Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Delta-oriented model-based integration testing of large-scale systems☆

Malte Lochau[a], Sascha Lity[b,*], Remo Lachmann[c], Ina Schaefer[c], Ursula Goltz[b]

[a] TU Darmstadt, Real-Time Systems Lab, Germany
[b] TU Braunschweig, Institute for Programming and Reactive Systems, Germany
[c] TU Braunschweig, Institute of Software Engineering and Automotive Informatics, Germany

ARTICLE INFO

Article history:
Received 19 November 2012
Received in revised form 4 November 2013
Accepted 17 November 2013
Available online 23 November 2013

Keywords:
Large-scale systems
Model-based testing
Regression testing
Variable software architectures

ABSTRACT

Software architecture specifications are of growing importance for coping with the complexity of large-scale systems. They provide an abstract view on the high-level structural system entities together with their explicit dependencies and build the basis for ensuring behavioral conformance of component implementations and interactions, e.g., using model-based integration testing. The increasing inherent diversity of such large-scale variant-rich systems further complicates quality assurance. In this article, we present a combination of architecture-driven model-based testing principles and regression-inspired testing strategies for efficient, yet comprehensive variability-aware conformance testing of variant-rich systems. We propose an integrated delta-oriented architectural test modeling and testing approach for component as well as integration testing that allows the generation and reuse of test artifacts among different system variants. Furthermore, an automated derivation of retesting obligations based on accurate delta-oriented architectural change impact analysis is provided. Based on a formal conceptual framework that guarantees stable test coverage for every system variant, we present a sample implementation of our approach and an evaluation of the validity and efficiency by means of a case study from the automotive domain.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

Modern software systems are increasingly large-scaled. They often exist in many different variants in order to adapt to their environment context. Additionally, they evolve over time into different versions in order to satisfy changing user requirements. This diversity in space and time causes high complexity during system design and implementation as well as for quality assurance. Software architecture (SA) specifications are of growing importance for coping with these variant-rich large-scale software systems. SAs provide an abstract view on the high-level structural system entities together with their explicit interdependencies which alleviates complexity during system development. This decreases development complexity by allowing a hierarchical decomposition of the overall system into smaller subsystems and their components. For quality assurance, an architectural design builds the basis for ensuring behavioral conformance of component implementations and interactions w.r.t. architectural specifications, e.g., using model-based component and integration testing.

The adoption of SA testing approaches to variant-rich software systems faces the challenge to ensure correctness of component implementations and interactions for any possible system variant which is, in general, infeasible due to their high number. Even if every system variant could be tested in isolation, this is very inefficient because of the commonality between the variants that is checked over and over again.

To counter this problem, we combine architecture-driven model-based testing principles and regression-based testing strategies for efficient, yet comprehensive conformance testing on the component and on the integration test level. We model the expected behavior of a system by its architectural specification in terms of components and connectors. The intra-component behavior is captured by state machines. However, applying the corresponding model-based testing techniques also to component integration testing on the basis of a parallel composition of those component state machines is, in general, impracticable for large-scale systems due to the well-known state-explosion-problem. Instead, component interaction scenarios to be tested are explicitly specified by message sequence charts as proposed, e.g., in Briand et al. (2012). In order to express system diversity, we apply the principles of delta modeling to the system description on the

☆ This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

* Corresponding author. Tel.: +49 531 391 3276.
E-mail address: lity@ips.cs.tu-bs.de (S. Lity).

architectural as well as on the component level. Delta modeling (Clarke et al., 2010; Schaefer et al., 2010, 2011) is a modular, yet flexible way to capture variability in time and in space by explicitly specifying the changes between system variants. In this way, we obtain a concise description of the commonality and variability of the expected behavior between system variants. Thereupon, we are able to automatically derive a *regression delta* explicitly capturing differences among arbitrary system variants/versions. In contrast to classical regression testing requiring fine-grained, therefore, expensive tracings of changes (Engström et al., 2008), e.g., by means of model differencing (Treude et al., 2007; Kelter and Schmidt, 2008), the regression delta allows us to efficiently reason about preplanned test artifact reuse among system versions/variants at all testing levels in a sound and uniform way. We further use this delta-oriented representation to reduce retesting common behaviors of system variants by relying on the principles of regression testing. The derivation of the regression testing obligations between two system variants is based on an impact analysis of the architectural and behavioral changes specified in the respective regression delta.

The contribution of this work is twofold: (1) we introduce delta-oriented architecture test modeling to achieve the systematic reuse of common test artifacts between different system variants and/or versions and (2) we apply delta-oriented regression planning for the systematic evolution of variable test artifacts among different software variants and/or versions. This article extends our previous work on delta-oriented component testing (Lochau et al., 2012) by leveraging the idea of delta-oriented test modeling and test artifact evolution to the integration test level and combines delta-oriented component and scalable integration testing of variant-rich software systems into one unified framework.

We illustrate and evaluate our approach in a prototypical implementation by means of a case study from the automotive domain, a simplified *Body Comfort System* (BCS). Its original version was proposed by Müller et al. (2009) in cooperation with an industrial partner and was enhanced to a variant-rich system by Oster et al. (2011). The BCS comprises a number of standard and optional functions. All BCS include by default a *Human Machine Interface* (HMI) as point of interaction with a driver, an electric *Exterior Mirror* (EM) with optional heating functionality, either an *Automatic Power Window* (AutoPW) or a *Manual Power Window* (ManPW), and a *Finger Protection* (FP) blocking the window movement when a finger is clamped in a window. Optionally, a BCS can consist of a *Remote Control Key* (RCK) enabling the locking/unlocking of the car as well as the controlling of the window movement, an *Alarm System* (AS) with optional *Interior Monitoring* (IM), a *Central Locking System* (CLS) with optional *Automatic Locking* (AL) when the car is driving, and for some functions like the CLS *Status LEDs* (LED) indicating the activation/inactivation of the corresponding function. The planned combination of those functional variants results in 11,616 possible variants of the BCS.

This article is structured as follows: In Section 2, we present the foundations for delta-oriented model-based testing. In Section 3, we describe our integrated framework for model-based component and integration testing. In Section 4, we apply the principles of delta modeling to obtain variable test models both on the component and the integration level. In Section 5, we show how to evolve the test artifacts between system variants and/or versions at all test levels based on the derivation of regression deltas and, thereupon, we describe an incremental testing work flow incorporating a systematic evolution of test artifacts. In Section 6, we present our prototypical tool chain. In Section 7, we describe the case study design and, thereupon, we discuss the results of our evaluation in Section 8. We compare our approach to related work in Section 9 and conclude in Section 10.

2. Fundamentals of delta-oriented model-based testing

In this section, we describe the general concepts of our incremental model-based testing approach for variant-rich large-scale (software) systems based on delta-oriented test model specifications. This generalized framework is later instantiated to guarantee efficient, yet reliable test coverage at both component as well as integration level.

2.1. Foundations of model-based testing

Model-based testing aims at the automation of black-box testing processes (Utting and Legeard, 2007). The fundamental test artifacts involved in those model-based testing processes for a single software system under test together with their interrelations are depicted in Fig. 1(a). A *test model tm* specifies the intended behaviors of the software implementation under test *sut*. The (partial) verification of the behavioral conformance of the *sut* to the test model *tm* is performed by selecting a finite set of *test cases*, i.e., representative executions of the software system extracted from the test model (De Nicola, 1987). The *sut* passes the experimental execution of a test case *tc* if it reacts as intended, i.e., as specified in the corresponding test model *tm*. Functional test cases usually define a sequence of controllable input stimuli to be injected into the software system under test, together with a corresponding sequence of system outputs stating the reactions expected for those inputs.

A test case *tc* is *valid* for a test model *tm* if it conforms to a behavior as specified by the test model. By $TC(tm)$ we refer to the set of all valid test cases defined by the test model *tm*. In general, this set contains an infinite number of test cases and the corresponding execution sequences are of potentially infinite length. The selection of a finite set of *representative* test cases of finite length into a *test suite* $TS \subseteq TC(tm)$ is usually based on *adequacy criteria* measuring the quality of test suites (Utting and Legeard, 2007). For instance, a model-based *coverage criterion CC* applied to a test model *tm* selects a finite subset $CC(tm) = TG \subseteq TG(tm)$ of *test goals*, i.e., particular structural elements occurring in the test model. By $TG(tm)$ we refer to the set of all possible test goals within test model *tm*.

For a test suite *TS* to satisfy a coverage criterion, each selected test goal $tg \in TG$ is to be *covered*, i.e., traversed by at least one test case $tc \in TS$. Correspondingly, we have an *n*-to-*m* correspondence

$$covers_{CC} \subseteq TC(tm) \times TG(tm)$$

between test cases $tc \in TS \subseteq TC(tm)$ and the set of test goals selected for a criterion *CC*. Please note that we may omit the index *CC* in the following if not relevant. These abstract notions for model-based testing are independent of the actual representation of test models, test cases, test goals, etc. Their concrete instantiation depends on the modeling formalism, the system level, and the development phase to be addressed by the corresponding testing campaigns (Utting and Legeard, 2007). As illustrated in Fig. 2, different kinds of test models and corresponding test artifacts are usually applied depending on the testing stage and the software units under consideration. For our architecture-based approach to model-based testing of single components, as well as their integration at the architectural level, we consider the following, mutually interrelated models:

- *Architecture model*. The decomposition of the overall software system into components and the specification of communication relationships between those components are specified in a high-level structural system description based on component-connector abstractions. We use those architecture models as a basis for planning component and integration testing of large-scale systems.

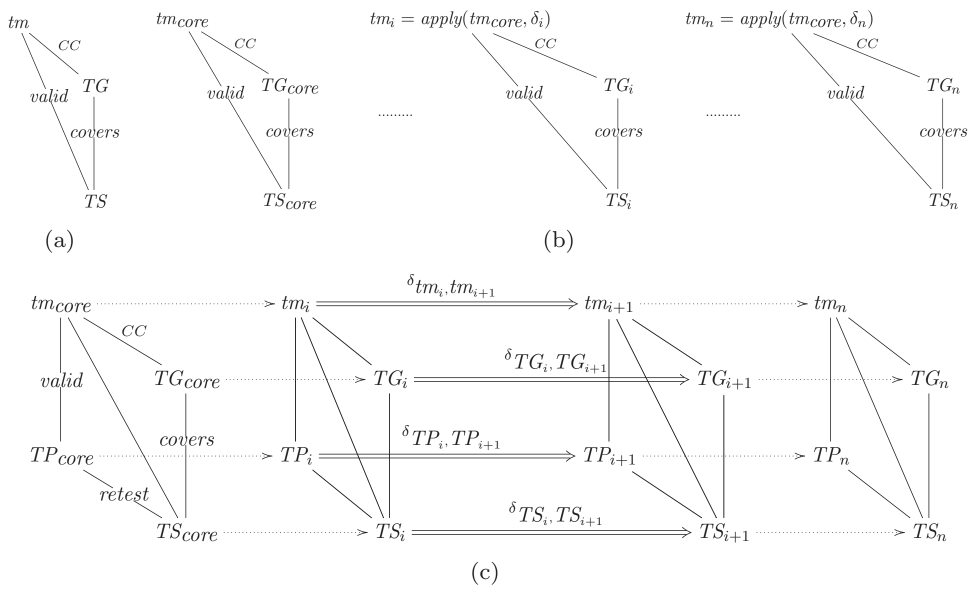


Fig. 1. Model-based testing artifacts for single implementations under test and their incremental evolution for collections of implementation variants under test.

- *State machine test model*. For intra-component behaviors, we use state machines as component test models to specify valid control flows in terms of state-based action/reaction sequences observable at component interfaces.
- *Message sequence chart test models*. For inter-component behaviors, we use message sequence charts as integration test models to specify representative valid interactions in terms of scenario-based signal sequences observable between component interfaces as a result of their parallel composition.

2.2. Delta-oriented test modeling

Until now, we have assumed that the software under test *sut* is a single system for which the corresponding test model *tm* and further test artifacts have been specifically designed from scratch. However, in many application domains software systems with inherent diversity exist as a result of assembling similar, yet well-distinguished system variants/versions from a common core platform. The diversity arises, e.g., within families of similar systems organized in a software product line (Pohl et al., 2005; Clements and Northrop, 2001) or due to evolutionary adaptions of existing software systems to subsequent system versions over time.

As illustrated in Fig. 1(b), the application of model-based testing to a family of software system variants (products) p_i, $1 \leq i \leq n$, requires a collection of corresponding system-specific test artifacts as described in Section 2.1. Designing model-based test artifacts for each system variant in isolation is inefficient because of the numerous commonalities and, therefore, lots of redundancies, and it is often even infeasible due to the high number of possible variants. Hence, a mechanism is required to explicitly specify the commonality and variability among the different members of those sets of similar systems in order to derive system variant-specific test artifacts in a preplanned and automated way.

