


Figure 2.2: Pick & Place Unit

To avoid those situations, direct switches from experiment mode are not allowed, but require an interleaving with the adjustment mode thus ensuring restricted readjustments of parameters for a safe therapy.

To summarize, the network of communicating DCU of the HIT system constitutes a highly configurable, runtime-adaptive, real-time/safety-critical embedded and distributed software system. The software of the DCU is generic (or, variable) thus constituting a family of similar, yet well-distinguished software variants with complex inter-dependencies and constraints between configuration parameters and respective software artifacts.

2.1.2 Automation-Engineering Control Software

As a second example, programmable logic controllers (PLC) are frequently used in software components for *controlling* cyclic executions in today's automation-systems (e.g., based on an IEC 61131-3 compliant run-time environment) [25]. Even today, PLC-based control software is already very complex to develop and to maintain, being responsible for regular machine functionality, as well as diagnostics, exception handling, visualization, and others. In the context of the upcoming Industry-4.0-evolution, the role of software is expected to increase even more in the automation-engineering domain in the near future. In addition, machines and plants are often tailor-made in many industrial branches (e.g., in bottling industry) thus relying on mechatronic products to be *configurable* to individual customers needs, specific platforms and application scenarios.

As a concrete example, Figure 2.2 shows the Pick & Place Unit (PPU), a bench-scale demonstrator of an automation-engineering system [39]. The PPU consists of multiple *communicating* components such as a stack, serving as input storage for arbitrary workpieces (WP, e.g., cans or bottles), a ramp serving as output storage for those WP, a stamp for labeling WP, and a crane for transporting WP by picking and placing them between different working positions. The PPU is able to handle three types of cylindrical WP: light plastic, dark plastic and metal. As illustrated in Figure 2.2, different *versions* of the PPU exist, being derived from the basic PPU for new (initially unforeseen) application scenarios. For instance, the basic version for scenario 0 contains no stamp and each WP is directly transported from the stack to the ramp. In contrast, in the version for scenario 3 and scenario 8, metallic WP are stamped before being transported to the ramp, whereas plastic WP are directly transported to the ramp. Conversely, the

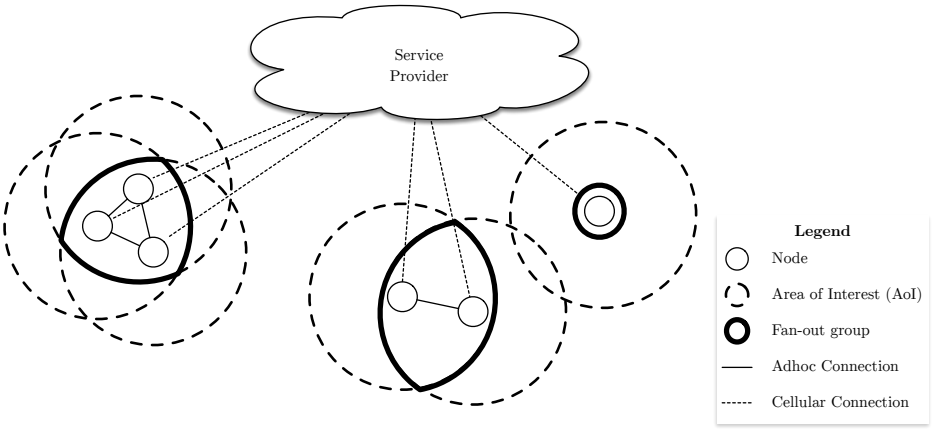


Figure 2.3: Augmented Reality Multi-Player Game Scenario

version for scenario 0 further allows for labeling plastic, where light plastic WP are stamped with adaptive pressure and metallic WP are labeled using standard pressure. As a consequence, the corresponding control-software components of the PPU are not only configurable for different predefined PPU variants, but also *evolve* over time to also handle new versions of the PPU.

To summarize, the different variants of the PPU lead to a high amount of possible runs, combined with an ever-growing number of versions for handling different application scenarios, each imposing a complex interplay between hardware, software and environment.

2.1.3 Mobile Communication Systems Software

As a third example, networks of *communicating* mobile devices being part of an ad-hoc wireless network also constitute a recent application domain of CPS. As a concrete example, Figure 2.3 shows a component of a cloud-based mobile augmented reality (AR) multi-player game scenario [40]. Circles (nodes) correspond to players participating in a game. Each player can move around arbitrarily, carrying devices and items according to a predefined goal. Each player further communicates with a cloud-based service provider via cellular connections in order to continuously deliver game data and disseminating events being relevant for her/his current game context. In addition, players interact with the physical environment and other players located nearby within an Area of Interest (AoI) virtually surrounding a player's physical location. Overlapping AoI may form *Fan-Out Groups* to establish decentralized ad-hoc connections, where the resulting bypassing of the service provider may reduce latency of the cellular network.

To summarize, due to the movements of the different players and their corresponding AoI, the cellular communication network has to be continuously *adapted at runtime* whenever entering or leaving a *Fan-Out Group*. In addition, also the number of players participating in a game may vary over time thus potentially requiring AoI adaptions by the service provider in order to ensure reliable communication and to reduce latency.

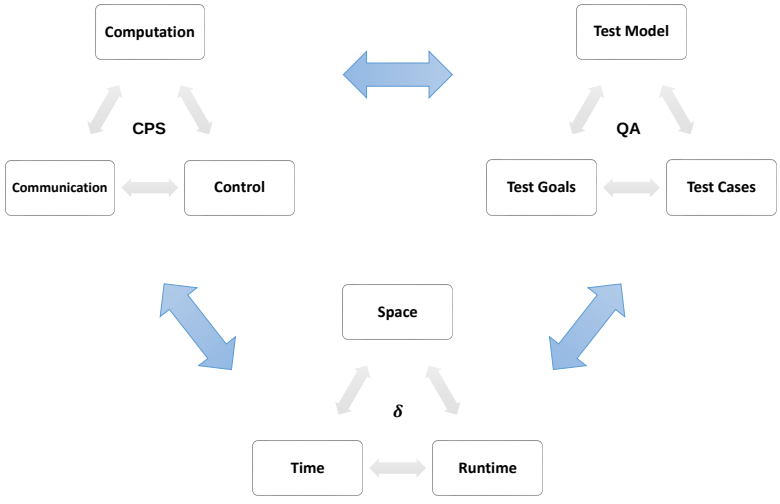


Figure 2.4: Overview on Research Challenges

2.2 Basic Notions and Classification of Research Challenges

Based on the observations made by considering the different examples as described in the previous section, Figure 2.4 shows a general schema for classifying the basic notions as apparent in the different areas of research addressed in this thesis. In particular, the three mutually depending dimensions of cyber-physical systems (CPS), quality assurance (QA), and variability (δ) each, again, has three inter-related facets as will be described in the following.

2.2.1 Cyber-Physical Systems

CPS have recently gained growing attention in many modern application areas. Besides the already mentioned domains of health care, automation engineering and mobile communication networks, CPS further appear in all areas involving smart and intelligent sensor/actuator networks such as transportation, manufacturing, energy, entertainment, environment monitoring, and many others [36]. Basically, CPS comprise a novel class of systems with a tight integration and coupling of purely virtual aspects of embedded software on the one hand, and physical aspects involving natural and human factors on the other hand. To this end, CPS are usually described by three essential characteristics as illustrated in Figure 2.4.

- CPS perform complex *computations*,
- CPS conduct *control* tasks involving feed-back loops with physical processes, and
- CPS consist of distributed and mobile *communication* networks including interacting sensor- and actuator-components.

The contribution summarized in this thesis focus on computation and communication aspects of CPS, whereas the control part is one goal of future research.

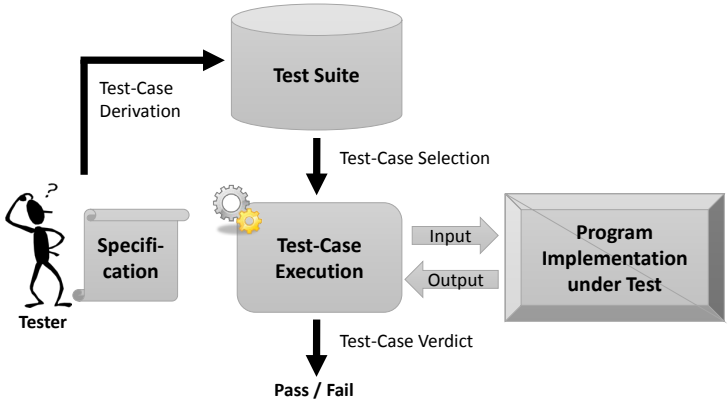


Figure 2.5: Software Testing

2.2.2 Quality Assurance by Model-based Testing

Practices and tools to support systematic quality assurance are an indispensable part of any software project, not only in the context of safety- or mission-critical systems engineering. In practice, *software testing* remains one of the most established and elaborated quality-assurance techniques [29]. This is, amongst others, due to the fact that testing is, in general, directly applicable to most kinds of real-world applications, involving software at different levels of abstraction. In addition, testing offers a light-weight quality-assurance methodology with an explicitly controllable trade-off between effectiveness and efficiency.

The term software testing, in its most general form, describes any activity that is concerned with investigating (and assuring) the *quality* of a given software or program under test. In this thesis, we focus on *dynamic* and *functional* testing at component and integration level. Dynamic testing involves experimental executions of the software under test under controlled conditions and functional testing investigates whether the software produces correct (i.e., expected) outputs for given inputs. Figure 2.5 illustrates the process and respective basic notions of software testing in general as used in this thesis. Based on a *specification* of a *program implementation under test*, a *tester* derives a set of *test cases* into a *test suite*. From this test suite, the tester then repeatedly *selects* test cases for *test-case execution*, by injecting experimental *inputs* into the program implementation under test, and by observing the resulting *outputs* produced by the program as reaction to those inputs. Again, based on the specification, the tester compares the observed output with the output expected for that input, in order to make a verdict for that test case (e.g., stating whether the program *passes* or *fails* the test case thus indicating an error). The two main questions to be addressed when establishing a testing methodology may be, therefore, summarized as

- *How to derive/select test cases for test-case execution?*
- *When to stop the derivation/execution of test cases?*

Depending on the particular testing methodology applied, the different steps of a testing methodology might be conducted (semi-)automatically. For instance, using *model-based testing*, a *test model* is used as specification of the program under test and serves two additional purposes: (1) to automatically generate test cases into a test suite, and (2)

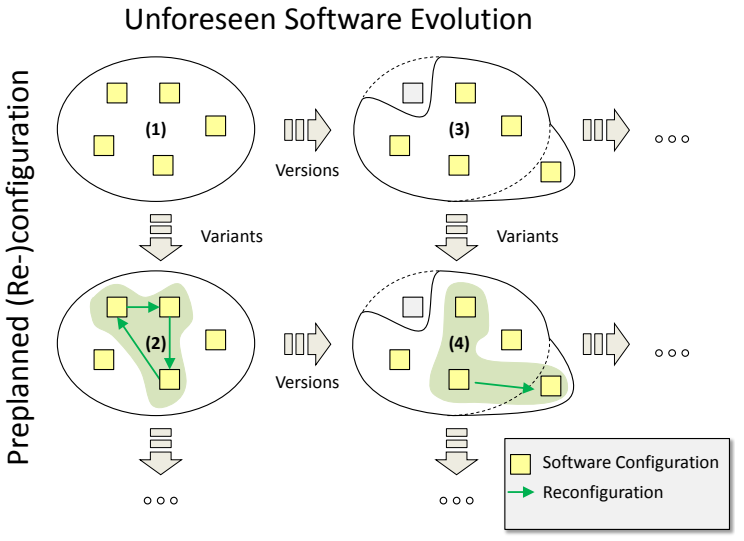


Figure 2.6: Dimensions of Software Variability

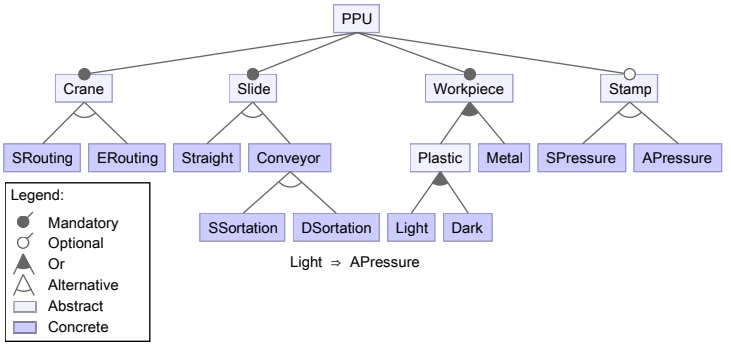
to measure adequacy of test suites by defining *test goals* (e.g., given by coverage criteria on the test model) to be satisfied by a test suite.