The delta-oriented approach for engineering variant-rich software systems provides such a mechanism (Clarke et al., 2010; Schaefer, 2010). In the delta modeling approach, the possible variants of a system family are represented by a core system p_{core} and a set of deltas δ specifying predefined changes to that core. The core system can be any valid system of the system family. Choosing an appropriate core system may be guided by structural criteria, e.g., requiring the selection of a smallest or a largest possible variant, respectively, by means of the amount and/or criticality of its implemented features. In practice, the selection of the core system is often determined by project-specific considerations such as the existence of a legacy system to be used as a basis for the system family, as well as by business factors, e.g., focusing on the most-selling

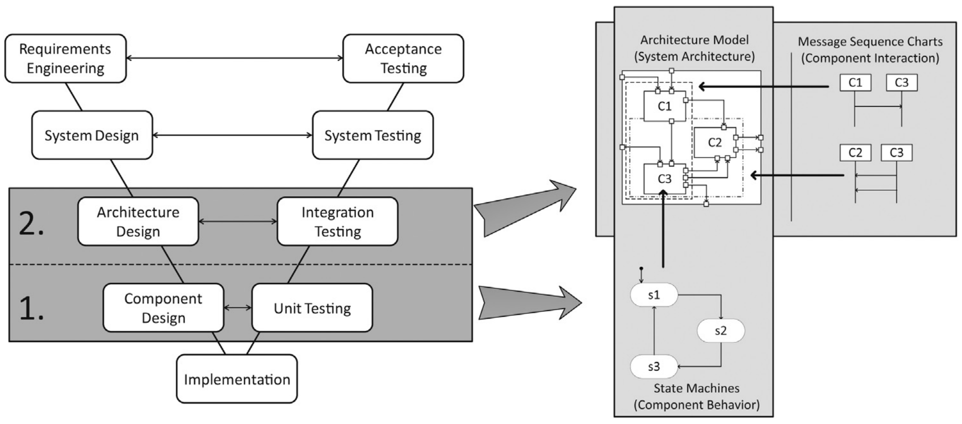


Fig. 2. Test stages and corresponding test models.

system. Depending on the concrete set of metrics and constraints under consideration, the selection of an appropriate core system imposes a constraint multi-objective optimization problem.

Transformation operations $op \in \delta$ defined in a delta δ comprise *additions*, *removals* and *modifications* of elements of the core system that can be combined in arbitrary ways. In contrast to model differencing (Treude et al., 2007; Kelter and Schmidt, 2008), where the changes are derived from the previous and the current system variant, in delta modeling the deltas encapsulate changes to be specified explicitly. Hence, each system variant p_i is generated by applying a corresponding set of deltas to the core. Here, we assume a direct mapping between a system variant and its corresponding deltas. For instance, test model variant tm_i is obtained from the core test model tm_{core} by applying the delta δ_i, which is denoted by $tm_i = apply(tm_{core}, \delta_i)$ in the following as indicated in Fig. 1b.

For integrating component and integration testing in delta-oriented software engineering, we adapt the principles of delta modeling for our architecture-based integration test model (cf. Section 4.3) and component test model specifications (cf. Section 4.2).

2.3. Incremental model-based testing of delta-oriented systems

For model-based testing of a family of systems, corresponding test artifacts are required for every system variant under test. As shown in Fig. 1(b), the set of test goals TG_i and the test suite TS_i of a system variant p_i are derived from a test model variant tm_i as usual. However, also for those sets of test artifacts, reuse potentials arise among the different system variants under test. As illustrated in Fig. 1(c), when stepping from one system under test p_i to a subsequent variant p_{i+1} under test, the corresponding test artifacts are incrementally evolved based on the principles of regression testing.

Originally, the purpose of regression testing is to efficiently verify that changes between different *versions* of a system are as intended (Agrawal et al., 1993; Engström et al., 2008). In the context of variant-rich large-scale systems, we also have to guarantee that the differences between system *variants* comply with the differences of the variant specifications. Hence, applying regression testing strategies to variant-rich systems aim at verifying that for two system variant implementations p_i and p_{i+1}

(1) the differences between p_i and p_{i+1} are correctly implemented and
(2) those differences do not incorrectly influence parts of p_i reused in p_{i+1} other than intended.

Accordingly, stepping from variant p_i to the next variant p_{i+1}, the test suite TS_i is adapted to TS_{i+1} to be valid for p_{i+1} and sufficiently covering the test goals in tm_{i+1}. Reuse potentials between p_i and p_{i+1} arise for

(1) *test cases* in TS_i obtained from tm_i that are also valid for TS_{i+1} thus reducing test case generation efforts and
(2) *test execution results* of test cases in TS_i executed on p_i and being reusable for p_{i+1} thus reducing test execution efforts.

For a test case $tc \in TS_i$ to be reusable in TS_{i+1}, we require that tc corresponds to system behavior commonly specified in tm_i and tm_{i+1}, i.e., addressing behaviors similar to p_i and p_{i+1} such that

$$exec(tm_i, tc) \approx_{te} exec(tm_{i+1}, tc)$$

holds. We assume the $exec(tm, tc)$ primitive to produce the execution trace by means of the sequence of visible actions resulting from the application of test case tc to test model tm.

By $\approx_{te}$ we refer to the testing equivalence te under consideration, e.g., execution trace equivalence (De Nicola, 1987).

For the reuse of test execution results obtained from $exec(p_i, tc)$ for reusable test cases $tc \in TS_{i+1}$, we further require

$$exec(tm_i, tc) \approx_{te} exec(tm_{i+1}, tc) \Rightarrow exec(p_i, tc) \approx_{te} exec(p_{i+1}, tc).$$

The reuse of test results refers to the well-known retest selection problem, regression testing techniques are faced with (Agrawal et al. (1993)):

> Select from the set TS_R of reusable test cases a minimum retest subset $TS_{RT} \subseteq TS_R$ such that TS_{RT} is capable to cover all potentially erroneous impacts of changes between the system implementations p_i and p_{i+1} such that
>
> $$exec(p_i, TS_{RT}) \approx_{te} exec(p_{i+1}, TS_{RT}) \Rightarrow exec(p_i, TS_R) \approx_{te} exec(p_{i+1}, TS_R).$$

Sets of test cases are partitioned into subsets of *reusable* $TS_R \subseteq TS_i$, *obsolete* $TS_O = TS_i \setminus TS_R$, and *new* test cases $TS_N = TS_{i+1} \setminus TS_R$. In contrast to traditional regression testing, we assume that obsolete test cases may be reusable again for another system variant later in the testing process. Therefore, we do not discard them but rather keep them in the overall repository of test artifacts. The set of test cases to be (re-)executed on p_{i+1} contains the *retest* set $TS_{RT} \subseteq TS_R$, and all new test cases in TS_N.

We propose the concept of deriving *regression deltas* $\delta_{tm_i, tm_{i+1}}$ explicitly capturing the differences between arbitrary test model variants tm_i and tm_{i+1} such that

$$tm_{i+1} = apply(tm_i, \delta_{tm_i, tm_{i+1}})$$

holds (cf. Fig. 1(c)). Corresponding regression deltas are systematically derivable also for the other kinds of test artifacts, namely test goal regression deltas $\delta_{TG_i, TG_{i+1}}$ and test suite regression deltas $\delta_{TS_i, TS_{i+1}}$. We further define for every system p_i a *test plan* $TP_i = TS_{RT} \cup TS_N \subseteq TS_i$ comprising the set of test cases to be (re-)tested on p_i. Accordingly, a test plan regression delta $\delta_{TP_i, TP_{i+1}}$ is derivable (cf. Fig. 1(c)) for the purposes of test artifact evolution.

We use these fundamental concepts and notions for the definition of a delta-oriented architecture-based regression testing which is split up in the *delta-oriented component and integration test modeling* and in the *delta-oriented component and integration test artifact evolution*.

3. Architecture-based component and integration testing

In this section, we present our approach to architecture-based component and integration testing. Its ingredients are depicted in Fig. 3. The central element is the system architecture in form of a component-connector model where the behaviors within components are specified by means of state machine models and a collection of message sequence charts specify sample interactions between components.

3.1. Architecture model

The architecture of a software system defines an abstract overview of its high-level system structure (Szyperski, 2002; Medvidovic and Taylor, 2000). In its most general form, an architecture model consists of a finite set $C = \{c_1, c_2, \ldots, c_l\}$ of *components* constituting the main entities of computation, e.g., implementing particular control/regulation tasks, a finite set $Con = \{con_1, con_2, \ldots, con_m\}$ of *connectors* denoting interaction dependencies between components, and a finite set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ of global *signals* for the communication between interacting components and/or the system environment via connectors.

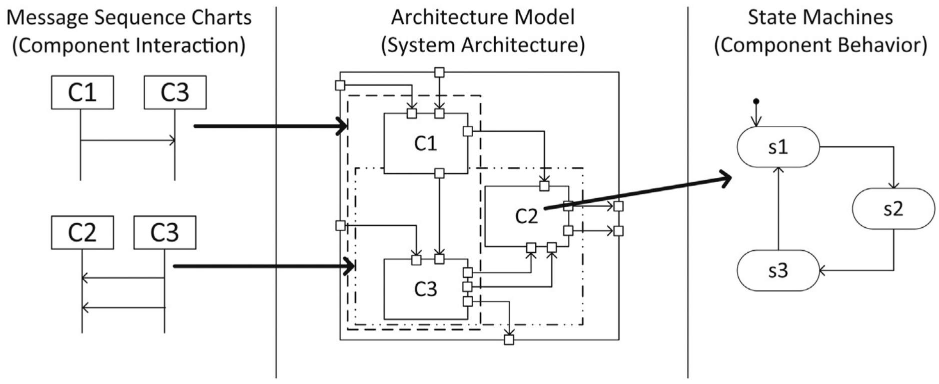


Fig. 3. Relationship between the test models organized in an architecture test model.

Example 3.1. A sample architecture model for the BCS case study is shown in Fig. 4. The component *Automatic Power Window* (AutoPW) controls the automated movement of a window, *Finger Protection* (FP) blocks the window movement whenever an obstacle is detected, *Human Machine Interface* (HMI) implements all interactions with the driver, and *Central Locking System* (CLS) controls the locking/unlocking of the car.

Most architecture modeling languages provide further constructs for defining hierarchical nesting of components, but we limit our considerations to flat architectural models to illustrate the key ideas of our testing approach. However, our concepts are adoptable to hierarchical components, etc., e.g., by introducing a preprocessing step performing a flattening prior to apply our approach. We also leave out further advanced constructs such as complex data types for signals and complex interface definitions, which are out of scope of this article. However, our architecture model can be extended to also incorporate those constructs, accordingly. Thereupon, the architectural delta language and the respective regression-based retesting analyses are to be enhanced, accordingly, to also cope with more advanced architectural modeling concepts. For instance, the delta-oriented architecture modeling approach presented in Haber et al. (2011, 2011) supports deltas on hierarchical components by first selecting the surrounding components and then selecting the nested elements to be changed by the delta. Thereupon, change impact analyses techniques, e.g., architectural slicing are to be adapted to also take sub components into account during dependency analysis.

Definition 3.1 *(Architecture model).* An *architecture model* is a triple (C, Π, Con), where C is a finite, non-empty set of *components*, Π is a finite, non-empty set of *signals* and $Con \subseteq C \times \Pi \times \Pi \times C$ is a set of *connectors*.

We assume signals $\pi \in \Pi$ to be typed over an appropriate value domain such as Boolean, integers, etc. By convention, we at least require the set C to contain a special component $\mathcal{E}$ which represents the *environment* of the system specified by an architecture model. Correspondingly, we assume a special element $\epsilon \in \Pi$ to represent signals exchanged with the environment $\mathcal{E}$. A connector $(c, \pi_o, \pi_i, c') \in Con$ enables a source component $c \in C$ to emit an output signal $\pi_o \in \Pi$ to a target component $c' \in C$ by matching π_o with a *compatible* input signal $\pi_i \in \Pi$.

Depending on the source and target components involved, we consider three different *configurations* for a connector $con \in Con$, i.e., $con = (c, \pi_o, \pi_i, c')$ with $c, c' \in C \setminus \{\mathcal{E}\}$ and $\pi_o, \pi_i \in \Pi \setminus \{\epsilon\}$ is an *internal connector*, $con = (\mathcal{E}, \epsilon, \pi_i, c')$ with $c' \in C$ is an *input connector*, and $con = (c, \pi_o, \epsilon, \mathcal{E})$ with $c \in C$ is an *output connector*. Internal connectors define interactions/communications between internal components, whereas input and output connectors denote interactions of a component with the external environment. Input connectors support communication scenarios, where input signals are delivered from the environment such as sensor values and output connectors denote system interfaces that allow for emitting signals that influence the environment, e.g., by controlling an actuator.

We further require for each connector $con = (c, \pi_o, \pi_i, c')$ that $c \neq c'$ holds, i.e., the source component has to be different from the target component thus avoiding self-loops in the component-connector graph. We do not impose any further requirements concerning connectedness properties of the resulting component-connector graph.

Example 3.2. The set *Con* of the sample architecture model in Fig. 4 contains, e.g., the input connector $(\mathcal{E}, \epsilon, pw\_but\_mv\_dn, HMI)$, the output connector $(AutoPW, pw\_auto\_mv\_up, \epsilon, \mathcal{E})$ and the internal connector $(FP, fp\_on, fp\_on, AutoPW)$.