2.2.3 Software Variability

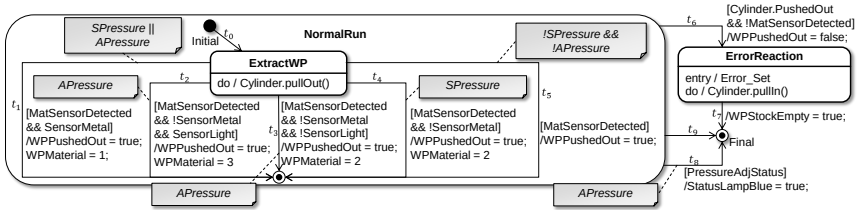
The three different kinds (or, dimensions) of *variability* potentially arising in nowadays software systems, namely *variability in space*, *variability at runtime*, and *variability over time* (cf. Fig.2.4, on the bottom)—together with their potential interplay—are illustrated in more detail in Fig. 2.6, which will be briefly described in the following.

Variability in Space. Each of the aforementioned illustrative examples demonstrates that many of nowadays software systems—independent of their application domains—merely constitutes solitary products developed once for one particular purpose and/or customer. Instead, as illustrated in Fig. 2.6(1), modern software systems often comprise families of similar, yet well-distinguished software *variants*, each corresponding to a particular *configuration*, derived from an underlying core software. In this way, a software system is, at least up to a certain degree, customizable to diverse users' needs, different execution platforms, locally-dependent legal restrictions etc. The set of—implicitly or explicitly specified—available configurations, therefore, tailors the *configuration space* of the software (i.e., the set of valid variants derivable from the core software).

Software product line engineering has become one of the most established methodologies—both in academic research as well industrial practice—to cope with the ever-growing variability in space of modern software systems [15]. The goal of product-line engineering is to develop families of software variants on top of a common core platform by explicitly specifying and systematically exploiting commonality and variability among the different family members in terms of their *features*. In this regard, each feature represents (1) a customer-visible and, therefore, configurable product characteristic within the *problem space* of the software, as well as (2) an increment in functionality within the *solution space* (e.g., a composeable code artifact implementing a particular combination of features). Concerning (1), the set of valid variants offered by product line



(a) PPU Feature Model



(b) Excerpt from the Feature-Annotated State-Machine Model

Figure 2.7: Model-based Specification of the PPU Product-Line

are usually further restricted by configuration constraints imposed by a *feature model* (e.g. a FODA feature diagram [22]). The *mapping* of features organized in a feature model onto solution-space artifacts enables extensive reuse potentials among different members in a family of similar software variants. This additional information may facilitate a remarkable gain in efficiency, as compared to variant-by-variant development. Software product line engineering consists of two interleaved phases, namely *domain engineering* and *application engineering*. During domain engineering, the features of the SPL are identified together with constraints restricting their valid combinations within product configurations. For this purpose, variability modeling approaches like feature models are frequently used as a formal foundation during domain analysis, thus building a sound basis for validating problem-space specifications (e.g., absence of anomalies such as inconsistent feature constraints). Corresponding approaches for automating feature-model analysis use a translation of (usually graphical) variability models into respective (logical) constraint-solving problems, e.g., constraint satisfaction problems (CSP), binary decision diagrams (BDD), and satisfiability problems (SAT) in order to apply constraint-solver capabilities [3].

Concerning solution-space engineering, a wide range of different mechanisms and tools for variability modeling, feature mapping, and variant derivation have been investigated in recent literature [1]. Most importantly, annotation-based approaches enrich configuration-specific solution-space artifacts with so-called presence conditions, usually in terms of propositional formulae over Boolean feature variables to specify the (subsets of) configurations whose variants share this artifact. In this regard, the ifdef macro of the C preprocessor constitutes the most prominent approach of annotation-based (compile-time) variability at source-code level [23], whereas similar mechanisms at design-modeling level (e.g., Statecharts) are usually referred to as template-based [16]. As a concrete example, Figure 2.7 shows the feature diagram (cf. Figure 2.7a) as well as an excerpt from the feature-annotated state-machine model (cf. Figure 2.7b) of the

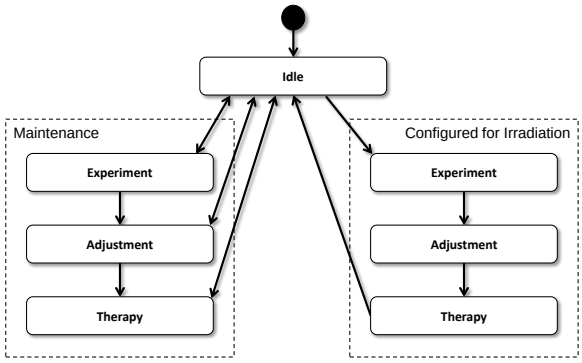


Figure 2.8: Reconfiguration Model of DCU DSPL

PPU, re-engineered as software product line [25]. The state-machine model describes behavioral variability while handling a WP by the PPU, depending on the particular PPU variant selected from the feature model.

Further variability-modeling and variant-derivation techniques that will be considered in this thesis include delta-oriented variability modeling and modality-based variability modeling [5]. The concepts and techniques presented in the main part of this thesis do not focus on a particular implementation methodology of software product lines, but rather aim at configurable (or, variable) software in a more general sense. In order to keep the explanations graspable, the following notions will be used as synonyms throughout this thesis:

- software product line — configurable software — variable software,
- feature — configuration option — (configuration) parameter,
- variant — configuration — product.

Variability at Runtime. Variability in space as apparent in configurable software systems such as software product lines implicitly requires a user or customer to make definite and irreversible configuration decisions once during initial software configuration. *Dynamic* software product lines weaken these restrictions by providing different binding times for configuration decisions (e.g., compile-time vs. run-time) as well as the possibility to reconfigure features throughout the life-cycle of a product [13]. In particular, as illustrated in Fig. 2.6(2), *variability at runtime* by means of reconfigurations from one software variant to another within a predefined configuration (sub-)space (highlighted in green) during execution allows to design and implement runtime- and even self-adaptive software systems within the conceptual framework of software product lines. As a concrete example, Figure 2.8 shows an excerpt from the reconfiguration model of the DCU after being re-engineered as DSPL [24]. The model restricts possible mode switches, depending on whether the system is currently under maintenance (i.e., no restrictions), or running an irradiation procedure (i.e., requiring a predefined sequence of mode switches).

Variability over Time. Both variability in space and variability at runtime constitute extents of so-called *preplanned* variability within a-priori predefined configuration spaces and reconfiguration options. However, in case of unforeseen stakeholders needs

(e.g., due to new, and initially unknown, user requirements, legal restrictions and execution platforms), software has to undergo continuous *software evolution* in terms of a-posteriori updates, bug-fixes etc. — both applied offline as well as even online — in order to cope with those ever-changing circumstances [30]. In practice, those countermeasures against software aging are often conducted in an ad-hoc manner thus lacking a proper documentation and change-impact analysis. This becomes even worse in case of evolving configurable software as illustrated in Fig. 2.6(3) as changes made to some artifact of an existing product family may (intentionally or erroneously) affect various variants at once. As illustrated in Fig. 2.2, the PPU does not only come in different *variants* as being derivable from the feature model in Figure 2.7a, but the whole PPU product line has to undergo a continuous evolution to new *versions* (e.g., due to adding new features such as a stamp).

Finally, as depicted in Figure 2.6(4), all of these three—more or less—orthogonal dimensions of variability may arbitrary interleave throughout the life-cycle of reconfigurable, evolving software systems (e.g., leading to continuously evolving dynamic software product lines).

Chapter 3

Model-based Quality Assurance of Variability in Space

This chapter summarizes concepts and techniques for model-based quality assurance of CPS in the presence of variability *in space*, namely in terms of configurable software as, for instance, apparent in software product lines. The contributions presented include an automated test-suite generation methodology for efficient white-box as well as model-based (black-box) test coverage of configurable software, a conceptual framework for lifting techniques for test-suite minimization and test prioritization to configurable software, and a theoretical framework to formally reason about the interplay between variability in space and compositionality of testing in systems consisting of multiple communicating and interacting components.

3.1 Test-Suite Generation for Configurable Software

Publications. *The contents summarized in this section have been published in Bürdek et al. [12] as well as in Lochau et al. [25]. Reprints of these publications can be found in Appendix A.*

Classification of Contributions. *CPS: Computation; QA: Test Models, Test Cases, Test Goals; δ: Variability in Space.*

Summary. Automated derivation of a sufficient set of *test cases* into a *test suite* satisfying a given set of *test goals* is a complicated and computationally expensive task. Most recent attempts employ model checkers for automated coverage-driven test-case generation, by utilizing counter examples delivered by the model checker as witnesses for non-reachability of a given test goal [6]. A counter example, therefore, constitutes an (abstract) test case (e.g., program input) whose execution on a given program under test is able to reach the particular test goal (e.g., a program location) under consideration. Hence, covering a (usually very large) set of test goals (e.g., as imposed by a code-coverage criterion such as basic-block coverage or condition coverage) requires to repeatedly generate further test cases into a test suite until the coverage criterion is finally satisfied (at least up to a certain degree). To this end, repetitive reachability-analysis queries are to be performed, where every new query potentially leads to an exhaustive state-space exploration by the model checker.

In the presence of variability in space (e.g., as in software product lines), this problem becomes even worse: instead of only covering one single software system, the generated test suite has to achieve sufficient test-goal coverage on a whole family of similar, yet well-distinguished software variants, usually derived from a common core code base. However, applying a variant-by-variant test-suite generation strategy to eventually achieve coverage of a whole family of software variants is often infeasible for realistic product lines in practice. This is due to the fact that the number of possible configurations and, therefore, the number of implementation variants derivable from the core software system, grows exponentially in the number of configuration options (e.g., Boolean features in case of software product lines). In addition, due to the (intentionally) very high degree of similarity among the different variants, a variant-by-variant test generation approach would potentially lead to a high fraction of redundant test-generator calls, yielding many duplicated (abstract) test cases from different variants.