By $AM(\Pi)$ we refer to the set of all architecture models defined over a set Π of signals. Besides components and connectors as first class entities, recent architecture modeling languages often provide further concepts, such as *port* and *interface* definitions, for component specifications (Medvidovic and Taylor, 2000). Ports define explicit interaction points of components usually comprising the signal specification including the type and direction for communicating with compatible ports of other components via a corresponding connector. In our foundational model for model-based architecture integration testing, we do not consider explicit specifications of component ports and interfaces as those are derivable from the component-connector configurations. For instance, a signal $\pi_i \in \Pi$ corresponds to an *input port* of a component $c \in C$ if there exist a connector $(c', \pi_o, \pi_i, c) \in Con$, whereas a signal $\pi_o \in \Pi$ corresponds to an *output port* if there exist a connector $(c, \pi_o, \pi_i, c') \in Con$. The set $\Pi_{i,c} \subseteq \Pi$ of all input ports and the set of all output ports $\Pi_{o,c} \subseteq \Pi$ of a component then defines the *interface* of the component.

Based on the definition of the architecture model, we define corresponding component test models and integration test models as an integral part of our architecture test model.

3.2. State machine test model for component testing

Component testing aims at verifying whether a component implementation satisfies a given behavioral specification given as a *component test model*. When applying model-based (black-box) testing, the behavior that is visible at the component interface is to be tested by means of experimental injections of sample sequences of controllable input signals and observing corresponding component reactions, i.e., output signal occurrences. Input signal injections cause input event occurrences in a component by means of discrete behavioral stimuli generated at the component interface. Output signals refer to output events released,

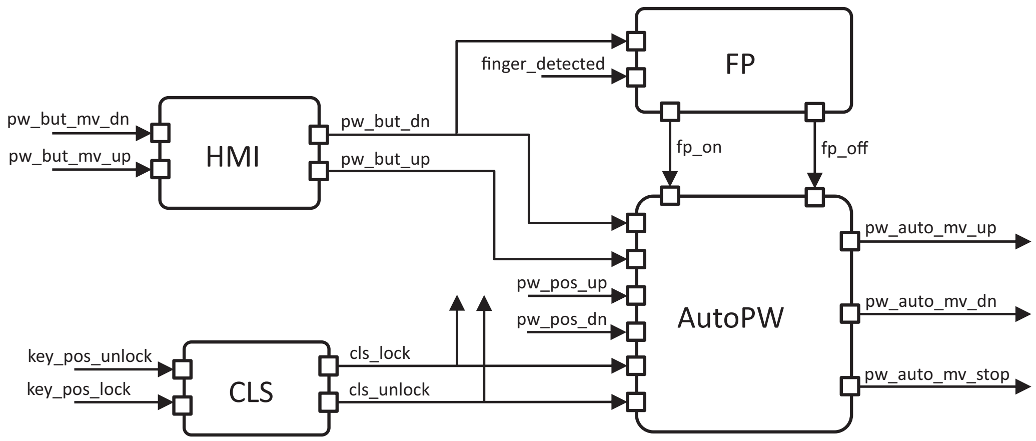


Fig. 4. Sample BCS architecture model.

e.g., for interactions with other components. State-machine-like modeling approaches are commonly used for modeling those intra-component behaviors (Utting and Legeard, 2007). State machines define event-driven input/output relations for component interfaces based on labeled state-transition graphs. Here, we assume flat state-transition graphs with transitions labeled over distinct sets of input events and output events.

Definition 3.2 *(State machine test model). A state machine test model* is a triple $sm = (S, s_0, T)$, where S is a finite set of *states*, $s_0 \in S$ is the *initial state* and $T \subseteq S \times \Pi \times S$ is a *transition relation* labeled over a set $\Pi = \Pi_{i,sm} \cup \Pi_{o,sm}$ of events partitioned into disjoint sets of *input events* $\Pi_{i,sm}$ and *output events* $\Pi_{o,sm}$.

We write $(s, \pi_o\,!\,, s') \in T$ to denote transitions labeled with output events $\pi_o \in \Pi_{o,sm}$ and $(s, \pi_i\,?\,, s') \in T$ to denote transitions labeled with input events $\pi_i \in \Pi_{i,sm}$. We abstract from transition labels representing internal actions, usually denoted as τ-steps. By $SM(\Pi)$ we refer to the set of all state machine test models defined over a set Π of events, where we require for each state machine $sm \in SM(\Pi)$ well-formedness criteria as usual, namely the state-transition graph have to be connected and every state has to be reachable.

State machine test models specify the potential behaviors of a component by means of paths through the state-transition graph. Thus, a (finite) transition sequence $(t_0, t_1, \ldots, t_{k-1}) \in T^*$ is a *valid test case specification* for a state machine test model $(S, s_0, T) \in SM(\Pi)$ if it corresponds to a *path*

$$s_0, t_0, s_1, t_1, \ldots, s_{k-1}, t_{k-1}, s_k$$

in the state-transition graph starting in the initial state s_0. By $TC(sm) \subseteq T^*$ we refer to the set of all *valid* test cases of a state machine test model sm, ranged over by tc, tc'. A *test run* for a test case $tc = (t_0, t_1, \ldots, t_{k-1}) \in TC(sm)$ is an execution *trace*

$$exec(sm, tc) = (\pi_0, \pi_1, \ldots, \pi_{k-1}) \in \Pi^*,$$

i.e., a sequence of transition labels π_j, $0 \leq j \leq k-1$ corresponding to the transitions $t_j = (s_j, \pi_j, s_{j+1}) \in T$ of tc. The execution of a test case results in a sequence of visible behaviors, i.e., sequences of controllable input events $\pi_i \in \Pi_{i,sm}$ representing test stimuli injected by the tester and observable output events $\pi_o \in \Pi_{o,sm}$ which are component reactions expected from the component implementation to pass the test case. In case of non-deterministic state machine test model specifications, there is an n-to-1 relation between the set of test cases and the set of test runs.

Coverage criteria for state machine test models are, e.g., *all-states*, *all-transitions*, etc. thus selecting certain structural elements of a state machine test model sm into finite sets $TG \subseteq TG(sm)$ of test goals (Utting and Legeard, 2007). Correspondingly, we assume an n-to-m coverage relation between test cases and a set of test goals selected for some coverage criterion.

Example 3.3. Consider the sample state machine test model of the component *Central Locking System* (CLS) in Fig. 5 that specifies the activation/deactivation of the *CLS* via a car key. Each input/output event refers to a port depicted as small pins at the component interface. For coverage criterion *all-transitions*, the set of test goals is defined as $TG = \{t1, t2, t3, t4\}$. Hence, the test case $tc = (t1, t2, t3, t4)$ suffices to cover the set TG, where the expected test run is given as $exec(sm, tc) = (key\_pos\_lock\,?\,, cls\_lock\,!\,, key\_pos\_unlock\,?\,, cls\_unlock\,!\,)$.

Each architectural component of a software system under test obtains a dedicated state machine test model as behavioral specification for component testing. For integration testing, i.e., testing interactions among components by means of signals exchanged via architectural connectors, corresponding behavioral specifications may be obtained by building the product automaton for the components under consideration. However, especially when considering large-scale systems, this approach is, in general, impracticable due to

- the high number of interacting components and their potential combinations to be considered for any possible interaction scenario and
- the exponential complexity of the resulting product automata, i.e., the so-called state-space explosion problem, resulting in many mutually equivalent execution traces that only differ w.r.t. the interleaving of literally concurrent steps of different components.

To avoid these problems, we use predefined collections of message sequence charts (MSCs, ITU, 1999) as integration test model (Briand et al., 2012).

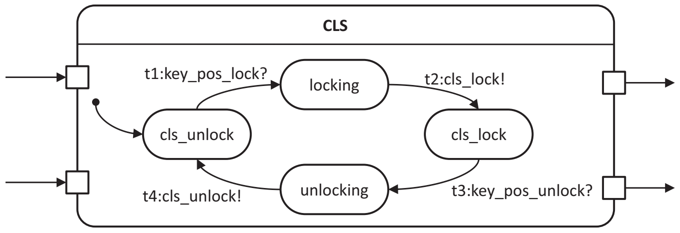


Fig. 5. Sample state machine test model for the Central Locking System component.

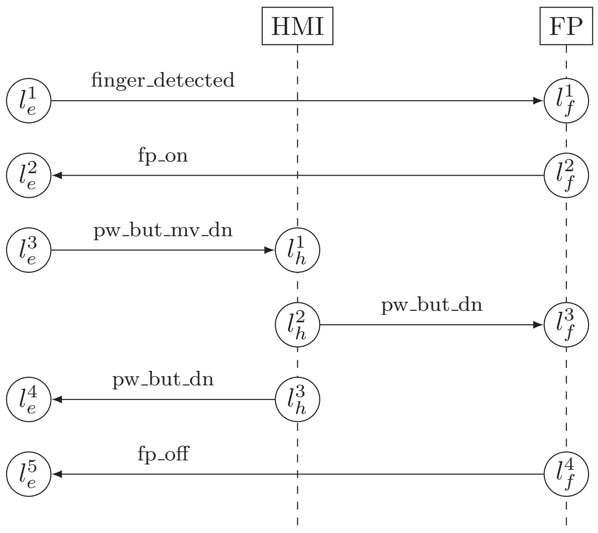


Fig. 6. Sample message sequence chart test scenario for sample BCS architecture.

3.3. *Message sequence chart test model for integration testing*

We assume an integration test model to define a collection of MSCs, where each MSC defines a valid sample interaction scenario of a subset of components and, therefore, representing an explicit test case specification. The set of interacting components occurring as instances in an MSC constitute a subsystem of the overall system under test participating in that particular integration test scenario.

Example 3.4. Consider the sample message sequence chart in Fig. 6 specifying an integration test scenario of the BCS case study. The test scenario describes the activation of the finger protection feature by detecting a finger clamped in the window and subsequently moving down the window. The MSC consists of two instances corresponding to the components *HMI* and *FP* (cf. Fig. 5) as well as the (implicit) environment instance. Each instance life line consists of a vertically ordered sequence of *locations* depicted as labeled circles representing sending or receiving event occurrences of messages, e.g., l_e^1 denotes the sending of a message (*finger_detected*) from the environment which is received at location l_f^1 of component *FP*.

Many different definitions and semantic interpretations of message sequence charts appear in the literature (Genest and Muscholl, 2005). Therefore, we formalize message sequence charts as used in our approach as integration test model specifications.

Definition 3.3 *(Message sequence chart).* A *message sequence chart* is a 4-tuple $(I, Loc, \lambda, <_{co})$, where I is a finite set of component *instances*, $Loc = \bigcup_{i \in I} Loc_i$ is a finite set of *locations* partitioned into subsets of instance locations Loc_i for each component instance $i \in I$, $\lambda : Loc \rightarrow \Pi$ is a *labeling function* assigning a label from a set Π of *signals* to each location and $<_{co} \subseteq Loc \times Loc$ is a *causal ordering* on instance locations.

Again, we assume the existence of a unique *environment instance* $\mathcal{E} \in I$ with special *external* locations $l_e \in Loc_{\mathcal{E}} \subseteq Loc$. Locations $l \in Loc$ are associated with *actions*, i.e., sending and receiving events of signals $\pi \in \Pi$ occurring in some orderings that correspond to the causal ordering relation $<_{co}$. Therefore, for the causal ordering relation $<_{co} = <_{tio} \cup <_{msg}$, we further require the following properties.

- $<_{tio} = \bigcup_{i \in I} <_{tio,i}$, where $<_{tio,i} \subseteq Loc_i \times Loc_i$ such that $<_{tio,i}$ is a *total ordering* of locations of instance $i \in I$.
- $<_{msg} \subseteq Loc \times Loc$ is a *message relation* such that for each location $l \in Loc \setminus Loc_{\mathcal{E}}$ there exists exactly one location $l' \in Loc \cup Loc_{\mathcal{E}}$, $l' \neq l$, such that either $l <_{msg} l'$ (*send message action*), or $l' <_{msg} l$ (*receive message action*) holds.

A location $l \in Loc$ denotes an *output message action* if $l <_{msg} l'$ and $l' \in Loc_{\mathcal{E}}$ holds, whereas l is a *receive message action* if $l' <_{msg} l$ holds. By $MSC(\Pi)$ we refer to the set of well-formed MSCs over a set Π of signals. Furthermore, by $MSC(\Pi, \mathcal{I})$ we denote the set of well-formed MSCs over a set Π of signals and a set $\mathcal{I}$ of instances, i.e., for each $msc \in MSC(\Pi, \mathcal{I})$ it holds that $I \subseteq \mathcal{I}$.