To handle these two essential challenges in multi-goal test-suite generation for configurable software systems, the proposed approach facilitates two orthogonal *reuse strategies* for reachability information delivered by a model checker.

- **Reuse among test goals.** The reachability information for the state space of a configurable program, obtained while generating a test case covering a particular test goal, is *reused for all other test goals* which are also reached by this test case.
- **Reuse among variants.** The reachability information for the state space of the configurable program, obtained while generating a test case covering a particular test goal, is *reused for all other program variants* sharing the parts in the state space reached by this test case.

The first reuse technique has already been explored in recent tools for test-suite generation for single software systems [7]. In contrast, the second reuse technique of systematically exploring similarities among different software variants instead of considering every variant one-by-one is, in general, referred to as *family-based* product-line analysis strategy [33]. The proposed methodology thus combines both dimensions of reuse in a compatible (i.e., arbitrarily interleaved) way as part of a family-based white-box test-suite generation algorithm for efficiently covering all variants of a configurable software. The output of the algorithm is a so-called *product-line test suite* which guarantees coverage of every test goal on all software variants derivable from configurable software, in which this particular goal is reachable.

The proposed approach utilizes symbolic model-checking capabilities of recent software model checkers by exploiting counter examples for non-reachability of a given test goal, serving as (abstract) test input. To this end, the approach employs so-called *runtime variability* (or, *variability encoding* [31]), by representing configuration-specific parts of the code using conditional statements over feature expressions (so-called presence conditions). This encoding facilitates a software model checker to keep track of control- and data-flow dependencies among program paths and respective configuration constraints for those paths during state-space exploration. In particular, the conjugated presence conditions, aggregated within path conditions of counter examples obtained for non-reachability of a given test goal, symbolically represent the set of software configurations for which the derived test case is (re-)usable. In order to ensure each test goal to be eventually covered in all configurations in which the goal is reachable, further reachability queries are repeatedly invoked for the same goal, but with the negated set of already covered configurations together with the feature-model constraints as precondition, until no further counter example is reported by the model checker. This *blocking-clause* technique is frequently used in all-solutions constraint-satisfaction problems. The interleaving of this second reuse technique with the first reuse technique (i.e.,

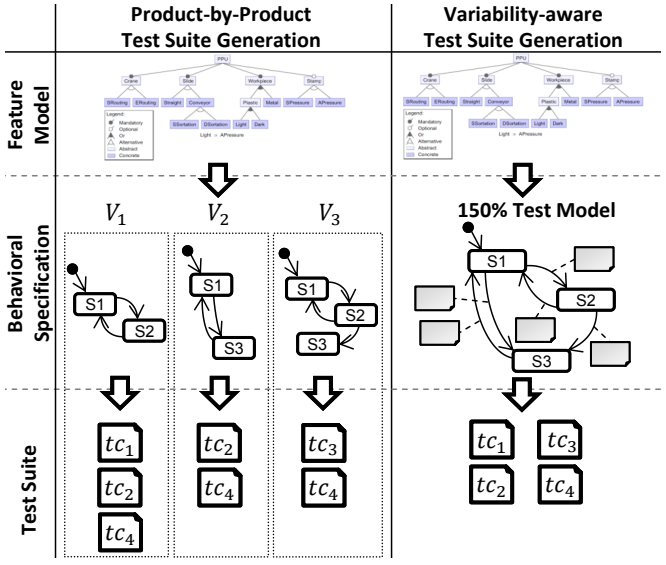


Figure 3.1: Model-based Test-Suite Generation for Configurable Software

reuse of prefixes of already explored program paths for covering other test goals also located on this path) leads to an incremental, test-goal-driven state-space exploration of the whole configurable software system.

The described approach is, in general, independent of the particular testing methodology and test-suite generation technique. This has been successfully demonstrated by further applying the approach to model-based (black-box) test-suite generation for software product lines apparent in the automation-engineering domain (e.g., the PPU case study). For instance, a metal WP is handled equally in most variants such that corresponding test cases are (re-)useable among those variants. As further example, test cases derived for exercising the error handling are even (re-)usable for all PPU variants.

The overall infrastructure is illustrated in Figure 3.1. The test-model specification for PLC control software using an IEC 61131-3 compliant run-time environment are given as UML-like input/output labeled state machines. A test case, therefore, constitutes a finite sequence of input/output actions (or, pairs of actions and reactions representing expected interactions between the system or component and the contextual environment). For variability modeling, transitions are annotated by presence conditions over feature variables, as organized in a feature diagram using FODA notation, thus resulting in a so-called 150% product-line test model superimposing all test-model variants. In this testing scenario, coverage criteria for test-suite derivation are defined with respect to structural elements of the test model (e.g., state coverage and transition coverage).

Tool Implementation. The tool implementation of the white-box test-suite generator is based on CPA/TIGER [7], an extension of the symbolic software model checker CPACHECKER [8] for coverage-driven test-case generation from C programs. The tool receives as inputs (1) a configurable software implemented in C enriched with Boolean feature parameters, (2) a feature model (i.e., a propositional formula over features), and (3) a coverage criterion expressed in the FShell Query Language (FQL), a DSL for specifying code-coverage criteria provided by CPA/TIGER.

The tool implementation of the model-based test-suite generator for state-machine test models of software product lines in the automation-engineering domain is based on the LTL model checker SPIN and the SAT solver SAT4J. Feature diagrams in FODA

notation are imported from the tool FeatureIDE and UML state-machine models are imported from PapyrusUML, where presence conditions are annotated as comments attached to respective model artifacts.

Evaluation Results. The experimental evaluation of the white-box testing approach consists of various experiments considering sample product-line implementations. The evaluation results show remarkable efficiency improvements (concerning both computational efforts for test-suite generation as well as the resulting number of test cases) of the novel methodology, as compared to test-suite generation strategy without systematic reuse of reachability information. However, the computational overhead caused by combining both reuse strategies may potentially lead to an overall increase of CPU time or even an increase of timeouts (i.e., test goals falsely deemed unreachable).

The experimental evaluation of the model-based testing approach consists of a successful application of the developed tool to the PPU case study. The results show that the required CPU time for achieving complete test coverage of all test-model variants decreases by about 50% and the number of test cases even by a factor of about 7, as compared to the variant-by-variant approach.

3.2 Test-Suite Optimization and Test Prioritization for Configurable Software

Publication. *The contents summarized in this section have been published in Baller et al. [2]. A reprint of this publication can be found in Appendix A.*

Classification of Contributions. *CPS: Computation; QA: Test Cases, Test Goals; δ: Variability in Space.*

Summary. The output of the test-suite generator for configurable software as described in the previous section is a so-called *complete product-line test suite* which covers every test goal on every program variant in which the goal is reachable, by at least one test case being applicable to that variant. One purpose of the two reuse strategies is to achieve complete product-line coverage with a preferably small number of test cases in order to reduce the subsequent test-execution effort. In this regard, *test-suite minimization* aims at eliminating redundant test cases from existing test suites such that the reduced test suite still achieves complete test-goal coverage, but with a minimum number of test cases. The resulting optimization problem can be reduced to the *minimum set-cover problem* which is known to be NP-complete [21]. Most existing approaches for heuristically approximating optimal solutions for the test-suite minimization problem have several limitations, especially with respect to configurable software testing.

- Most existing approaches are limited to handling test suites derived for covering test goals in single-variant software.
- Most existing approaches do not take into account that each individual selection of test cases may cause different testing costs and profits.

To address these limitations, the proposed framework for test-suite minimization and test prioritization for configurable software takes into account dependencies between test cases, test goals *and* software variants as apparent in a complete product-line test suite obtained from a family-based test-suite generator. In addition, the framework incorporates additional fine-grained information denoting multiple (potentially conflicting) cost/profit objectives for prioritizing test cases, test goals and variants under test.

An excerpt from a multi-objective test-suite optimization problem for configurable software is shown in Figure 3.2, where test goals are referred to by the more general term

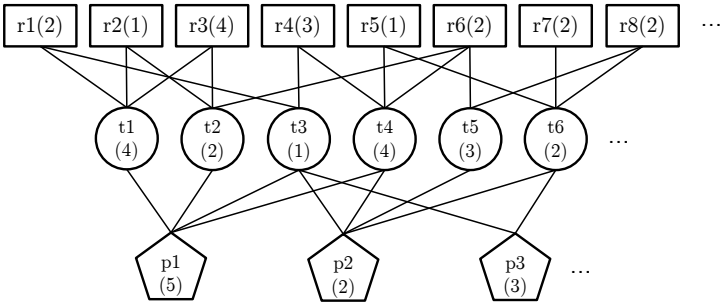


Figure 3.2: Test-Suite Optimization Problem of Configurable Software [2]

of *test requirements* in the following. Here, test goals constitute one possible instance of the more generalized notion of *test requirements*. In particular, the example has eight test requirements $(r1, r2, \ldots, r8)$ and a corresponding test suite containing 6 test cases $(t1, t2, \ldots, t6)$, where a line between a requirement and a test case denotes that the requirement is *satisfied* by that test case (e.g., a test goal is covered/reached by that test case).

In this example, the test cases $t1$ and $t4$ constitute a *minimal* test suite as both together suffice to satisfy every given test requirement. However, in the given example, the test suite is generated for complete coverage of a configurable software, consisting of three variants (or, product configurations $p1, p2, p3$). Lines between test cases and variants thus denote that a particular test case is *applicable* to a variant. In this regard, the test cases $t1$ and $t4$ are not sufficient to cover every test requirement on every variant (e.g., test case $t6$ is the only test case satisfying requirement $r7$ on variant $p3$).

In addition to the dependencies between test requirements and test cases as well as between test cases and variants under test, the example further defines *weights* (positive integer values attached in brackets) for all three kinds of testing artifacts, providing further fine-grained information by means of

- *test-case costs* quantifying the estimated effort for executing a selected test case,
- *test-goal profits* quantifying the estimated benefit gained from satisfying a certain requirement by a test case (e.g., according to the fault criticality of the requirement)
- *variant costs* quantifying the estimated effort for deriving a particular product configuration for test execution.

Based on this additional information, multi-objective test-suite optimization is concerned with selecting a set of test cases that minimizes the overall testing costs and maximizes the overall testing profits. For instance, selecting the three test cases $t2$, $t3$ and $t4$ would cause overall test-case costs of only 7 as compared to 8 resulting from selecting $t1$ and $t4$ as mentioned before. The resulting *weighted minimum set cover problem* can be further refined to a *partial minimum set cover problem*, by imposing constraints in terms of *upper bounds* for the overall costs and *lower bounds* for the overall profits. For instance, by setting a profit bound of 11 (instead of 13) in our example, requirement $r1$ might be ignored thus making test case $t3$ redundant. In addition, the selection of test requirements and test cases further influences the overall testing costs caused by the costs of variants required for executing the selected test cases. Hence, the challenge of combining those extensions to the original test-suite minimization problem

with the notion of complete coverage of configurable software may be summarized as follows.

> *Which test cases to execute on which product configurations in order to maximize testing profits under minimal testing costs?*

Note that this generalized problem statement is, in general, independent of the particular testing methodology under consideration.

Tool Implementation. The constrained multi-objective optimization problem underlying the test-suite optimization problem for configurable software is a computationally complex problem. The tool implementation comprises two competitive techniques for approximating (presumably) optimal solutions. The first approach is based on an encoding of the problem as an Integer linear program (ILP) in order to utilize off-the-shelf ILP solvers for approximating optimal solutions. The second approach consists of an incremental greedy-based heuristic prioritization algorithm for selecting sequences of variants under test to be tested for achieving optimal profits under reduced costs in a presumably minimal amount of time.