The MSC semantics defines sets of possible MSC executions in terms of valid location sequences conforming to the causal ordering relation $<_{co}$. A *test case* defined by a message sequence chart $msc \in MSC(\Pi)$ is a sequence $tc = (l_1, l_2, \ldots, l_k) \in Loc^*$ of k locations of msc. A test case $(l_1, l_2, \ldots, l_k)$ is further *valid* for an MSC $(I, Loc, \lambda, <_{co})$ if it is a linear extension of $(Loc, <_{co}^{+})$. In addition, we assume synchronous communication semantics, i.e., we require each location l_i in a test case tc that refers to a send action to be instantaneously followed by the corresponding location l_{i+1} of the receiving action, i.e., $l_i <_{msg} l_{i+1}$. By $TC(msc) \subseteq Loc^*$ we refer to the set of all *valid* test cases of a message sequence chart test model $msc \in MSC(\Pi)$. For a test case $tc = (l_1, l_2, \ldots, l_k)$ of $msc \in MSC(\Pi)$ we define a corresponding *test run* by

$$exec(msc, tc) = (\lambda(l_i), \lambda(l_j), \ldots, \lambda(l_k)) \in \Pi^*,$$

where $(l_i, l_j, \ldots, l_k)$ results from the restriction of tc onto input/output message actions.

Meaningful test scenarios of MSCs start with at least one input action from the environment and corresponding receive actions by the system components. In order to have observable behavior, there should be at least one output action to the environment. Note that there is a 1-to-n relation between a message sequence chart and its set of test cases, but a 1-to-1 relation between each test case and its respective test run.

Example 3.5. A valid test case for the sample message sequence chart in Fig. 6 is, e.g., $tc = (l_e^1, l_f^1, l_e^2, l_f^2, l_e^3, l_h^1, l_h^2, l_f^3, l_h^3, l_e^4, l_f^4, l_e^5)$ with its corresponding test run *exec*(*msc*,*tc*) = (*finger_detected*, *fp_on*, *pw_but_mv_dn*, *pw_but_dn*, *fp_off*).

For adopting the model-based testing artifact notions to integration testing using message sequence charts as test models, we consider the following correspondences. An integration test model for an architecture model $am = (C, \Pi, Con)$ consists of a finite collection $MSC \subseteq MSC(\Pi, C)$ of message sequence charts. Similar to the coverage criteria *all-states*, *all-transitions*, etc. of state machine test models, we consider *all-components* and *all-connectors* as architecture-based adequacy criteria for integration testing thus selecting corresponding integration test goals TG_{MSC} to be covered by the given integration test suite TS as the union of test suites $TC(msc)$ of each $msc \in MSC$. For instance, in case of *all-components*, each component $c \in C$ appears as an instance in at least one test case. In case of *all-connectors*, for each connector $con \in Con$ a respective pair of instance locations exchanging the corresponding signal is part of at least one test case.

We now integrate component test models and integration test models in an architecture test model.

3.4. *Architecture test model*

Based on the previous definitions for model-based testing, we define an architecture test model for integration testing that combines the structural architectural decomposition, as well as the behavioral test models for component and integration testing. The relationship between the three models assembled in an architecture test model is depicted in Fig. 3.

Definition 3.4 *(Architecture test model).* An *architecture test model* is a 4-tuple (Π, am, CTM, ITM), where Π is an finite set of *signals*, $am = (C, \Pi, Con) \in AM(\Pi)$ is an *architecture model* over signals Π, $CTM : C \to SM(\Pi)$ is a *component test model assignment function*, and $ITM \subseteq MSC(\Pi, C)$ is a finite set of *integration test models*.

Besides the structural decomposition into components $c \in C$ with dedicated intra-component behavioral specifications given as state machine test models $sm = CTM(c) \in SM(\Pi)$, an architecture test model further specifies inter-component behaviors by means of interaction scenarios $msc \in ITM$ among subsets $I \subseteq C$ of components and/or with the environment $\mathcal{E} \in C$.

The globally defined set of signals Π play the key role for combining and integrating the different views onto the architecture of a software under test.

(1) in the architecture model *am* the elements from the set Π are interpreted as connector signals exchanged between respective component ports,
(2) in state machine test models $sm = CTM(c)$ of architectural components $c \in C$, signals from the set Π are interpreted as input and output events being part of transition labels and
(3) in message sequence chart test models $msc \in ITM$ signals from the set Π refer to send and receive actions of locations occurring when signals are exchanged between component ports.

For an architecture test model (Π, am, CTM, ITM) to be *well-formed*, we require those three parts to refer to elements in Π in a *consistent* way. Consistency between a component interface and the state machine test model of that component requires input events triggering state machine transitions to refer to input port signals, and output events emitted by state machine transitions to refer to output port signals. Consistency between the architecture model and the set of message sequence chart test models requires pairs of actions, i.e., sending a message by some instance/component c and receiving of this message by an instance/component c' within an MSC scenario to correspond to respective connector definitions. By $ATM(\Pi)$ we refer to the set of all well-formed architecture test models defined over a set Π of signals. We implicitly assume the behavior of each component within an MSC test scenario to be supported by and to be consistent with the corresponding state machine specifications.

Architecture test models build the basis for two kinds of test scenarios, i.e., (1) during component testing the test cases obtained from the state machine test models of the components under test are executed on those components and (2) during integration testing the MSC test cases are executed on the respective sets of components and corresponding connectors participating in the interaction scenario. Hence, each test case *tc* affects a corresponding *test scenario subarchitecture* am_{tc} of *am*. In case of component testing, only the component itself constitutes the entire test scenario subarchitecture. In case of integration testing, am_{tc} comprises the set of components involved in the interaction scenario, and the interfaces are given by the subset of ports associated with the remaining input connectors and output connectors.

4. Delta-oriented architecture-based test modeling

The architecture test model introduced in the previous section allows for model-based component and integration testing of single software systems *p*. As described in Section 2.2, when considering families of similar systems p_i, $1 \le i \le n$, e.g., the set of system variants that corresponds to product configurations defined by a software product line or the system versions emerging as a result of software evolution over time, etc., corresponding families of architecture test models atm_i, $1 \le i \le n$ have to be specified.

For this, we introduce a variable architecture test model on the basis of a core test model atm_{core} and sets of model deltas $\delta_{atm,i}$ yielding respective architecture test model variants atm_i for system variants p_i when applied to the core model. Therefore, the delta-oriented architecture test model is decomposed into (1) architecture model deltas $\delta_{am,i}$ for constructing architecture model variants am_i, (2) state machine deltas $\delta_{CTM,i}$ for deriving the set CTM_i of component test model variants, i.e., state machine test model variants $sm_{i,j}$ for each component $c_j \in C_i$ being part of am_i, and (3) integration test model deltas $\delta_{ITM,i}$ for adapting the set ITM_i of integration test specifications to only those message sequence charts $msc \in ITM_i$ corresponding to valid interaction scenarios among components being part of am_i.

4.1. Delta-oriented architecture model

Based on our definition of an architecture model over a set Π of signals, we assume a core architecture model $am_{core} = (C_{core}, \Pi, Con_{core})$ of a delta-oriented architecture model to constitute a well-formed architecture model representing the architecture of a valid system variant. We further assume the set of components $C_{core} \subseteq \mathcal{C}$ to be implicitly contained in a globally predefined set $\mathcal{C}$ that includes all architecture components apparent in at least one architecture model variant. Correspondingly, we refer to the global set of all connectors by $\mathcal{CON} \subseteq \mathcal{C} \times \Pi \times \Pi \times \mathcal{C}$.

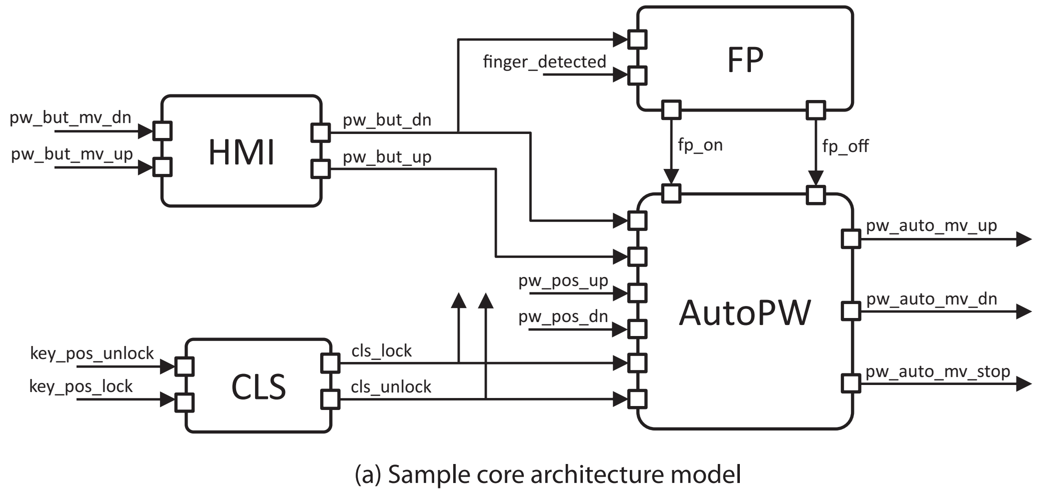
An *architecture model delta* transforms the core architecture model by applying appropriate delta operations for changing the component-connector graph, i.e., by adding and removing components and connectors in a consistent way, whereas more fine-grained *modifications*, e.g., concerning changes of connector input/output signals, are emulated by removing the former model element and adding an entire new variant of that element to the model variant. Formally, an architecture model delta is defined as follows.

Definition 4.1 *(Architecture Model Delta).* An *architecture model delta* $\delta_{am} \subseteq Op_{AM}$ defines a set of delta operations, where for each $c \in \mathcal{C}$, $\langle add\, c\rangle \in Op_{AM}$ and $\langle rem\, c\rangle \in Op_{AM}$ and for each $con \in \mathcal{CON}$, $\langle add\, con\rangle \in Op_{AM}$ and $\langle rem\, con\rangle \in Op_{AM}$ holds.

When deriving an architecture model variant am_i from the core model am_{core}, we have to ensure that a set of delta operations in δ_{am} is *applicable* to the core model or an intermediate model during the generation process. This means that a model element can only be added to the core or an intermediate model if it does not yet exist, and a model element can only be removed if it actually exists in the core or an intermediate model. Furthermore, we require every valid sequence of delta applications to result in a unique architecture model variant. To guarantee these two properties, we assume a partial ordering $after_{AM}$ to be defined on the set of architecture model deltas such that a delta δ_{am} is to be applied after a delta δ'_{am} if $\delta_{am} after \delta'_{am}$ holds. This *after* order is crucial for handling dependencies among the corresponding transformations caused by deltas as described in Clarke et al. (2010). We define the architecture delta application as follows.

Definition 4.2 *(Architecture model delta application).* The *application* of an architecture model delta $\delta_{am} \subseteq Op_{AM}$ to an architecture model $am \in AM(\Pi)$ is defined by a function $apply_{AM} : AM(\Pi) \times \mathcal{P}(Op_{AM}) \to AM(\Pi)$ with $apply_{AM}(am, \delta_{am}) = am' = (C', \Pi, Con')$, where

- if $\delta_{am} = \emptyset$, then $am' = am$,
- if $\delta_{am} = \{\langle op_{am}\rangle\} \cup \delta'_{am}$, then $am' = apply_{AM}(apply_{AM}(am, \{\langle op_{am}\rangle\}), \delta'_{am})$,
- if $\delta_{am} = \{\langle add\, c\rangle\}$, then $C' = C \cup \{c\}$ and $Con' = Con$,
- if $\delta_{am} = \{\langle rem\, c\rangle\}$, then $C' = C \setminus \{c\}$ and $Con' = Con$,



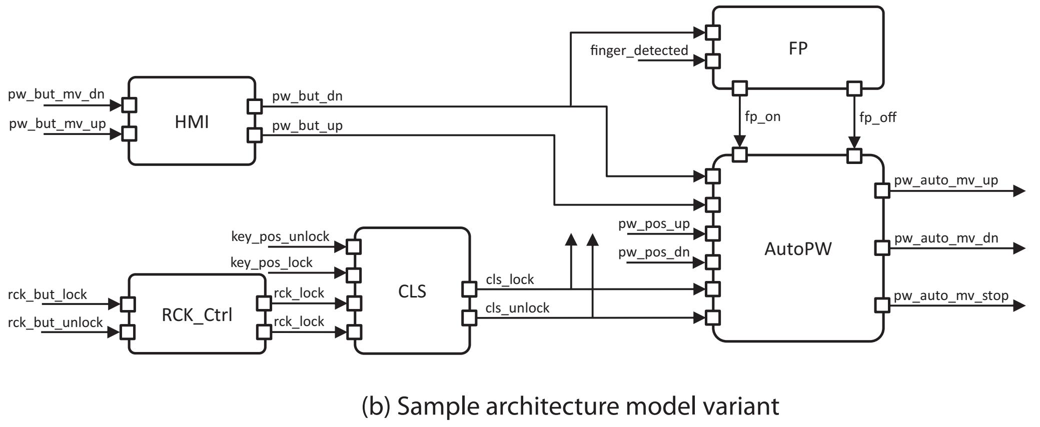


Fig. 7. Sample BCS architecture model variant after architecture model delta application.

- if $\delta_{am} = \{\langle add\,con\rangle\}$, then $Con' = Con \cup \{con\}$ and $C' = C$, and
- if $\delta_{am} = \{\langle rem\,con\rangle\}$, then $Con' = Con \setminus \{con\}$ and $C' = C$.