Evaluation Results. The experimental evaluation concerning the comparison of both techniques has been performed on synthetic data sets with numbers of test requirements, test goals and variants under test ranging between 50 and 500 elements as well as different extents of dependencies between those elements, and Gaussian distribution of cost and profit values. The efficiency improvements of the greedy-based approach as compared to the ILP-based approach amounts to a factor of about 21.49 (i.e., from 15 hours to 41 minutes for the most complex data set being processable by ILP without running into a timeout). Conversely, loss in precision of the greedy-based approach is down to 68.80% as compared to ILP, and increases with increasing complexity of input data sets.

3.3 Compositional Testing Theory for Variable Software

Publication. *The contents summarized in this section have been published in Luthmann et al. [27]. A reprint of this publication can be found in Appendix A.*

Classification of Contributions. *CPS: Computation, Communication; QA: Test Models; δ: Variability in Space.*

Summary. From a theoretical point of view, software testing, in general, is concerned with establishing a behavioral *conformance relation* between a specification s and an implementation i under test [18]. Concerning formal approaches to reasoning about theoretical aspects of model-based testing of software systems with multiple interacting components in particular, both the component specifications and the component implementations are usually represented as (variations of) labeled transition systems (LTS) [9], where the LTS of the implementation components are unknown (black-box assumption). In this regard, two different notions of behavioral conformance may be distinguished in recent research on model-based testing.

- *Extensional* characterizations of behavioral conformance rely on *observational equivalence relations* [18, 17]. An implementation i of a component is observationally equivalent to a specification s if no observer process (tester) exists that is ever able to distinguish behaviors (e.g., sequences of actions) observed at i from those allowed by s.

- *Intentional* characterizations of behavioral conformance rely on *alternating simulation relations* assuming both component specifications s as well as implementations i to be represented as input/output labeled transition systems (IOLTS) [38, 20]. Based on this representation, *test cases* are derived from specifications s, given as (alternating) sequences of controllable input actions and observable output actions, and being applied to implementations i.

The input/output conformance (**ioco**) relation on IOLTS, initially proposed by Tretmans [35], is one of the most prominent conformance testing theory, combining both extensional and intentional aspects into one formal framework. Intuitively, the **ioco** relation, formally defined as

$$i \,\mathbf{ioco}\, s : \;\Leftrightarrow\; \forall \sigma \in \mathit{Straces}(s) : \mathit{Out}(i \,\mathbf{after}\, \sigma) \subseteq \mathit{Out}(s \,\mathbf{after}\, \sigma),$$

holds between implementation i and specification s if the set of possible output actions (*Out*) observable at implementation i following (**after**) each possible alternating I/O sequence σ specified in s is allowed by s. To rule out trivial implementations never showing any outputs, the notion of suspension traces (*Straces*) includes the special *quiescent* action δ denoting (and, therefore, explicitly permitting) the absence of any output in a certain state.

In recent past, a wide range of formal properties of, and extensions to, **ioco** have been investigated. Nevertheless, concerning model-based testing of component-based software in the presence of variability, **ioco** still shows several essential weaknesses.

- The **ioco** relation permits underspecification in two ways. First, input behaviors being unspecified in s may be implemented arbitrarily in implementation i, thus implicitly relying on optimistic environmental assumptions. However, negative testing in a pessimistic setting would require a distinction between critical and uncritical unintended input behaviors, which is not supported by **ioco**. Second, for non-deterministic input/output behaviors of specification s, **ioco** requires implementation i to show at most output behaviors being permitted by s. Furthermore, the notion of quiescence (i.e., observable absence of any outputs) enforces i to show at least one of the output behaviors of s (if any). However, no explicit distinction between obligatory and allowed output behaviors is made by **ioco**, which is of particular importance in case of behavioral variability.

- The **ioco** relation constitutes a special kind of alternating simulation relation between specification s and implementation i. Unfortunately, although being a crucial property for extensional testing relations, **ioco** is not a preorder.

- The **ioco** relation lacks a unified notion of component-based systems testing, being compatible with potential solutions that solve the aforementioned weaknesses.

The proposed solution to handle these weaknesses of **ioco** utilizes *Modal Interface Automata with Input Refusals (IR-MIA)* as novel behavioral formalism for both component specifications s and the implementation under test i in component-based variable software systems. IR-MIA are based on Modal Interface Automata (MIA) [10], being a combination of Interface Automata (i.e., I/O automata permitting underspecified input behaviors) and (I/O-labeled) Modal Transitions Systems (i. e., LTS with distinct mandatory and optional transition relations). In this regard, the modal refinement relation $\sqsubseteq$ on IR-MIA serves as variant-derivation mechanism for specification variants $s' \sqsubseteq s$ as well as implementation variants $i' \sqsubseteq i$ both constituting IOLTS as usual. Like MIA, IR-MIA permit both optimistic and pessimistic environmental assumptions as well

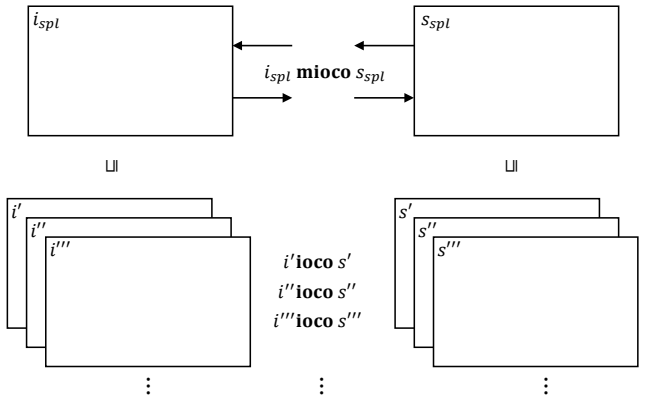


Figure 3.3: Modal Input/Output Conformance is preserved under Modal Refinement.

as non-deterministic input/output behaviors. To this end, the *universal state* of MIA (allowing every possible behavior) is re-interpreted as *failure state* in IR-MIA, rejecting every input behavior. This construction allows to distinguish between unspecified, yet uncritical (i.e., to be ignored) environmental inputs from those being unspecified, but critical and, therefore, to be *refused* by the implementation. In order to handle behavioral variability in IR-MIA, modal refinement further allows distinguishing obligatory from allowed output behaviors, as well as between implicitly underspecified and explicitly forbidden input behaviors. To summarize, the resulting testing theory on IR-MIA unifies positive and negative conformance testing with optimistic and pessimistic environmental assumptions and behavioral variability.

Theoretical Results. The I/O conformance relation defined on IR-MIA, called **modal-irioco**, enjoys several essential properties, especially with respect to component-based systems testing. The main theoretical properties of **modal-irioco** (or, **mioco** for short) can be summarized as follows.

- **Correctness.** The correctness claim for **modal-irioco** is based on the fact that **mioco** is preserved under modal refinement as illustrated in Figure 3.3. In particular, *soundness* states that if **mioco** holds between a (variable) specification s_{spl} and its (variable) implementation i_{spl} (e.g., a software product line), both given as IR-MIA, then every derivable implementation variant $i' \sqsubseteq i$ is conforming to some specification variant $s' \sqsubseteq s$ with respect to the **ioco** relation on the corresponding IOLTS. Conversely, *completeness* states that if **mioco** does not hold, then there exists at least one implementation variant $i'' \sqsubseteq i$ for which no conforming specification variant $s'' \sqsubseteq s$ exists.

- **Preorder.** The **mioco** relation is a preorder on the subset of input-enabled IR-MIA. Input-enabledness can be easily achieved for any given IR-MIA by a canonical, semantic-preserving construction called input completion.

- **Compositionality.** The compositionality claim for **modal-irioco** with respect to parallel composition with multi-cast and hiding is illustrated in Figure 3.4. Given two component specifications s_1 and s_2 and respective component implementations i_1 and i_2 such that **mioco** holds between i_1 and s_1 as well as between i_2 and s_2, then it follows that **mioco** also holds between the implementation re-

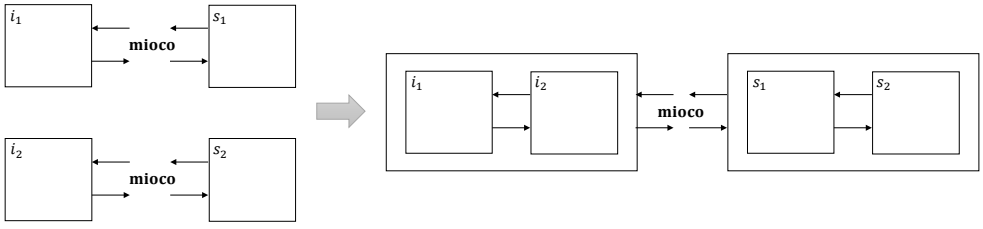


Figure 3.4: Compositionality of Modal Input/Output Conformance Testing.

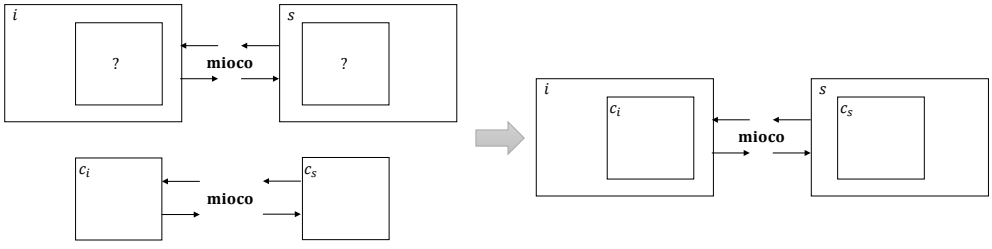


Figure 3.5: Decomposition of Modal Input/Output Conformance Testing.

sulting from the parallel composition of i_1 and i_2 and the specification resulting from the parallel composition of s_1 and s_2.

- **Decompositionality.** This claim for **modal-irioco** allows for decomposing modal input/output conformance testing as illustrated in Figure 3.5. To this end, the quotient operator on IR-MIA serves as the inverse to parallel composition thus supporting environmental synthesis for component-based testing in contexts, also known as the unknown-component problem. As a result, implementation i and specification s may be decomposed into components c_i and c_s which can be tested in separate. Hence, **mioco** holds between i and s if (1) **mioco** holds between i and s without c_i and c_s and (2) **mioco** holds between c_i and c_s.

Chapter 4

Model-based Quality Assurance of Variability at Runtime

This chapter summarizes selected research on model-based quality assurance of software systems with variability at runtime. Both presented publications are mainly concerned with extending formal modeling and automated validation techniques for configurable software to (runtime-) reconfigurable software systems.

4.1 Complex Binding-Time Constraints in Reconfigurable Software

Publication. *The contents summarized in this section have been published in Lochau et al. [24]. A reprint of this publication can be found in Appendix A.*

Classification of Contributions. *CPS: Computation, Communication, Control; QA: Test Models; δ: Variability at Runtime.*

Summary. Besides variability in space as apparent in (statically) configurable software as described in the previous chapter, many modern application domains further tend to exhibit run-time variability in terms of dynamically reconfigurable software parameters. To this end, predefined reconfiguration options are introduced into the software and potentially remain unbound (or, resettable) until, or even after, the final installation and first activation of the system. Concerning product-line engineering in particular, potential run-time adaptation scenarios anticipated already during domain analysis require dynamic re-allocations of feature parameters and related solution-space artifacts.

In this regard, dynamic software product-line engineering aims at lifting elaborated design and implementation principles already being well-established for product-line engineering, to also developing highly-reconfigurable runtime-adaptive systems in a comprehensive and feature-oriented way [4]. Hence, in order to unify variability in space and variability over time within the same conceptual framework, dynamic software product-line engineering extends classical product-line engineering by two essential concepts.

- The variability in a dynamic software product line is bound incrementally in a stepwise manner, rather than being configured in one big step during application

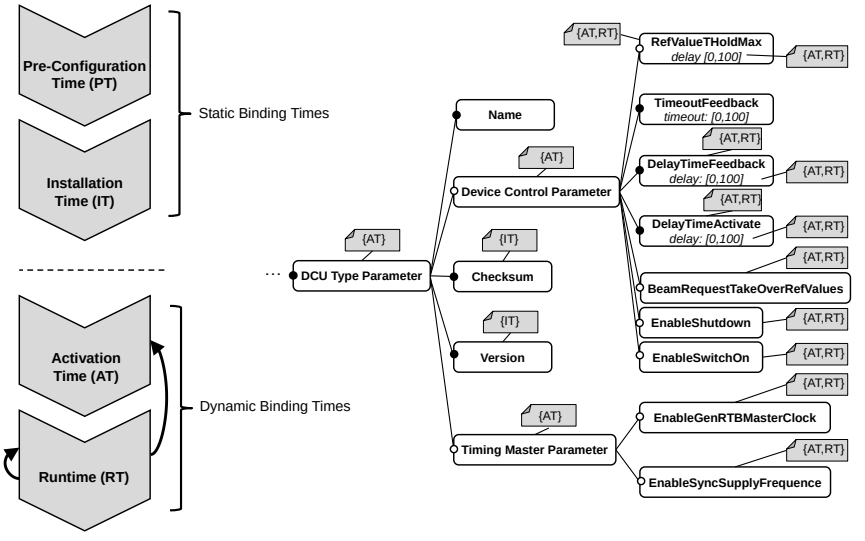


Figure 4.1: Feature Model extended with Complex Binding Time Constraints

engineering. The correct ordering of (and, implicit dependencies among) configuration decisions emerging throughout the resulting *staged (pre-)configuration* process depends on the declared *binding time* of each individual feature.

- The variability of dynamic software product lines is separated into features with *static* and with *dynamic* binding times. Static features are (pre-)configured once during initial product derivation (e.g., prior to installation and first activation), whereas dynamic features are (re-)configurable throughout the entire product life cycle, and even at runtime.

In the context of safety- and mission-critical systems (e.g., medical-device software like in the DCU example), a precise specification formalism and accompanying validation techniques and tools are required to ensure, up to a reasonable extent, correctness properties to hold for every derivable system variant prior to its initial activation. In addition, the combination of both dimensions of variability (i.e., in space and at runtime) imposes further challenges including new kinds of potential threats to validity.

As a consequence, new kinds of configuration constraints arise during problem-space modeling for dynamic software-product lines, including logical constraints among features as usual, as well as temporal constraints to hold for staged configuration processes with multiple binding times, as well as complex constraints on the validity of reconfiguration behaviors. The proposed extensions to domain feature modeling to express those constraints are illustrated in Figure 4.1, showing an excerpt of the configuration model for the DCU system re-engineered as dynamic software product line. Each domain feature may be annotated with multiple binding times, each corresponding to (static or dynamic) configuration stages (e.g., Installation Time, Runtime etc.), on which a global ordering is defined (as shown on the right). In addition, complex binding-time constraints can be expressed by relating *logical* configuration constraints and/or *causal* dependencies among configuration steps during a staged configuration process (e.g., sub-features have to be configured during the same, or some later stage as their parent feature). In addition to those extended feature-model constraints, automata-based reconfiguration models as already shown in Figure 2.8 impose *temporal* constraints on the

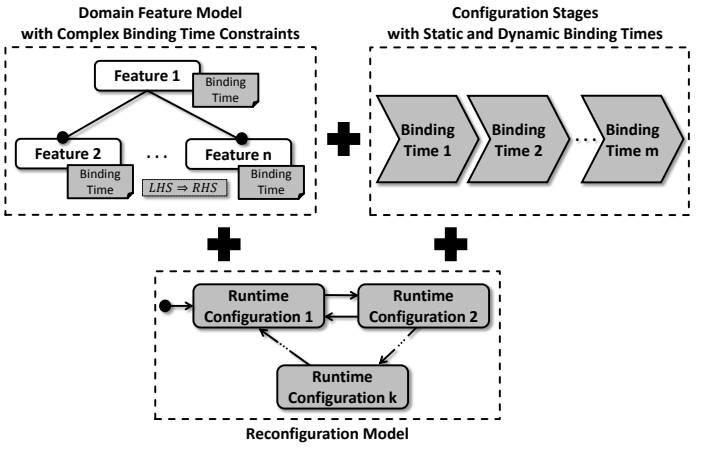


Figure 4.2: Model-based DSPL Specification and Verification Framework

validity of (generally non-terminating) reconfiguration processes, potentially occurring throughout the entire life-cycle of the software system.

To summarize, the different components of the modeling and validation framework for staged reconfiguration processes with complex binding time constraints is depicted in Figure 4.2, conservatively extending domain-modeling concepts as used in classical product-line engineering. As a consequence of those extensions, new kinds of anomalies might arise during domain engineering, potentially leading a software system that eventually fails in adapting itself to certain contextual situations, which may have catastrophic consequences. As a first solution, the framework comprises an efficient and automated validation methodology based on existing constraint-solving techniques. However, compared to classical product-line domain analysis which is usually based on Boolean constraint solving, dynamic product-line validation further requires capabilities going beyond stateless constraint solving for checking temporal properties of reconfiguration processes (e.g., absence of reconfiguration deadlocks). In particular, the following properties are automatically verifiable for a given specification within the proposed framework.

- **Proper Initialization.** This property requires that for a given specification, there exists at least one valid initial staged configuration process finally ending up in a complete configuration satisfying the feature-model constraints.
- **Reachability.** This property requires that for a given specification, all states of the reconfiguration automaton are eventually reachable by a valid reconfiguration process of the system.
- **Progress/Liveness.** This property requires that for a given specification, no valid reconfiguration process will ever get stuck due to any constraints.

Tool Implementation. The tool implementation of the described framework supports FeatureIDE for creating and editing feature diagrams. Those diagrams are then transformed into an existing EMF meta-model for plain feature models, which is extended with binding-time information and corresponding constraints. The model transformation has been implemented using eMoflon. For analysis of non-temporal properties, an

off-the-shelf SAT-solver can be used. Second, all logical, causal and temporal constraints of a given specification are encoded into one state-machine model which is further translated into the Promela language. In this way, the explicit-state model-checker SPIN can be applied for checking temporal properties of staged configuration and reconfiguration processes.

Evaluation Results. The experimental evaluation concerning applicability to the DCU system shows that the proposed modeling and validation technique is, in general, applicable to real-world systems and is scalable to industrial-strength sizes, except for the validation of progress/liveness properties, frequently leading to timeouts. In particular, the constraint-solver-based validation is very efficient but naturally suffers from its limited expressive power. In contrast, the SPIN-based approach is sufficiently expressive, but potentially runs into scalability issues in case of larger-scaled real-world systems. Hence, pursuing both approaches in combination seems to allow for a flexible trade-off between efficiency and accuracy.

4.2 Multi-Instantiation Constraints in Reconfigurable Software

Publication. *The contents summarized in this section have been published in Weckesser et al. [40]. A reprint of this publication can be found in Appendix A.*

Classification of Contributions. *CPS: Computation, Communication; QA: Test Models; δ: Variability in Space, Variability at Runtime.*

Summary. Feature models based on the FODA feature-diagram notation such as shown in Figure 2.7a provide rich and well-established graphical notations for specifying variability of configurable software systems such as software product lines [22]. In addition, various techniques and tools have been developed for automating feature-model analysis and validation, including feature-model consistency (i.e., satisfiability of configuration constraints) and further crucial anomaly-detection capabilities (e.g., absence of dead and core features) [3].

Feature diagrams allow for specifying custom-tailored configuration spaces of configurable software systems in terms of configuration constraints denoted as (restricted forms of) propositional formulae over a finite set of user-visible Boolean feature variables. A valid configuration of a FODA-like feature diagram, therefore, consists of a feature combination (i.e., a subset of selected features) satisfying all configuration constraints. Nevertheless, many constraints to be imposed on configurations as well as reconfigurations of nowadays real-world applications are not properly expressible in terms of selection/deselection choices on binary configuration options (i.e., presence/absence decisions for Boolean features). In order to cope with more complicated configuration options and respective constraints, various extensions to feature models have been proposed [3]. In particular, two key extensions to feature diagrams are frequently considered in recent literature.

- *Feature attributes* and respective (non-Boolean) constraint expressions, supporting any type of *value domain* for configuration options beyond Boolean. For instance, feature attributes allow for denoting additional user-configurable numerical quality properties of features such as adjustable timeout values in case of safety-critical systems [24].
- *Feature multiplicities* and respective cardinality constraints, supporting selections of *multiple instances* of configuration options. Similar to UML-like cardinality annotations, the selection of a feature instance includes *recursive copies* (or, clones)

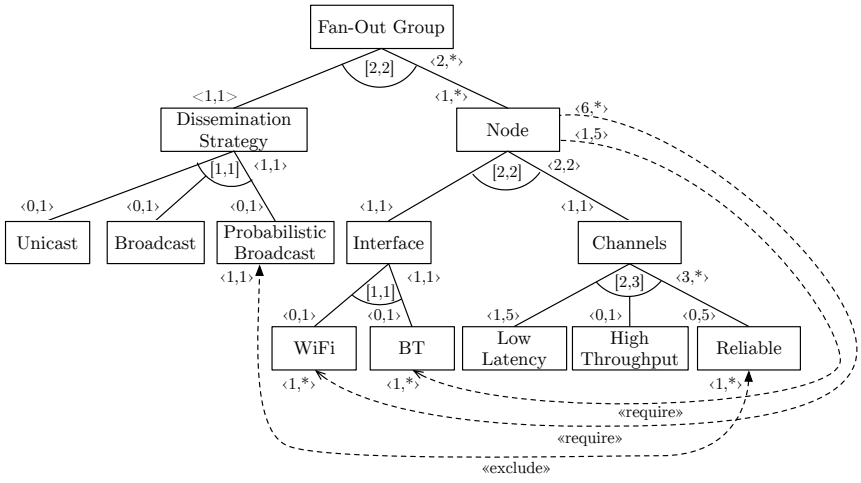


Figure 4.3: Cardinality-based Feature Model for Fan-Out Group Configuration of AR Game Event Dissemination System

of their corresponding sub-trees, being configurable for this individual feature instance.

Semantically, those concepts impose non-trivial extensions to the notion of product configuration, (i.e., arbitrary value domains of configuration parameters and multi-sets of feature selections) both potentially yielding infinite configuration spaces. As a result, both extensions complicate feature-model semantics which makes tool support for automated consistency checking even more crucial for extended feature-modeling concepts to become accepted in practice. Concerning feature attributes, various promising attempts already exist for specifying and automatically analyzing non-Boolean feature constraints, mostly due to the ever-growing availability of mature SMT solvers. In contrast, semantics of cardinality-based feature models lack a direct correspondence to recent constraint-solver theories, but rather require a more in-depth treatment and problem encoding to facilitate automated reasoning.