Although the definition of the function $apply_{AM}$ requires every (intermediate) model to be a well-formed architecture model during delta application, we may weaken this property to only hold after the set of all deltas for a particular system variant has been applied thus allowing intermediate models being ill-formed (cf. Schaefer et al., 2011).

The delta-oriented modeling approach allows any potential architecture model $am_{core} \in AM(\Pi)$ to be chosen as core model of a model family.

Proposition 4.1. *(***Existence of architecture model delta***). For each core architecture model $am_{core} \in AM(\Pi)$ and each architecture model variant $am \in AM(\Pi)$, there exists an architecture model delta $\delta_{am} \subseteq Op_{AM}$ with $am = apply_{AM}(am_{core}, \delta_{am})$.*

Thus, every possible model variant $am \in AM(\Pi)$ is derivable from an arbitrarily chosen core model am_{core} via at least one model delta δ_{am}. We refer to Lochau et al. (2012) for a detailed proof of a similar result for state machine deltas.

Example 4.1. Assume the architecture model in Fig. 7(a) to constitute the core architecture model. The application of the model delta $\delta_{am} = \{\langle add\,RCK\_Ctrl\rangle, \langle add\,(\mathcal{E}, \epsilon, rck\_but\_lock, RCK\_Ctrl)\rangle, \langle add\,(RCK\_Ctrl, rck\_lock, rck\_lock, CLS)\rangle, \langle add\,(\mathcal{E}, \epsilon, rck\_but\_unlock, RCK\_Ctrl)\rangle, \langle add\,(RCK\_Ctrl, rck\_unlock, rck\_unlock, CLS)\rangle\}$ generates the variant in Fig. 7(b) in which the newly added component *RCK_Ctrl* allows the additional control of the *CLS* by a remote control key.

The concepts introduced for delta-oriented architecture models are likewise applicable to component test models and integration test models.

4.2. Delta-oriented component and integration test model

Similar to architecture model deltas, *state machine deltas* transform the state-transition graph of the core model by adding and removing states and transitions, where we assume the initial state s_0 to remain unchanged. By $\mathcal{S}_c$ and $\mathcal{T}_c$ we denote the globally predefined set of all states and transitions ever occurring in possible state machine test model variants assigned to component $c \in \mathcal{C}$ whenever c is part of the corresponding architecture model variant. Similar to architecture model deltas, a state machine delta $\delta_{sm} \subseteq Op_{SM}$ defines a set of delta operations, where for each $s \in \mathcal{S}_c$, $\langle add\,s\rangle \in Op_{SM}$ and $\langle rem\,s\rangle \in Op_{SM}$ and for each $t \in \mathcal{T}_c$, $\langle add\,t\rangle \in Op_{SM}$ and $\langle rem\,t\rangle \in Op_{SM}$ holds.

We may omit index c for the sets $\mathcal{S}_c$ and $\mathcal{T}_c$ in the following if not relevant. Analogue to the architecture model delta definition, we assume an *after* order on a set of state machine deltas to ensure that the deltas are applicable during variant generation and that the resulting state machine variant is uniquely defined. Due to the similar structuring of component-connector graphs of architecture models and state-transition graphs of state machine models, Definition 4.2 is almost directly adoptable for the definition of a state machine delta application and the same holds for the existence result of state machine deltas for arbitrary core models as proven in Proposition 4.1. For details we refer to our previous work (Lochau et al., 2012).

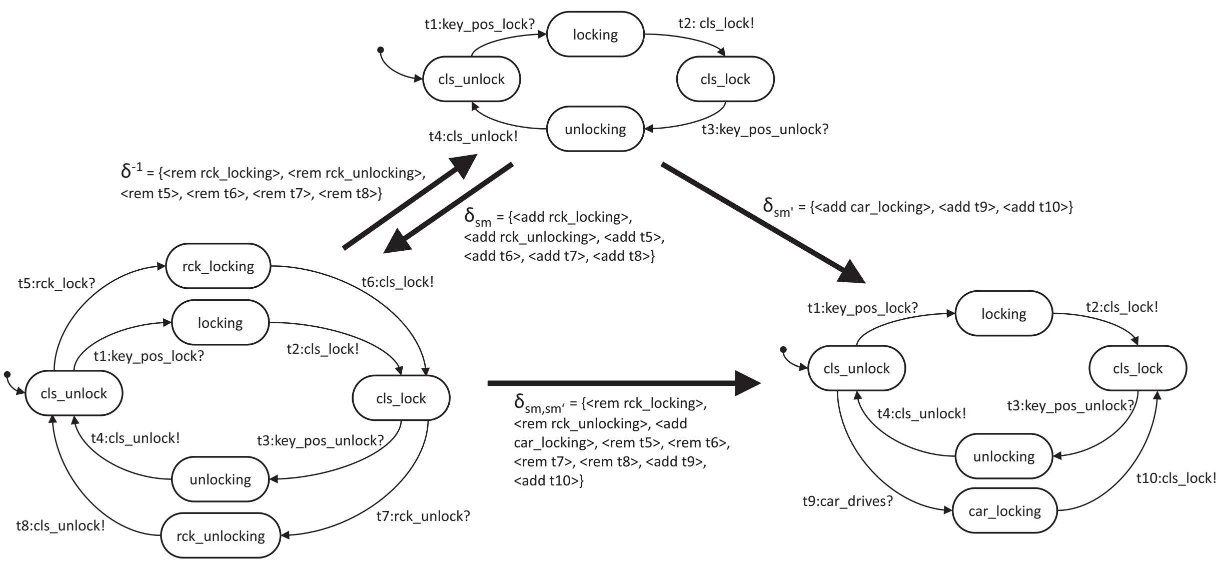

t1:key_pos_lock?
locking
t2: cls_lock!
cls_unlock
cls_lock
unlocking
t4:cls_unlock!
t3:key_pos_unlock?
δ-1 = {<rem rck_locking>, <rem rck_unlocking>, <rem t5>, <rem t6>, <rem t7>, <rem t8>}
δsm = {<add rck_locking>, <add rck_unlocking>, <add t5>, <add t6>, <add t7>, <add t8>}
δsm' = {<add car_locking>, <add t9>, <add t10>}
δsm,sm' = {<rem rck_locking>, <rem rck_unlocking>, <add car_locking>, <rem t5>, <rem t6>, <rem t7>, <rem t8>, <add t9>, <add t10>}
rck_locking
t5:rck_lock?
t6:cls_lock!
t2:cls_lock!
rck_unlocking
t7:rck_unlock?
t8:cls_unlock!
car_locking
t9:car_drives?
t10:cls_lock!

Fig. 8. Two sample CLS component state machine test model variants after state machine delta applications.

Example 4.2. Assume the state machine on the top of Fig. 8 to constitute the core state machine test model for the *Central Locking System* (cf. Fig. 5). When generating the variant on the left hand side, this core model is transformed by applying the delta δ_{sm}. In the resulting component variant, the locking/unlocking of the car is further controlled by a remote control key (*rck_lock* and *rck_unlock*). For generating the variant shown on the right hand side, we have to apply the delta $\delta_{sm'}$. Herein, an additional safety function is realized that locks the doors whenever the car drives (*car_drives*).

Finally, for the derivation of variants of integration test specifications *ITM* from a core $ITM_{core} \subseteq MSC(\Pi, \mathcal{C})$, we simply assume corresponding integration test model deltas to add and remove entire message sequence charts $msc \in MSC(\Pi, \mathcal{C})$ in an appropriate way. Hence, an *integration test model delta* $\delta_{ITM} \subseteq Op_{ITM}$ defines a set of delta operations, where for each $msc \in MSC(\Pi, \mathcal{C})$, $\langle add\, msc\rangle \in Op_{ITM}$ and $\langle rem\, msc\rangle \in Op_{ITM}$ holds. The application of those integration test model deltas are defined similar to the previous model delta definitions.

4.3. Delta-oriented architecture test model

Based on the definition of a core architecture model, a core state machine test model, and a core collection of message sequence chart test models, we require a core architecture test model $atm_{core} = (\Pi, am_{core}, CTM_{core}, ITM_{core})$ to constitute a well-formed architecture test model of some valid system variant. Thereupon, a component-wise definition of an *architecture test model delta* for transforming this core architecture test model to a model variant *atm* can be given as follows.

Definition 4.3 *(Architecture test model delta).* An *architecture test model delta* is a triple $(\delta_{am}, \Delta_{CTM}, \delta_{ITM})$, where δ_{am} is an *architecture model delta*, $\Delta_{CTM} : \mathcal{C} \to \mathcal{P}(Op_{SM})$ is a *component state machine delta assignment function* and δ_{ITM} is an *integration test model delta.*

The *application* of an architecture test model delta $\delta_{atm} = (\delta_{am}, \Delta_{CTM}, \delta_{ITM})$ to an architecture test model $atm \in ATM(\Pi)$ yields an architecture test model variant $atm' = apply_{ATM}(atm, \delta_{atm})$, where

- $am' = apply_{AM}(am, \delta_{am}) = (C', \Pi, Con')$,
- for each $c \in C'$, $CTM'(c) := apply_{SM}(CTM(c), \Delta(c))$ and
- $ITM' = apply_{ITM}(ITM, \delta_{ITM})$.

For a well-defined application of a delta $\delta_{atm,i}$ for system variant p_i to a core test model architecture $atm_{core} = (\Pi, am_{core}, CTM_{core}, ITM_{core})$, we have to ensure that CTM_{core} is defined on the entire set $\mathcal{C}$ of components. Therefore, we assume for each component $c \in \mathcal{C}$ a mapping to some core state machine test model $CTM_{core}(c) \in SM(\Pi)$ even if $c \notin C_{core}$, e.g., using a plain machine definition $sm_{c,core} = (\{s_0\}, s_0, \emptyset)$ solely introducing the initial state s_0. Correspondingly, the definition of the function $\Delta_{CTM,i}$ for system p_i requires a state machine delta assignment for each component $c \in \mathcal{C}$ even if $c \notin C_i$, e.g., by setting $\Delta(c) = \emptyset$.

As the architecture test model assembles definitions and applications of model deltas already introduced, the corresponding results concerning well-formedness, existence, etc. are applicable to architecture test model definitions, accordingly. In addition, well-formedness requires the integration test model variant $ITM_i = apply(ITM_{core}, \delta_{ITM,i})$ to comply with the architecture model variant am_i as described in Section 3.4.

Example 4.3. Assume a core architecture test model $atm_{core} = (\Pi, am_{core}, CTM_{core}, ITM_{core})$ that comprises the core architecture model depicted in Fig. 4, the core state machine $CTM_{core}(CLS)$ shown in Fig. 5, and the integration test specification $msc \in ITM_{core}$ illustrated in Fig. 6. The architecture test model delta $\delta_{atm} = (\delta_{am}, \Delta_{CTM}, \delta_{ITM})$ transforms the core model to yield a model variant, where δ_{am} has been described in Example 4.1, $\Delta_{CTM}(CLS) = \delta_{sm}$ is depicted in Fig. 8 on the left side, and δ_{ITM} adds msc' to specify a scenario in which the car is locked and unlocked using a remote control key.

The delta-oriented architecture test model introduced builds the basis for an incremental model-based testing methodology incorporating principles known from regression testing for a preplanned reuse of test artifacts.

5. Incremental architecture-based testing of large-scale systems

We now describe a comprehensive approach for efficient model-based component and integration testing of families of large-scale software systems based on the delta-oriented evolution of test artifacts (cf. Section 2.3). We show in detail how the corresponding architecture test model *regression delta* is constructed from the individual regression deltas of the component and integration test models. When stepping from system variant p_i to the subsequent variant p_{i+1}, we incrementally test families of large-scale software systems by exploiting the architecture test model regression delta explicitly capturing the differences between two variants under test.

5.1. Regression delta construction

As shown in Fig. 1(c), the incremental component test artifact evolution is divided into four levels corresponding to the individual parts of state machine test artifacts defined in Section 3.2. For each step from state machine variant sm_i to a subsequent variant sm_{i+1} the following is performed.

(1) Reuse of common component test artifacts of sm_i for sm_{i+1},
(2) Invalidation of test artifacts of sm_i not being reusable for sm_{i+1} and
(3) Generation of new artifacts for sm_{i+1} missing in those of sm_i.

As illustrated in Fig. 1, for the artifact evolution it has to be guaranteed that the resulting test artifacts preserve the relationships described in Section 2.1.

Based on delta-oriented component state machine test model specifications, the differences between component variants w.r.t. the core test model are explicitly specified in the corresponding model delta of that variant. However, for the incremental evolution of component test artifacts, the differences of a component state machine test model sm_{i+1} to the previous state machine test model variant sm_i of that component are to be investigated. Therefore, we define a *regression delta* that encapsulates all changes between two component test model variants when evolving from sm_i to sm_{i+1}.

Definition 5.1 *(State machine regression delta).* A *state machine regression delta* $\delta_{sm_i,sm_{i+1}} \subseteq Op_{SM}$ for a pair $(sm_i, sm_{i+1}) \in SM(\Pi) \times SM(\Pi)$ is a state machine delta such that $sm_{i+1} = apply_{SM}(sm_i, \delta_{sm_i,sm_{i+1}})$.