Nevertheless, cardinality-based feature-modeling is emerging in nowadays application domains. For instance, in many modern communication systems, not only the *type*, but also the *amount* of particular resources is explicitly configurable by the customer, especially in terms of (virtually) unrestricted resources as apparent in the context of cloud-based application scenarios. As a concrete example, the augmented reality multiplayer game scenario already introduced in Figure 2.3 constitutes a runtime-adaptive system including dynamic reconfigurations that do not only affect the *presence or absence* of features, but also the *available amount* (instances) of configurable resources (e.g., players and fan-out groups).

Figure 4.3 shows a cardinality-based feature model for (re-)configuration of fan-out groups. Similar to the FODA notation, configuration parameters (features) reside in a tree-like diagram, denoting a feature-decomposition hierarchy. Cardinality-based feature diagrams extend FODA-like feature diagrams with cardinality-interval annotations (l, u), where $l \in \mathbb{N}_0$ denotes a lower bound and $u \in \mathbb{N}_0 \cup \{*\}$, $l \leq u$, denotes an upper bound on the possible *number* of feature instances. The special symbol "$*$" denotes an *unbounded* cardinality thus permitting an unrestricted maximum number of feature instances. Based on this notion, cardinality-based feature diagrams as shown in Figure 4.3 offer the following modeling concepts.

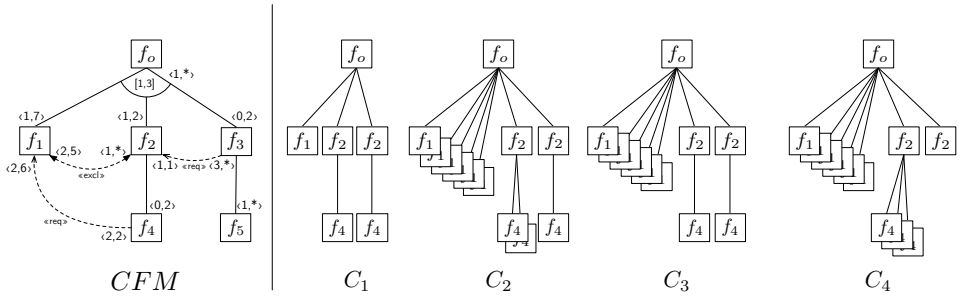


Figure 4.4: Sample Configurations of a Cardinality-based Feature Model

- *Feature instance cardinality*, annotated as $\langle l, u\rangle$ on the left-most position on top of a feature node, defines the minimum and maximum number of a feature instance selectable from the sub-tree clone of its parent feature instance. For instance, $\langle 1, 1\rangle$ allows for exactly one *Dissemination Strategy*, whereas $\langle 1, *\rangle$ denotes that arbitrary many, but at least one *Node* must be contained in every *Fan-Out Group*.

- *Feature group type cardinality*, annotated as $[l, u]$ at group arcs, defines the minimum and maximum number of types of feature instances from the set of all direct sub-features of a selected feature instance. For instance, $[1, 1]$ denotes that either instances of *WiFi* or of *BT* must be selected for the *Interface*, whereas $[2, 3]$ denotes that at least two types of *Channels* from the given three options must be instantiated in a *Fan-Out Group*.

- *Feature group instance cardinality*, annotated as $\langle l, u\rangle$ at the right-hand side of each group arc, defines the minimum and maximum number of feature instances of any type selectable from the set of all direct sub-feature types. For instance, $\langle 3, *\rangle$ denotes that arbitrary many, but at least three *Channel* instances are required for a *Node*.

- *Cross-tree edges* by means of require- and exclude-edges, annotated with $\langle l, u\rangle$ at both the source and target feature nodes, define constraints on the number of instances among arbitrary pairs of features. For instance, if at least one instance of *Reliable* is selected in some sub-tree clone, then no instance of *Probabilistic Broadcast* is permitted in the *Fan-Out Group* and vice versa. In addition, if between 1 and 5 *Nodes* are selected in a *Fan-Out Group*, then *BT* is used for all *Nodes*, and *WiFi* otherwise.

Figure 4.4 illustrates the configuration semantics of cardinality-based feature models by means of three sample configurations for a simple example. Here, multiple instances of the same feature occurring in a configuration are visualized with their unfolded sub-tree copies.

The combination of the different kinds of cardinality annotations within a cardinality-based feature model may lead to very complicated dependencies among feature selections and corresponding sub-tree clones. In this regard, the notion of *anomaly* is frequently used to summarize undesirable (semantic) properties of feature diagrams [3]. Concerning cardinality-based feature diagrams in particular, one major observation is that all essential anomalies can be explained through (and, therefore, automatically detected as) *dead cardinality anomaly*. Given a cardinality-interval annotation (l, u), cardinality k with $l \leq k \leq u$ is *dead* in a cardinality-based feature model if there exists no valid configuration in which k instances of the corresponding feature(s) occur. Hence, a dead

cardinality may either indicate bounds of intervals being too wide, or gaps within intervals, or false unbounded intervals or even inconsistent models (i.e., every cardinality is dead). Concerning the example in Figure 4.4, we observe the following cases.

- *Dead Cardinality Bounds.* The lower bound 1 of the group instance cardinality interval $\langle 1, * \rangle$ of f_0 is a *dead cardinality*, as at least one instance of both f_1 and f_2 must be selected. Furthermore, the lower bound 1 of group type cardinality $[1, 3]$ of f_0 is also *dead* and the lower bound of the target feature node cardinality interval $\langle 2, 6 \rangle$ of the require-edge from f_4 to f_1 is 6 instead of 2.

- *Cardinality-Interval Gaps.* The group instance cardinality of f_0 contains a *gap* at $(6, 6)$ as no valid combination of feature instances of f_1, f_2, and f_3 with an overall number of 6 is possible. Similarly, the feature instance cardinality interval $\langle 1, 7 \rangle$ of f_1 contains a *gap* at $(2, 5)$.

- *False-Unbounded Intervals.* The group instance cardinality $\langle 0, * \rangle$ of f_0 is *false unbounded* as the maximum number of possible child-feature instances is 11 (thus making any upper-bound cardinality beyond 11 a dead cardinality). In addition, the interval $\langle 1, * \rangle$ on the right-hand side of the exclude-edge between f_1 and f_2 is false unbounded as the upper bound is limited to 2.

In contrast to those false-unbounded cases, feature f_5 is actually *unbounded* thus making the entire model unbounded (i.e., having an infinite number of possible configurations).

To summarize, the proposed cardinality-based feature models constitute a non-trivial extension to FODA-like feature diagrams, yielding potentially infinite configuration spaces and novel kinds of anomalies.

Theoretical Results. The two main theoretical properties for cardinality-based feature model can be summarized as follows.

- **Existence of Normal Form.** For any given cardinality-based feature model, there exists a semantically equivalent normal-form representation without dead features. The normal-form representation preserves the feature-tree structure of the given model, but potentially adapts cardinality-interval annotations by excluding any dead cardinality, where compound intervals are used to represent interval gaps.

- **Decidability of Model Consistency.** For any given cardinality-based feature model, model consistency is decidable, even in case of unbounded models. More generally, for any given cardinality-based feature model, its respective normal-form representation is effectively computable. To this end, a global bound on the configuration space is derivable from the syntactic structure of the model such that any dead cardinality is observable within this bound.

Tool Implementation. Normal-form computation and dead-cardinality detection for cardinality-based feature models is implemented by the tool CardyGAn [32]. Cardinality-bound analysis is based on an encoding of cardinality-based feature model semantics as linear optimization problem using ILP (Integer Linear Programming) thus facilitating the application of the existing ILP solver CPLEX. As ILP requires convex search spaces, it is not applicable to interval-gap analysis which is, therefore, based on a respective SMT encoding for the SMT solver Z3.

Evaluation Results. The experimental evaluation has been conducted on a data set of synthetically generated cardinality-based feature models with up to 5,000 features and different cross-tree constraint ratios and cardinality-interval size distributions. The

results show that bound analysis based on ILP-solving scales remarkable well also to large-scale models, whereas gap analysis based on SMT solving is, in its current state, only applicable to models with at most 200 features.

Chapter 5

Model-based Quality Assurance of Variability over Time

This chapter summarizes two recent works concerned with two dimensions of variability: variability in space in terms of configurable software systems and variability over time in terms of evolution of those systems. In particular, the first work proposes techniques for model-based evolution of problem-space artifacts of software product lines (i.e., feature models), whereas the second work proposes a theoretical foundation for formal reasoning about the behavioral impact of variability both in space and over time within a unified formal framework for model-checking of software product lines.

5.1 Reasoning about Problem-Space Evolution of Configurable Software

Publication. *The contents summarized in this section have been published in Bürdek et al. [11]. A reprint of this publication can be found in Appendix A.*

Classification of Contributions. *CPS: Computation; QA: Test Models; δ: Variability in Space, Variability over Time.*

Summary. The development of configurable software systems such as software product lines usually requires a high upfront investment. As a consequence, they are often supposed to stay in operation for a very long lifetime. Even if domain scoping is conducted very carefully in order to anticipate a wide range of possible diverse customers' needs, it is not possible to foresee all future requirements potentially arising throughout the whole life-cycle in advance. As a result, existing configurable software has to undergo continuous evolution (i.e., variability over time) in order to adapt to ever-changing requirements, new platforms, legal restrictions etc. Concerning software product lines in particular, this process of iteratively interleaving domain- and application-engineering steps is often referred to as *reactive* product-line engineering. As a product-line infrastructure is naturally centered around the feature model (e.g., a FODA-like feature diagram), the proposed approach takes a model-centric view on product-line evolution, by always starting with modifications (edits) to the domain feature model.

As a concrete example, consider the feature diagram of the PPU in Figure 2.7a and the corresponding evolution scenario illustrated in Figure 2.2. As a result of such evo-

lution scenarios, the PPU feature diagram has been modified, accordingly, leading from smaller changes such as adding and removing particular feature nodes and constraints, to arbitrary complex restructurings of large parts to the entire diagram. Coping with software evolution in general and product-line evolution in particular imposes two essential challenges during reactive product-line engineering in practice.

- Modifications are often conducted in ad-hoc manner, without proper documentation of the *syntactic differences* by means of (sequences of) edit operations leading from an old version to a new version of the feature diagram.
- Even local syntactic modifications may lead to global *semantic differences*, being hard to comprehend for the modeler.

To address both challenges, the proposed approach combines two consecutive phases: (1) a syntactic comparison of two consecutive versions of a feature diagram, and (2) semantic change-impact classification resulting from the syntactic differences identified in phase (1).

Concerning (1), the syntactic model-differencing technique applied follows a *state-based* approach in the sense that an old and a new version of a feature diagram are available, but no further information exists about the editing processes between both. In order to derive those editing processes a-posteriori in an automated way, differences between both model versions are split up into atomic edit steps. Those edit steps are supposed to correspond to set of edit operations typically offered by visual editors for feature diagrams. As a consequence, the resulting set of edit operations can be considered *sound* (i.e., every sequence of edit operations only produces syntactically well-formed models) and *complete* (i.e., every well-formed model can be produced from every other well-formed model via at least one sequence of edit operations). Based on this set of atomic edit operations, a catalogue of more complex edit operations is defined, typically being applied to feature diagrams by developers during product-line evolution. This catalogue has been derived from evolution scenarios as observed for the PPU example.