Assume that the state machine sm_i can be derived from the core by delta δ_{sm_i} and state machine sm_{i+1} by $\delta_{sm_{i+1}}$, accordingly. The *inverse* $\delta^{-1} \subseteq Op_{SM}$ of a component state machine delta $\delta \subseteq Op_{SM}$ is defined by inverting each individual delta operation $op \in \delta$ to $op^{-1} \in \delta^{-1}$, where $\langle add\ e\rangle^{-1} = \langle rem\ e\rangle$ and $\langle rem\ e\rangle^{-1} = \langle add\ e\rangle$ for any model element $e \in \mathcal{S} \cup \mathcal{T}$.

Proposition 5.1. **(State machine regression delta construction**). *For two state machine deltas $\delta_{sm_i} \subseteq Op_{SM}$ and $\delta_{sm_{i+1}} \subseteq Op_{SM}$, the regression delta is given as $\delta_{sm_i,sm_{i+1}} = (\delta_{sm_i} \setminus \delta_{sm_{i+1}})^{-1} \cup (\delta_{sm_{i+1}} \setminus \delta_{sm_i})$.*

We refer to Lochau et al. (2012) for a detailed proof.

Example 5.1. The regression delta between the test model *sm* (on the left) and *sm′* (on the right) in Fig. 8 results in $\delta_{sm,sm'} = (\delta_{sm} \setminus \delta_{sm'})^{-1} \cup (\delta_{sm'} \setminus \delta_{sm}) = \{\langle rem\ rck\_locking\rangle, \langle rem\ rck\_unlocking\rangle, \langle add\ car\_locking\rangle, \langle rem\ t5\rangle, \langle rem\ t6\rangle, \langle rem\ t7\rangle, \langle rem\ t8\rangle, \langle add\ t9\rangle, \langle add\ t10\rangle\}$.

Based on the state machine regression delta, we now describe the derivation of further types of regression deltas for the component test artifacts.

Test goal delta. The incremental evolution of the set of test goals and, therefore, the derivation of the *test goal regression delta* $\delta_{TG_i,TG_{i+1}}$ depends on the coverage criterion applied. Considering simple structural criteria like *all-states* and *all-transitions* (Utting and Legeard, 2007), i.e., criteria with $TG \subseteq \mathcal{S} \cup \mathcal{T}$, the test goal regression delta is directly derivable from the test model regression delta as follows.

- $\forall \langle rem e\rangle \in \delta_{sm_i,sm_{i+1}} : e \in TG_i \Rightarrow \langle rem\ e\rangle \in \delta_{TG_i,TG_{i+1}}$ and
- $\forall \langle add\ e\rangle \in \delta_{sm_i,sm_{i+1}} : e \in TG_{i+1}(sm_{i+1}) \Rightarrow \langle add\ e\rangle \in \delta_{TG_i,TG_{i+1}}$.

Accordingly, using complex coverage criteria, e.g., path-oriented criteria like MC/DC coverage (Utting and Legeard, 2007), we require a (partial) regeneration of the set of test goals, where the test model regression delta indicates those model parts in sm_{i+1} which are potentially affected.

Example 5.2. In Fig. 8 the test goal set for the model variant *sm* is defined by the all-transition criterion as $TG = \{t1, t2, t3, t4, t5, t6, t7, t8\}$. Based on the test model regression delta specified in Example 5.1, the test goal regression delta is constructed such that $\delta_{TG,TG'} = \{\langle rem\ t5\rangle, \langle rem\ t6\rangle, \langle rem\ t7\rangle, \langle rem\ t8\rangle, \langle add\ t9\rangle, \langle add\ t10\rangle\}$ holds.

Test suite delta. As explained in Section 2.3, we partition component test suites $TS_i = TS_{V,i} \cup TS_{O,i}$ into subsets of *valid* and *obsolete* test cases. When evolving $TS_i = TS_{V,i} \cup TS_{O,i}$ to $TS_{i+1} = TS_{V,i+1} \cup TS_{O,i+1}$ via the test suite regression delta $\delta_{TS_i,TS_{i+1}}$, the changes may affect both sets. Therefore, we also divide the test suite regression delta into sub deltas $\delta_{TS_{V,i},TS_{V,i+1}}$ and $\delta_{TS_{O,i},TS_{O,i+1}}$. By $T_{tc} \subseteq \mathcal{T}$, we refer to the subset of transitions from $\mathcal{T}$ such that $tc \in T_{tc}^*$ and T_{tc} is *minimal*. A test case *tc* is valid for test model $sm_i = (S, s_0, T)$ if $T_{tc} \subseteq T$, whereas $T_{tc} \nsubseteq T$ holds for obsolete test cases, respectively. An obsolete test case $tc \in TS_{O,i}$ for test model variant sm_i becomes valid for component variant sm_{i+1} if

$$\forall t \in T_{tc} \setminus T : \exists \langle add\ t\rangle \in \delta_{sm_i,sm_{i+1}} \Rightarrow \langle add\ tc\rangle \in \delta_{TS_{V,i},TS_{V,i+1}} \wedge \langle rem\ tc\rangle \in \delta_{TS_{O,i},TS_{O,i+1}}$$

holds, i.e., the set of transitions of *tc* missing in the set *T* of sm_i is added to sm_{i+1} via the test model regression delta. Correspondingly, valid test cases $tc \in TS_{V,i}$ become obsolete if

$$\exists t \in T_{tc} : \langle rem\ t\rangle \in \delta_{sm_i,sm_{i+1}} \Rightarrow \langle add\ tc\rangle \in \delta_{TS_{O,i},TS_{O,i+1}} \wedge \langle rem\ tc\rangle \in \delta_{TS_{V,i},TS_{V,i+1}}$$

holds.

The set of reusable test cases $TS_R = TS_{V,i} \cap TS_{V,i+1}$, therefore, contains those test cases being valid for the corresponding component variant of p_i as well as p_{i+1}. In addition to TS_R, further test cases may be required in TS_{i+1} to cover all test goals in TG_{i+1}. A test goal $tg \in TG_{i+1}$ is not covered by TS_R if either

- $\langle add\ tg\rangle \in \delta_{TG_i,TG_{i+1}}$, i.e., the test goal is newly added to sm_{i+1}, or
- $\forall tc \in TS_{V,i} : covers(tc, tg) \Rightarrow tc \in TS_{O,i+1}$, i.e., all test cases of sm_i covering *tg* are obsolete for sm_{i+1}.

For covering those test goals, previously obsolete test cases $tc \in TS_{O,i} \cap TS_{V,i+1}$ with $covers(tc, tg)$ may exist thus becoming valid again by adding them to TS_R.

Otherwise, a new test case $tc_{tg} = gen(sm_{i+1}, tg)$ is required, where $\langle add\ tc_{tg}\rangle \in \delta_{TS_{V,i},TS_{V,i+1}}$. By $tc = gen(sm, tg)$ we denote a generation mechanism for a test case *tc* covering a test goal *tg* on test model *sm* and, therefore, assume the existence of a (black-box) *test case generator* in the following (cf. e.g., Fraser et al., 2009). Furthermore, $TS = gen(sm, TG)$ denotes the generation of a test suite covering the set *TG* of all test goals for a coverage criterion on test model *sm*. The set of all new test cases generated for sm_{i+1} is added to the set $TS_{N,i+1}$ according to the notions used in regression testing.

Example 5.3. Consider the test cases $tc1 = (t1, t2)$ and $tc2 = (t1, t2, t3, t4)$ generated from the core test model in Fig. 8 for the *all-transition* coverage criterion. When stepping from the core model to *sm*, *tc*1 and *tc*2 both are reusable for the model variant and one new test case $tc3 = (t5, t6, t7, t8)$ is generated. When evolving to *sm′* the test cases *tc*1 and *tc*2 are reusable again, whereas *tc*3 becomes obsolete, thus one new test case $tc4 = (t9, t10)$ is to be generated to cover all test goals. The test case *tc*3 becomes valid again in subsequent component variants of the Central Locking System also being controlled by the remote control key.

Test plan delta. Stepping from component test model sm_i to a subsequent model variant sm_{i+1}, we further have to adapt the test plan $TP_i \subseteq TS_{V,i}$ to $TP_{i+1} \subseteq TS_{V,i+1}$ for sm_{i+1}. The evolved test plan $TP_{i+1} = TS_{N,i+1} \cup TS_{RT}$ comprises all new test cases $tc \in TS_{N,i+1}$ as well as reusable test cases $tc \in TS_{RT} \subseteq TS_R$ being selected for retest to verify that changes do not erroneously affect common behaviors already tested by TS_R. For the selection of TS_{RT}, different *retest strategies* appear in the literature on classical regression testing (Engström et al., 2008), e.g., *retest-all*, i.e., $TS_{RT} = TS_R$, *retest-non*, i.e., $TS_{RT} = \emptyset$, and *retest-random*, where an arbitrary subset $TS_{RT} \subseteq TS_R$ is chosen.

Change impact analysis via *program slicing* (Gupta et al., 1992) and/or *test model slicing* (Kamischke et al., 2012) may further support appropriate retest selection by means of the following criterion.

$$tc \in TS_{RT} :\Leftrightarrow (exec(sm_i, tc) \approx_{te} exec(sm_{i+1}, tc)) \not\Rightarrow exec(c_i, tc)$$
$$\approx_{te} exec(c_{i+1}, tc)$$

Summarizing, the test plan regression delta $\delta_{TP_i,TP_{i+1}}$ is defined by the following rules.

- $\forall tc \in TP_i \setminus TS_{RT} : \langle rem\ tc \rangle \in \delta_{TP_i,TP_{i+1}}$
- $\forall tc \in TS_{RT} \setminus TP_i : \langle add\ tc \rangle \in \delta_{TP_i,TP_{i+1}}$
- $\forall tc \in TS_{N,i+1} : \langle add\ tc \rangle \in \delta_{TP_i,TP_{i+1}}$

Soundness of the approach. For the soundness of the presented approach, we require the resulting test artifacts to be (1) *valid*, i.e., every test suite solely contains test cases being valid for the respective test model variant, and (2) *complete*, i.e., guaranteeing complete test goal coverage of the test model variant.

Let $TA_1, TA_2, \ldots, TA_n$ be the collection of test artifacts incrementally built for a sequence of components $c_1, c_2, \ldots, c_n$ via deltas on test artifacts as described above.

Theorem 5.1. *(Validity of component test suites) For component test suites TS_i of each TA_i, $1 \le i \le n$, $TS_{V,i} \subseteq TC(sm_i)$ holds.*

Theorem 5.2. *(Completeness of component test suites) For component test suites TS_i and sets of test goals TG_i of each TA_i, $1 \le i \le n$, (1) $TG_i = TG(sm_i)$ holds, and (2) TS_i covers TG_i.*

We refer to Lochau et al. (2012) for detailed proofs.

Architecture model and integration test model regression delta. For the architecture model as well as the integration test model, a similar construction is applicable. We construct an architecture model regression delta $\delta_{am_i,am_{i+1}} \subseteq Op_{AM}$ that (1) removes components $c \in C_i$ and connectors $con \in Con_i$ from am_i not contained in am_{i+1}, (2) adds those components $c' \in C_{i+1}$ and connectors $con' \in Con_{i+1}$ to am_{i+1} not contained in am_i and (3) does not affect elements common to both architecture models.

The integration test model regression delta $\delta_{ITM_i,ITM_{i+1}} \subseteq Op_{ITM}$ is constructed based on the architecture model regression delta $\delta_{am_i,am_{i+1}}$ such that (1) it removes those test models $msc \in MSC(\Pi, C_i)$ from ITM_i not contained in ITM_{i+1}, (2) it adds those test models $msc' \in MSC(\Pi, C_{i+1})$ to ITM_{i+1} not contained in ITM_i and (3) it does not affect test models $msc'' \in ITM_i \cap ITM_{i+1}$ defined in both test models. Correspondingly, the evolution of the set $TG_{ITM,i}$ of integration test goals, e.g., components and connectors, towards the set $TG_{ITM,i+1}$ by $\delta_{TG_{ITM,i},TG_{ITM,i+1}}$ can be derived from $\delta_{am_i,am_{i+1}}$ similar to the test goal evolution of state machine models. On the basis of the integration test model and the integration test goal regression delta, the integration test suite delta $\delta_{TS_{ITM,i},TS_{ITM,i+1}}$ evolves the test suite $TS_{ITM,i}$ to $TS_{ITM,i+1}$ similar to the approach as described above, i.e., by adapting the sets of valid and obsolete integration test cases. The integration test suite regression delta is used to define the integration test plan regression delta $\delta_{TP_{ITM,i},TP_{ITM,i+1}}$ for the incremental evolution of the test plan $TP_{ITM,i}$ to $TP_{ITM,i+1}$ similar to the test plan delta for state machine models, where the integration test plan comprises those test cases to be (re-)tested on the respective system variant.