Concerning (2), the semantic impact of (atomic as well as complex) edit operations on feature diagrams may be classified with respect to their effect on the set of valid product configurations (i.e., feature combinations) [34].

- A *generalization* preserves all existing configurations and (potentially) adds new configurations (e.g., by adding a new optional feature or by removing a cross-tree constraint).
- A *specialization* (potentially) removes certain existing configurations and does not add any new configuration (e.g., by removing an optional feature or by adding a cross-tree constraint).
- A *refactoring* does neither add nor remove any configuration (i.e., it transforms a feature diagram into a semantically equivalent diagram).
- Otherwise, the change is called an *arbitrary edit*.

A concrete example of this concept is depicted in Figure 5.1, showing a complex edit operation denoting a feature-diagram refactoring. The pattern-based classification can be very useful in subsequent maintenance steps (e.g., in case of a refactoring, no further evolution of solution-space artifacts is required). However, the modifications actually being classifiable this way (i.e., by solely considering the syntactic differences between two diagrams) are typically limited to edit operations affecting local model fragments

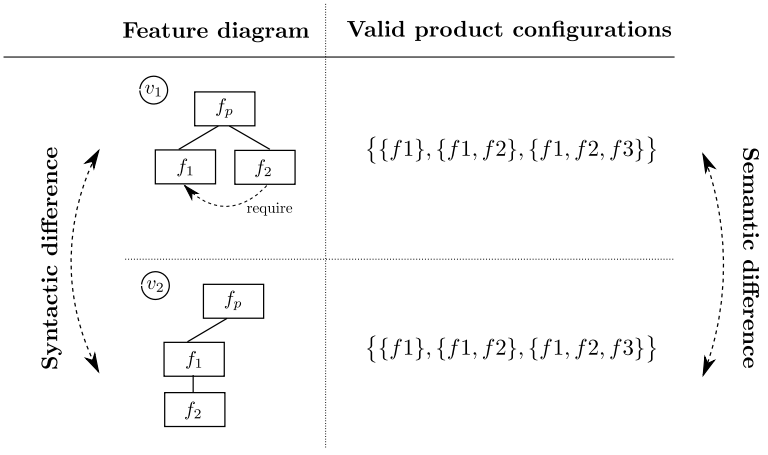


Figure 5.1: Syntactic and Semantic Differences of Feature Diagrams

with a restricted context of application. Hence, in case of larger modifications consisting of sequences of edit operations, a pattern-based classification is, in general, no more possible but rather requires constraint-solver calls as usual [34].

Tool Implementation. A tool implementation of the methodology is based on the model-differencing framework SiLift as well as the model-transformation tool Henshin. The abstract syntax of feature diagrams is defined as EMF-compliant meta model and supports the import of feature diagrams from FeatureIDE.

Evaluation Results. The applicability and feasibility of the approach has been evaluated on the PPU case study as well as artificial data sets. Concerning the computation of syntactic differences, the results show that the technique scales to feature models with at least 500 feature nodes. Concerning the effectiveness of the semantic classification, the technique is capable to classify a high fraction of edit operations observed throughout the evolution history of the PPU case study, including both atomic as well as complex edit operations.

5.2 Reasoning about Solution-Space Evolution of Configurable Software

Publication. *The contents summarized in this section have been published in Lochau et al. [26]. A reprint of this publication can be found in Appendix A.*

Classification of Contributions. *CPS: Computation, Communication; QA: Test Models, Test Goals; δ: Variability in Space, Variability over Time.*

Summary. Various approaches have been proposed in recent literature to specify variability within solution-space modeling and programming languages. To this end, variations of, and extensions to existing core calculi have been defined in order to capture the very essence of behavioral variability in a formal way [5]. However, most of the recent attempts in this area of research exhibit at least one of the following deficiencies.

- Variability-modeling and -reasoning concepts often rely on a so-called 150% representation, by means of an a-priori *superimposition* of all derivable variants into one model. Based on this integrated representation, variability within those models is emulated by adapting existing and/or adding new language constructs, such as

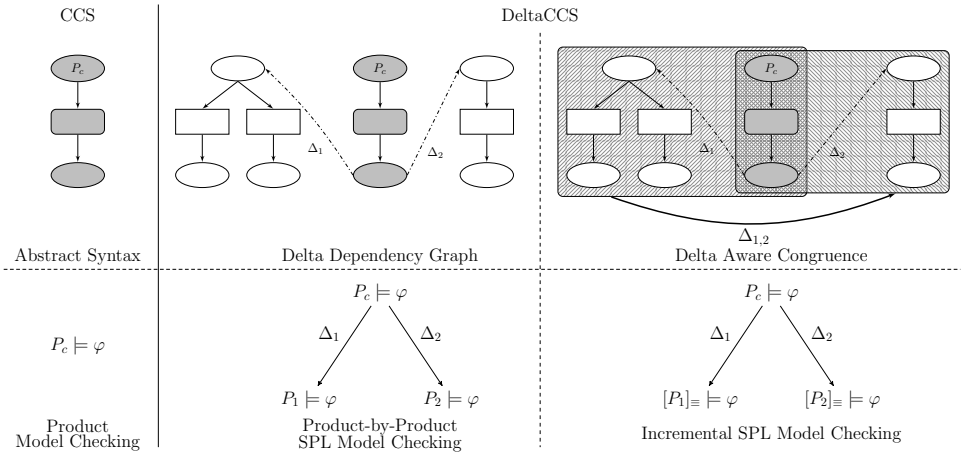


Figure 5.2: Variability-Modeling and -Reasoning Strategies of DeltaCCS

selection/projection of variable parts, (guarded) choice among variable parts, and modal refinement of variable parts. Although many promising variability-aware analysis strategies and tools (e.g., family-based product-line analysis [33]) have been lifted to become applicable to those kinds of superimposed representations, 150% specifications potentially become intractable for realistic applications, due to the additional computational overhead. In addition, almost no support exists so far to properly integrate and handle both variability in space and over time.

- Variability-modeling and -reasoning concepts concentrate on model properties arising from structural/syntactical variability at the syntactical level. In contrast, the *behavioral impact* of variability is often not considered in a systematic way, or is even completely out of scope. In addition, most existing approaches lack an integrated concept and an accompanying theoretical framework for systematically propagating (i.e., re-using) established behavioral properties among (predefined) variants and/or (initially unforeseen) versions in a unified way.

The proposed calculus DeltaCCS is supposed to serve as a formal foundation for, and modular specification and incremental verification of, behavioral (temporal) correctness properties φ in the presence of variability both in space and over time. To this end, DeltaCCS extends the well-known process calculus CCS [28] by a variability mechanism by adopting principles of delta modeling [14]. The key idea of delta modeling in general, and DeltaCCS in particular, is to separate the definition of a *core process* from the definition of (sets of) change directives, so-called *delta* operations. When applied to core process P_c, a delta Δ encapsulates well-defined and arbitrarily fine-grained changes to the core-process definition in a determined way, in order to derive a similar, yet slightly varying process $P' = \Delta(P_c)$. In this regard, process P' may either constitute a predefined *variant* of process P_c thus modeling variability in space, or P' constitutes a new (i.e., initially unforeseen) *version* of P_c in case of variability over time (e.g., as imposed by an unforeseen evolution scenario).

Figure 5.2 provides a conceptual overview on the theoretical framework for reasoning about behavioral variability as provided by DeltaCCS. For specifying behavioral properties φ to be automatically verified (e.g., by a model-checker) on a behavioral process specification P, denoted $P \models \varphi$, DeltaCCS supports a restricted subset of the modal μ-calculus that is sufficient to express fundamental safety- and progress proper-

ties. In order to derive, for instance, two different process variants (or, versions), P_1 and P_2, from a common core process P_c, two (sets of) delta operations, Δ_1 and Δ_2, are applied to the abstract-syntax representation P_c. In particular, a delta application introduces variability by altering the definition of designated sub-processes of P_c in a determined way thus yielding arbitrarily fine-grained behavioral changes. As depicted in Figure 5.2 in the upper part, the dependency-graph representation of CCS terms is augmented with deltas in order to characterize and detect conflicts among multiple applicable deltas at the syntactic level.

Concerning the impact of applications of deltas Δ on the LTS term-rewriting semantics $[P_c]$ of CCS terms P_c, behavioral variability is not emulated by an a-priori determined resolution of variation points as frequently done in 150%-based approaches. Instead, the process semantics $[\Delta(P_c)]$ after applying (sets of) deltas Δ for a particular process variant/version $\Delta(P_c)$ is obtained by altering the rewriting semantics $[P_c]$ of the core process P_c on-the-fly, by overriding the CCS recursion rule. This allows for a precise localization and direct propagation of behavioral variability from the syntactic onto the semantic level, thus becoming integral part of the generative structural operational semantics of CCS.

To formally reason about the behavioral impact of delta applications $\Delta(P_c)$, the calculus further defines a delta-aware congruence notion $[P_c]_{\equiv}$ and a corresponding normal form on CCS process terms in the context of delta applications. Thereupon, the notion of *bisimulation invariance* allows for modular (and maximally local) reasoning about the preservation (i.e., re-use) of properties φ holding for process P_c to also hold after applying Δ, without the need for re-checking φ again on process variant/version $\Delta(P_c)$. Based on this uniform theory of behavioral change due to variability in space and over time, DeltaCCS supports recent strategies [33] for effectively and efficiently analyzing variable software (e.g., model-checking property φ on every derivable variant).

- In a *product-by-product* strategy, property φ is checked on each variant/version $\Delta(P_c)$ in a separate run anew from scratch.

- In a *family-based* strategy, property φ is checked on all variants/versions $\Delta(P_c)$ in one run. To this end, the on-the-fly delta-application mechanism of DeltaCCS permits to derive operational semantics for all possible subsets of deltas into one LTS. Corresponding delta-annotations allow to track variability information during model-checking in order to re-use analysis results among different variants having shared deltas.

- In an *incremental* strategy, constituting a novel combination of both aforementioned strategies, variants/versions are checked in a step-wise manner, by partially re-using model checking results already obtained from previous results. For instance, assume property φ to be checked on process $P_1 = \Delta_1(P_c)$ first, and on process $P_2 = \Delta_2(P_c)$ afterwards. To this end, a so-called *regression delta* $\Delta_{1,2}$ is derivable from Δ_1 and Δ_2 such that $P_2 = \Delta_{1,2}(P_1)$ holds. The regression delta then allows for precisely localizing all differences between P_1 and P_2 at the level of sub-processes and for reasoning about the impact of those differences on property φ, based on the notion delta-aware congruence.

To summarize, DeltaCCS provides a core calculus for a uniform reasoning about semantic impacts of behavioral change due to variability in space and/or over time.

Theoretical Results. The main theoretical corner stones of the DeltaCCS framework can be summarized as follows.