5.2. Delta-oriented architecture-based test artifact evolution

The integration of both the component and integration regression delta approach into an architecture test model regression delta builds the basis for a comprehensive delta-oriented incremental test artifact evolution. Therefore, we define an incremental testing process as depicted in Fig. 9(a) and (b) incorporating the incremental test artifact evolution on both testing levels. Starting from the core architecture test model atm_{core}, the test artifacts are initially derived as usual, i.e.,

- for each component $c \in C_{core}$, the set TG_c of state machine test goals is selected from the component test model $sm_c = CTM_{core}(c)$ and a corresponding test suite $TS_c \subseteq TC(sm_c)$ is generated and
- for testing the component integration as specified in the architecture model am_{core}, an integration test suite $TS_{ITM_{core}} = \bigcup_{msc \in ITM_{core}} tc \in TC(msc)$ is applied to the corresponding test scenario subarchitectures am_{tc} to cover the set of integration test goals $TG_{ITM_{core}}$.

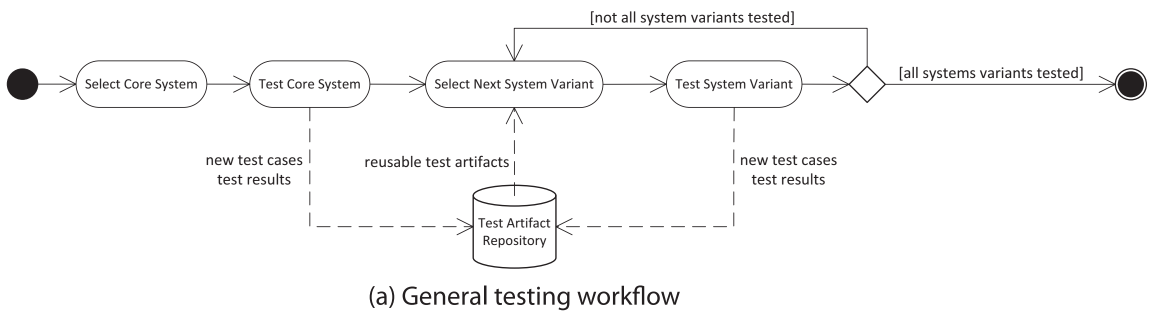
We record these test artifacts in our *test artifact repository* used as a database for the subsequent testing campaigns of the remaining system variants, enabling the access to reusable test artifacts. As the next step in our incremental testing process (cf. Fig. 9(a)), we select and test a system variant from the set of remaining system variants under test until all variants are tested. Here, we reuse existing test artifacts from the repository based on their incremental evolution and further record newly derived testing artifacts. The incremental evolution of those test artifacts while stepping from system variant p_i under test to a subsequent system variant p_{i+1} is performed on the basis of the corresponding architecture test model regression delta.

Definition 5.2 *(Architecture test model regression delta).* An *architecture test model regression delta* for a pair $(atm_i, atm_{i+1}) \in ATM(\Pi) \times ATM(\Pi)$ of architecture test models is a triple $(\delta_{am_i,am_{i+1}}, \Delta_{CTM_i,CTM_{i+1}}, \delta_{ITM_i,ITM_{i+1}})$, where

- $\delta_{am_i,am_{i+1}} \subseteq Op_{AM}$ is an architecture model regression delta,
- $\Delta_{CTM_i,CTM_{i+1}} : \mathcal{C} \to \mathcal{P}(Op_{SM})$ is a component state machine regression delta assignment function, where for each $c \in \mathcal{C}$, $\Delta_{CTM_i,CTM_{i+1}}(c) = \delta_{sm_{c,i},sm_{c,i+1}} \subseteq Op_{SM}$ such that $\langle add\ e \rangle, \langle rem\ e \rangle \in \delta_{sm_{c,i},sm_{c,i+1}}$ with $e \in \mathcal{S}_c \cup \mathcal{T}_c$ holds, and
- $\delta_{ITM_i,ITM_{i+1}} \subseteq Op_{ITM}$ is an integration test model regression delta.

Thus, the architecture test model regression delta is decomposed into the respective regression deltas of the different models as defined above. The evolution of component and integration test artifacts when stepping from architecture test model variant atm_j to variant atm_{j+1}, $1 \le j \le n$ is performed as follows, where in Fig. 9(b) the incremental component test artifact evolution is shown as an example.

Component test artifact evolution. For each component $c \in C_{j+1}$, the corresponding state machine regression delta $\delta_{sm_i,sm_{j+1}} = \Delta_{CTM_j,CTM_{j+1}}(c)$ refers to the state machine test model variant $sm_{c,i} = CTM_i(c)$ defined in the previously considered architecture test model variant atm_i in which component c recently occurred,



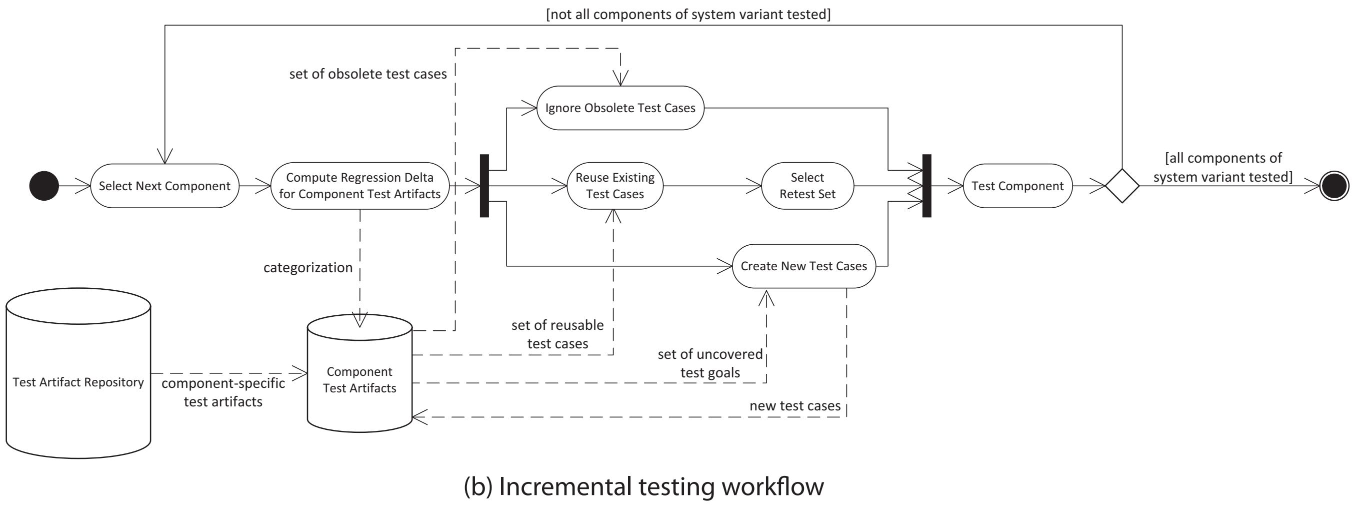

[not all components of system variant tested]
set of obsolete test cases
Ignore Obsolete Test Cases
Select Next Component
Compute Regression Delta for Component Test Artifacts
Reuse Existing Test Cases
Select Retest Set
Test Component
[all components of system variant tested]
Create New Test Cases
categorization
set of reusable test cases
set of uncovered test goals
Test Artifact Repository
component-specific test artifacts
Component Test Artifacts
new test cases
(b) Incremental testing workflow

Fig. 9. Incremental architecture-based testing process.

i.e., with the greatest index $i \leq j$ such that $c \in C_j$. Based on the test artifacts obtained for component c (cf. Fig. 9(b)) from that previous system variant under test, the component test artifact evolution is applied as described in the previous section. The regression delta computed for the component-specific testing artifacts supports the artifact categorization for the corresponding testing campaign. Based on this categorization, (1) obsolete test cases are ignored, (2) existing test cases that remain valid for the new component variant are reused and analyzed whether to be retested, and (3) for each currently not covered test goal new test cases are generated and recorded in the global test artifact repository. The incremental evolution of the integration test artifacts is done in a similar way as a result of the corresponding integration test regression delta applications described in the previous section.

Integration test artifact evolution. Based on the integration test model regression delta $\delta_{ITM_j,ITM_{j+1}}$ the set of valid MSCs is adapted such that each MSC scenario refers to a valid subarchitecture of the corresponding architecture model am_{j+1} of the subsequent architecture test model atm_{j+1}. The integration test goal set $TG_{ITM,\,j}$ of the architecture test model atm_j evolves to $TG_{ITM,\,j+1}$ of atm_{j+1} by applying $\delta_{TG_{ITM,\,j},TG_{ITM,\,j+1}}$, accordingly. Covering the integration test goal set $TG_{ITM,\,j+1}$ of atm_{j+1} requires the evolution of the integration test suite $TS_{ITM,\,j}$ via $\delta_{TS_{ITM,\,j},TS_{ITM,\,j+1}}$ such that new and common integration test goals are covered by newly defined or existing reusable integration test cases in $TS_{ITM,\,j+1}$ to ensure a complete test coverage of atm_{j+1}. Based on the integration test suite regression delta, the test plan regression delta $\delta_{TP_{ITM,\,j},TP_{ITM,\,j+1}}$ is derived by adapting the previous test plan $TP_{ITM,\,j}$ arranged for atm_j to $TP_{ITM,\,j+1}$ of atm_{j+1} such that $TP_{ITM,\,j+1}$ includes newly defined integration test cases as well as those reusable test cases selected for retesting. Here, retest obligations potentially arise due to different component variants within an already tested test scenario subarchitecture, e.g., changes at component interface specifications and/or impacts on intra-component behaviors. We consider $\Delta_{CTM_j,CTM_{j+1}}$ for the detection of changing intra-component behaviors, whereas the changes of component interfaces can be determined using the connector definitions in $\delta_{am_j,am_{j+1}}$. Furthermore, we reuse the knowledge of the integration of the different component variants within already tested test scenario subarchitectures such that retesting is solely required if new component interactions for the corresponding test scenario potentially emerge.

In accordance to Theorems 5.1 and 5.2 concerning the incremental component test artifact evolution, we require for the *soundness* of the incremental integration test artifact evolution every architecture test model to be (1) *valid*, i.e., every test scenario defined by either a component or an integration test case refers to a valid component and/or interaction behavior for a subarchitecture of the underlying architecture variant, and (2) *complete*, i.e., the integration test model guarantees complete integration test coverage w.r.t. the architectural coverage criterion under consideration.

Example 5.4. Consider system p as depicted in Fig. 7(b) to evolve to a subsequent system p' as shown in Fig. 10. The architecture model regression delta $\delta_{am,am'} = \{\langle add(\Sigma, \epsilon, pw\_rm\_up, RCK\_Ctrl)\rangle, \langle add(\Sigma, \epsilon, pw\_rm\_dn, RCK\_Ctrl)\rangle, \langle add(\Sigma, \epsilon, car\_drives, CLS)\rangle, \langle add(RCK\_Ctrl, pw\_but\_up, pw\_but\_up, AutoPW)\rangle, \langle add(RCK\_Ctrl, pw\_but\_dn, pw\_but\_dn, AutoPW)\rangle\}$ transforms the architecture model am of p to the architecture model variant am' of p' by adding connectors for additional interactions. The component state machine regression delta assignment function $\Delta_{CTM,CTM'}$, e.g., for the Central Locking System $\Delta_{CTM,CTM'}(CLS)$ is derived as described above (cf. Examples 5.1–5.3). As the architecture model regression delta contains neither add nor remove operations for any components in this example, the integration test model regression delta $\delta_{ITM,ITM'} = \emptyset$ imposes no transformations. The

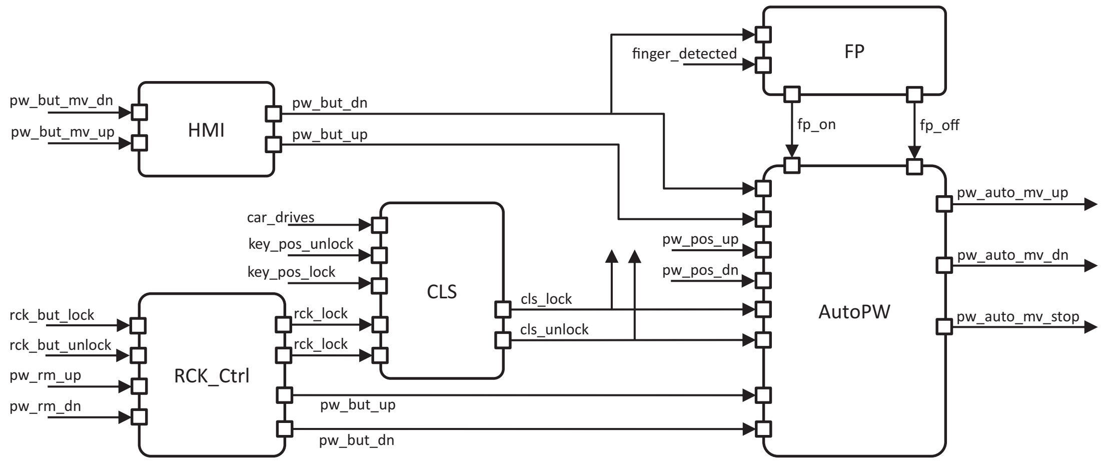


Fig. 10. Sample BCS architecture model variant after regression delta application.

corresponding architecture test model regression delta is assembled from those three regression deltas. Considering *all-connectors* as integration coverage criterion, the integration test goal set TG_{ITM} evolves such that the five connectors introduced in the architecture model regression delta are added to the integration test goal set TG'_{ITM}. For covering the new test goals, two additional MSC test case specifications are defined, the first for describing a scenario where the window is moved up and down controlled by the remote control key, and the second enforces automated locking to be active when the car is driving.