- **Delta-Conflict Resolution.** Given a DeltaCCS specification, a *partial order* on the set of all Deltas is statically derivable from an abstract-syntax tree representation which allows for resolving potential application conflicts among deltas.
- **Correctness of Family-based DeltaCCS Semantics.** Given a process P and a delta Δ, for the family-based DeltaCCS LTS semantics restricted to Δ, denoted $[P_c, \Delta]$, and the LTS semantics $[\Delta(P_c)]$ of a variant/version obtained by Δ, it holds that $[P_c, \Delta] \simeq [\Delta(P_c)]$, where $\simeq$ denotes *bisimulation equivalence*.
- **Correctness of Delta-Aware Congruence.** Given process P and delta Δ, it holds that from $P \equiv \Delta(P)$ it follows that $P \simeq \Delta(P)$ holds.
- **Correctness of Property Preservation.** Given process P and delta Δ such that $P \equiv \Delta(P)$ holds, then from *bisimulation invariance* it follows that $P \models \varphi$ holds iff $\Delta(P) \models \varphi$ holds.
- **Delta Normal Form.** Given process P and delta Δ, $\Delta(P)$ is in *delta normal form* ΔNF if all deltas within set Δ are *maximally distributed* (i.e., variability is localized in a maximally fine-grained manner in sub-process terms of $\delta(P)$).

Tool Implementation. The DeltaCCS model-checker implementation is based on the framework Maude and supports product-by-product, family-based as well as incremental model-checking strategies for properties expressed in (a restricted version) of the modal μ-calculus.

Evaluation Results. The applicability of the approach as well as experiments concerning efficiency trade-offs achievable by the different model-checking strategies supported by DeltaCCS have been conducted on several case studies. The evaluation results indicate potential performance advantages gained by incremental model-checking, as compared to family-based and product-by-product strategies.

Chapter 6

Conclusion

This chapter briefly summarizes the contributions described in this thesis and provides a short outlook on possible future work.

The contributions summarized in this thesis are concerned with different aspects of the specification and quality-assurance of CPS in the presence of variability in space, over time and at runtime. A majority of the presented approaches focus on computation and communication aspects of CPS, by employing *evolving dynamic software product lines* and *model-driven software development* as development paradigms and *model-based testing* as quality-assurance technique.

In order to handle variability in space, a test-generation approach has been presented for efficient test coverage of all variants of a configurable software system. To further improve testing efficiency of the resulting test suites, a subsequent test-suite optimization and test prioritization approach has been defined. Concerning variability in software systems consisting of multiple communicating components, a formal framework based on input/output conformance testing has been presented including in-depth investigations of compositionality and decomposition properties.

For handling variability at runtime, existing concepts from (dynamic) software product line engineering have been adapted by novel modeling and reasoning techniques. In particular, the extensions support constraints on binding times of configuration options as well as reconfiguration behaviors. In addition, an integrated approach has been presented to facilitate multiple feature instances, as being crucial in (self-)adaptive component-based or mobile communication systems.

Finally, to handle variability over time in terms of unforeseen software evolution, techniques have been presented for reasoning about software-artifact changes and their potential impact on crucial system properties To this end, model differencing on problem-space artifacts as well as formal regression analysis on solution-space artifacts of configurable software have been employed.

Most of the research challenges addressed in this thesis are motivated by intensively investigating open problems as observable in case studies from different recent CPS application domains (cf. Chapter 1). In this regard, the evaluation results gained from successfully applying the novel techniques to those case studies reveal promising perspectives for future research based on the results achieved so far.

First, the control aspects and their interplay with the other aspects of CPS have only been partially addressed so far in recent research, concerning appropriate modeling and engineering approaches, effective quality-assurance strategies as well as handling the different dimensions of variability.

Second, entirely novel testing concepts are required to not only handle variability in

space, but also for systematically testing variability at runtime (e.g., model-based testing techniques for adaptive systems behavior) and variability over time (i.e., regression-testing techniques for evolving dynamic software product lines).

Finally, arbitrary interleaving all three dimensions of variability (cf. Figure 2.6) throughout the life-cycle of modern CPS possesses entirely new challenges concerning all three disciplines considered in this thesis (i.e., CPS engineering, quality-assurance by model-based testing and integrated variability management).

Bibliography

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer Science & Business Media, 2013.

[2] H. Baller, S. Lity, M. Lochau, and I. Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *ICST*. IEEE, 2014.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years later: A Literature Review. *Information Systems*, 35, 2010.

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

[5] F. Benduhn, T. Thüm, M. Lochau, T. Leich, and G. Saake. A survey on modeling techniques for formal behavioral verification of software product lines. In *VaMoS*, pages 80–87. ACM, 2015.

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE*, pages 326–335, 2004.

[7] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *ESOP*, LNCS 7792, pages 472–491. Springer, 2013.

[8] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[9] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems – Advanced Lectures*. Springer, 2005.

[10] F. Bujtor, S. Fendrich, G. Lüttgen, and W. Vogler. Nondeterministic Modal Interfaces. Technical report, University of Augsburg, 2014.

[11] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, Udo Kelter, and Andy Schürr. Reasoning about Product-Line Evolution using Complex Feature Model Differences. *Automated Software Engineering – Special Issue on Long Term Evolution of Software Systems*, 2015.

[12] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *FASE*, pages 84–99, 2015.

[13] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Corts, and M. Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014.

[14] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. *SIGPLAN Notices*, 46(2):13–22, 2010.

[15] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[16] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*, LNCS 3676, pages 422–437. Springer, 2005.

[17] R. de Nicola. Extensional Equivalences for Transition Systems. *Acta Inf.*, 24(2):211–.237, 1987.

[18] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1):83–133, 1984.

[19] B. H. C. Cheng et al. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer, 2009.

[20] C. Gregorio-Rodrguez, L. Llana, and R. Martnez-Torres. Input-Output Conformance Simulation (iocos) for Model Based Testing. In *FORTE*, LNCS 7892, pages 114–129. Springer, 2013.

[21] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2:270–285, 1993.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[23] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, pages 805–824, 2011.

[24] M. Lochau, J. Bürdek, S. Hölzle, and A. Schürr. Specification and automated validation of staged reconfiguration processes for dynamic software product lines. *Software and System Modeling*, 16(1):125–152, 2017.

[25] M. Lochau, J. Bürdek, S. Lity, M. Hagner, C. Legat, U. Goltz, and A. Schürr. Applying Model-based Software Product Line Testing Approaches to the Automation Engineering Domain. *Automatisierungstechnik*, 62(11):771–780, 2014.

[26] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. Incremental Model Checking of Delta-Oriented Software Product Lines. *Journal of Logical and Algebraic Methods in Programming*, 2015.

[27] L. Luthmann, S. Mennicke, and M. Lochau. Compositionality, Decompositionality and Refinement in Input/Output Conformance Testing. In *FACS*, pages 174–191, 2016.

[28] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[29] G. J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[30] D. L. Parnas. Software aging. In *ICSE*, pages 279–287. IEEE Computer Society Press, 1994.

[31] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE*, pages 347–350. IEEE Computer Society, 2008.

[32] T. Schnabel, M. Weckesser, R. Kluge, M. Lochau, and A. Schürr. Cardygan: Tool support for cardinality-based feature models. In *VaMoS*, pages 33–40. ACM, 2016.

[33] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.

[34] T. Thüm, D. Batory, and C. Kästner. Reasoning About Edits to Feature Models. In *ICSE*, pages 254–264. IEEE Computer Society, 2009.

[35] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence, 1996.

[36] University of Berkeley. Cyper-Physical Systems, last visited: June 2017. `http://cyberphysicalsystems.org`.

[37] M. Utting and B. Legeard. *Practical Model-Based Testing. A Tools Approach*. M. Kaufmann, 2007.

[38] M. Veanes and N. Bjorner. Alternating simulation and IOCO. *STTT*, 14(4):387–405, 2012.

[39] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit. Technical Report TUM-AIS-TR-01-14-02, TU München, 2014.

[40] M. Weckesser, M. Lochau, T. Schnabel, B. Richerzhagen, and A. Schürr. Mind the gap! automated anomaly detection for potentially unbounded cardinality-based feature models. In *FASE*, pages 158–175, 2016.

Appendix A

List of Selected Publications

This Appendix contains the selected publications described in this thesis, listed in chronological order of appearance. The following overview summarizes the individual contributions of the author of this thesis to the different publications.

[2] H. Baller, S. Lity, M. Lochau, and I. Schaefer: *Multi-Objective Test Suite Optimization for Incremental Product Family Testing.* In Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST), pages 303–312, IEEE, 2014.

Contributions: Conceptualization, Writing: Introduction/Conclusion (100%), Theory (75%), Evaluation (25%), Related Work (75%), Reviewing and Supervision.

[25] M. Lochau, J. Bürdek, S. Lity, M. Hagner, C. Legat, U. Goltz, and A. Schürr: *Applying Model-based Software Product Line Testing Approaches to the Automation Engineering Domain.* In Automatisierungstechnik, 62(11), Vol. 62(11), pages 771–780, 2014.

Contributions: Conceptualization (Original Draft and Revision), Writing: Introduction/Conclusion (100%), Concept Presentation (100%), Evaluation (25%), Related Work (25%), Supervision of Prototype Implementation, Design of Case Study and Experiments.

[12] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, D. Beyer: *Facilitating Reuse in Multi-Goal Test-Suite Generation for Software Product Lines.* In Proceedings of Fundamental Approaches to Software Engineering (FASE), pages 84–99, Springer, 2015.

Contributions: Conceptualization, Writing: Introduction/Conclusion (100%), Theory (75%), Evaluation (50%), Related Work (75%), Supervision of Prototype Implementation.

[27] L. Luthmann, S. Mennicke, M. Lochau: *Compositionality, Decompositionality and Refinement in Input/Output Conformance Testing.* In Proceedings of the 13th International Conference of Formal Aspects of Component Software (FACS), pages 174–191, Springer, 2016.

Remark: The attached extended version additionally contains all proofs.

Contributions: Theory and Formalization, Writing: Introduction/Conclusion (100%), Theory and Proofs (50%), Related Work (75%), Reviewing and Supervision.

[40] M. Weckesser, M. Lochau, T. Schnabel, B. Richerzhagen, A. Schürr: *Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-Based Feature Models.* In Proceedings of Fundamental Approaches to Software Engineering (FASE), pages 158–175, Springer, 2016.

Contributions: Theory and Conceptualization, Writing: Introduction/Conclusion (100%), Concept Presentation (75%), Evaluation (25%), Related Work (75%), Supervision of Prototype Implementation.

[11] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr: *Reasoning about Product-Line Evolution using Complex Feature Model Differences.* In Automated Software Engineering Journal, Special Issue on Long Term Evolution of Software Systems, Vol. 23(4), pages 687–733, Springer, 2016.

Contributions: Theory and Conceptualization (Original Draft and Revision), Writing: Introduction/Conclusion (25%), Concept Presentation (25%), Evaluation (25%), Related Work (25%).

[26] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck: *Incremental Model Checking of Delta-Oriented Software Product Lines.* In Journal of Logical and Algebraic Methods in Programming, Vol. 85(1), pages 245–567, Elsevier, 2016.

Contributions: Theory and Formalization (Original Draft and Revision), Writing: Introduction/Conclusion (100%), Theory and Proofs (75%), Evaluation (25%), Related Work (25%), Supervision of Prototype Implementation and Case Study Design.

[24] M. Lochau, J. Bürdek, S. Hölzle, A. Schürr: *Specification and Automated Validation of Staged Configuration Processes for Dynamic Software Product Lines.* In Journal of Software & Systems Modeling, Vol. 16(1), pages 125–152, Springer, 2017.

Contributions: Theory and Formalization (Original Draft and Revision), Writing: Introduction/Conclusion (100%), Concept Presentation (100%), Evaluation (25%), Related Work (75%), Supervision of Prototype Implementation and Case Study Design.