6. Implementation

In this section, we describe the implementation of our incremental model-based testing approach in a sample tool chain.

The general structure of the sample implementation of our model-based incremental component and integration testing approach is shown in Fig. 11. The framework comprises two components, namely *eDeltaMBT*, i.e., a collection of ECLIPSE plug-ins based on the Eclipse Modeling Framework[1] (EMF) as well as the Xtext Framework,[2] and IBM RATIONAL RHAPSODY.[3] EMF allows for the specification of a common data model for the various testing artifacts by means of corresponding meta-models. In addition, EMF provides capabilities for an automated generation of editors for modeling purposes based on those defined meta-models. We use Xtext for the definition of a domain specific language derived from the corresponding meta-model for modeling delta-oriented architecture models.

As depicted in Fig. 11, *eDeltaMBT* comprises the following plug-ins.

- *Configuration Repository* – The plug-in enables the configuration of system variants used for the automated generation of the corresponding test models for component as well as integration testing purposes.
- *DeltaSM* – The plug-in allows the modeling of delta-oriented component state machine test models in an EMF-based editor. The editor provides the definition of a core test model together with the definition of corresponding state machine deltas. Furthermore, the plug-in supports the modeling process by the validation of modeling constraints guaranteeing well-formed delta models. Based on the RHAPSODY library for `Java`, the component state machine test models are directly imported for the automated test case generation.
- *DeltARX* – The plug-in enables the modeling of delta-oriented architecture models in an Xtext-based editor. The editor provides the specification of a core architecture model and the specification of architecture model deltas.
- *TArE* – The plug-in implements the import of test cases from RHAPSODY and the subsequent delta-oriented test artifact evolution for component as well as integration testing.

IBM RATIONAL RHAPSODY is a full-featured UML CASE tool thus providing the modeling and visualization of state machines, message sequence charts, etc. as well as add-ons for automated test case generation. In addition, the RHAPSODY library for `Java` enables the automated import/export of models and test cases. In our toolchain, we apply IBM RATIONAL RHAPSODY in two ways.

(1) We import the generated component state machine test models and apply the add-on ATG for automated test case generation for testing at component level.
(2) We specify integration level test cases using message sequence charts as provided by RHAPSODY.

The EMF meta-models and the plug-in architecture of *eDeltaMBT* allows for a flexible replacement of IBM RATIONAL RHAPSODY by alternative tools with similar functionality. For this, the import/export of testing artifacts has to be adapted, e.g., the imported artifacts are to be transformed into our internal representation.

We applied this prototypical tool chain implementation to evaluate our incremental model-based component and integration testing approach. The case study design as well as the results of our evaluation are discussed in the following sections.

7. Body comfort system case study

In this section, we present the case study *Body Comfort System* following the presentation guidelines of Runeson and Höst (2009). We first introduce the case study design and then further present the process of data collection describing the kind of data we used for evaluation. The data analysis is explained afterwards and threats to validity concerning the case study are discussed.

[1] http://www.eclipse.org/modeling/emf/.
[2] http://www.eclipse.org/Xtext/.
[3] http://www-03.ibm.com/software/products/us/en/ratirhapfami/.

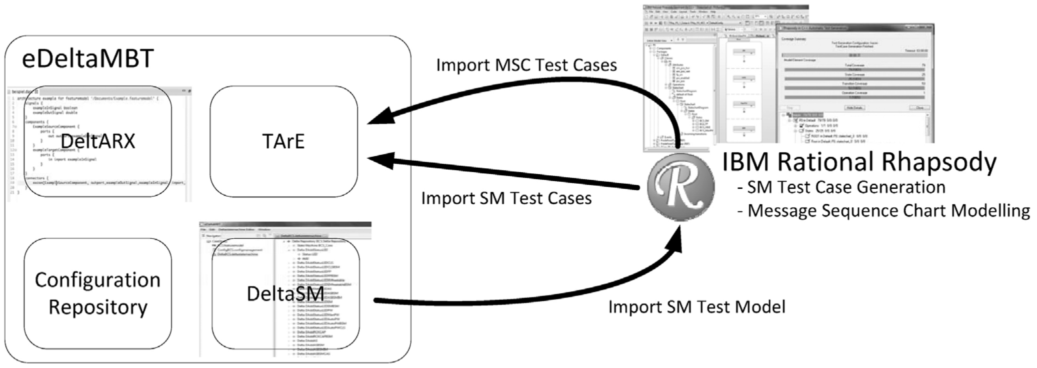


Fig. 11. Sample tool chain for incremental model-based component and integration testing.

7.1. Case study design

Software testing is a crucial discipline in automotive development for quality assurance. Since the degree of software in modern cars constantly increases, e.g., for implementing innovative functionality like driver assistance systems, testing has become a more and more complex task. In addition, the high degree of variability in form of various system configurations requires more efficient testing techniques taking the commonality and variability of variant-rich systems into account like, e.g., regression testing techniques. Therefore, we select the process of quality assurance in the automotive domain as *case* for our case study. We design the study as a descriptive and improving case study, where we present the current software system testing process and compare it to our novel approach by evaluating and analyzing the gain in efficiency concerning the testing effort. Therefore, we define our study *objective* as follows.

> Evaluation of the combination of architecture-driven model-based testing principles and regression-inspired testing strategies for efficient, yet comprehensible variability-aware conformance testing of variant-rich systems.

In order to investigate the objective, we select a sample *Body Comfort System* of a car as *unit of analysis*. We are able to use the BCS as unit of analysis based on its availability through an industrial cooperation. Thus, already existing development artifacts, e.g., system models are (re-)usable for evaluation purposes. As already mentioned in Section 1, the BCS comprises standard and optional functionality, e.g., every system includes a *Human Machine Interface*, whereas a *Central Locking System* is an optional system function. The set of possible configurations of those (optional) system functions results in more than 11,000 valid variants of the BCS. This allows us to reason about the testing effort improvements of our regression based approach compared to testing each system variant in isolation. Due to its design, our case study is *holistic*, as we study the BCS as a whole and do not have different units of analysis.

Furthermore, to investigate our study objective, we derive research questions, which we will answer by applying our novel approach to the *Body Comfort System*. We refine our objective into the following *research questions*.

- *RQ 1*: Is the delta-oriented modeling approach *applicable* to support a *consistent* specification of component and integration test models for large-scale systems under test?
- *RQ 2*: Is the delta-oriented modeling approach *applicable* to *concisely* capture and modularize commonality and variability among system variants/versions?
- *RQ 3*: Is the delta-oriented modeling approach *a sound basis* for a preplanned and precise selection of reusable test artifacts?
- *RQ 4*: Do we achieve a *gain in efficiency* from the delta-oriented testing approach compared to testing each system family member in isolation due to systematic test artifact reuse?
- *RQ 5*: Do we achieve a *gain in efficiency* from the delta-oriented testing approach compared to classical regression testing due to a precise test case classification based on explicit variability specifications rather than implicit model differencing analyses?

We answer our research questions based on the evaluation of our approach in Section 8.

Our approach is based on strategies for regression testing as well as model-based testing approaches, whereas no explicit theory is defined as frame of reference. Thus, previous studies may implicitly influence our case study design. We discuss the related work of our approach in Section 9. For data collection, we use *direct* and *independent* collection techniques to aggregate both *qualitative* and *quantitative* data. We describe the data collection process and its analysis in more detail in the following sections. As a result of our industrial cooperation, we have to take the confidentiality of existing development artifacts as well as evaluation results into account. We regulate those issues with our partner and are allowed to publish our results with a slight abstraction from the concrete system information, e.g., concrete system artifacts.

7.2. Data collection

The data collection is a decisive step to guarantee an effective evaluation of the case study and, therefore, of our novel approach. Based on the selected unit of analysis, we first reviewed existing development artifacts, e.g., the requirements specification, system models, etc. to obtain system knowledge. In addition, we interviewed system experts (1) to obtain information how the current testing processes is realized and (2) to consolidate the acquired system knowledge. The interviews were performed in an unstructured way to allow for a more open questionnaire. Summarizing, the existing documents and the interviews define the *qualitative* data we used for data analysis and evaluation.

We used the qualitative data to transform the existing system models into delta-oriented component test models and integration test models for the application of our incremental testing approach. For this, we defined a set of delta-oriented behavioral models and architecture models. We used a feedback loop with the interviewees to validate whether these models still conform to the original system behavior and system architecture, respectively. The feedback loop represents a data triangulation guaranteeing comparable results in the data analysis. Thus, we were able to compare our approach to existing testing approaches applied in the automotive industry, e.g., testing each system variant in isolation.

By applying metrics, e.g., model size, test case categorization, etc. to the existing development artifacts as well as to the data determined by applying our incremental testing approach, we also obtained *quantitative* data. We also considered this data for data analysis as described in the next section.

Summarizing, the data collection comprises the following archival data: interview results, requirements specification, system models, test artifacts, and metric-specific numbers.

7.3. Data analysis

Our data analysis aims at a hypothesis confirmation. As hypotheses, we refer to our research questions (cf. Section 7.1) and investigated whether each hypotheses was confirmed or not.

We collected the data required to apply our testing approach and sample tool chain and analyzed the result by, e.g., deriving statistical data. Based on the core models and corresponding deltas, we build each system variant consecutively, generated the corresponding test artifacts and categorized them. We categorized the test artifacts according to our approach into *new*, *reusable*, *retest* and *invalid* artifacts. Afterwards, we compared the results obtained for every system against the results of recent testing strategies used in industry, namely testing each system variant in isolation. The statistical data was then used to answer our research questions and, therefore, our hypothesis. The results obtained for every research question are presented in detail in Section 8.

7.4. Threats to validity

We already discussed some potential drawbacks of incremental component testing in our previous work (Lochau et al., 2012). For the incremental integration testing approach and the corresponding case study similar threats arise. We categorize those threats into threats to construct validity, internal validity, external validity and reliability (Verner et al., 2009; Runeson and Höst, 2009). In each category, we (1) discuss those threats related to the case study and (2) threats concerning our approach.

Threats to construct validity. In general, the interpretation of system specifications may vary among different stakeholders. In addition, available data may be missing, redundant and/or inconsistent. The same applies for the documents we reviewed as well as for the re-engineered delta-oriented test models. Therefore, we interviewed the system experts to get information and feedback before and after the document review phase and the model re-engineering phase to counteract those issues as far as possible.

We have identified no threats to the construct validity concerning our approach.

Threats to internal validity. We interviewed domain as well as system experts to get an overall impression concerning different aspects of the testing process recently conducted the automotive domain. This helped us to detect all relevant factors to be considered during evaluation.

Regarding our approach and the result obtained from its application in our case study depend on the experiences of the test engineer to define meaningful and comprehensive message sequence chart test models as test cases for component integration testing. Based on our experiences we conclude that (1) ensuring both relevant test scenarios and corner cases, as well as standard scenarios to be sufficiently covered is problematic and (2) the quality of the MSC test models of the initial system under test is particularly crucial for the remaining incremental testing campaign. This drawback exists in general when defining test models and test cases manually by a test engineer and is not specific to our approach. In addition, the integrity of our approach depends on the consistency among the three interrelated models. Here, additional constraints and appropriate tool support are required for consistency checking to support the responsible test engineer. In contrast to classical regression testing approaches applying, e.g., expensive code/model differencing (Treude et al., 2007; Kelter and Schmidt, 2008) techniques, our approach supports the exhaustive, yet precise and automatically derivable identification of reusable test artifacts among system variants. However, considering the selection of retest sets we rely on state-of-the-art change impact techniques applied for regression testing as usual.

Threats to external validity. As we focus on the problem of efficiently testing variant-rich systems as apparent in many related problem domains comprising mass customization of software systems, we assume the results of the case study to be, up to a certain extent, generalizable to other projects in similar domains. However, further evaluation and case studies must be considered to consolidate this assumption. The evaluation results presented in Section 8 indicate that our strategy increases the testing efficiency independent of the concrete system under test.

Concerning our approach, the complexity of the coverage criteria and the corresponding test goals under consideration is crucial for the quality of the defined test suites. We considered *all-connectors* which is a rather simple coverage criterion. For further experiments, more complex criteria have to be investigated. Furthermore, we limited our considerations to simple architecture models making the adaption of the delta modeling approach for architecture models, as well as the construction of regression deltas for integration testing more graspable. Applying the approach to richer architecture modeling languages requires additional efforts for consistent delta definitions and regression delta derivations. However, due to its modularity and flexibility, the delta modeling approach is, in general, applicable to arbitrary modeling languages supporting more complex structural constructs, e.g., comprising hierarchy, complex interface definitions, etc. Based on the additional constructs, the construction of the regression delta, as well as the change impact analysis for retest selection becomes more complex. Therefore, the application of our approach to state-of-the-art architecture description languages in further experiments is to be performed.

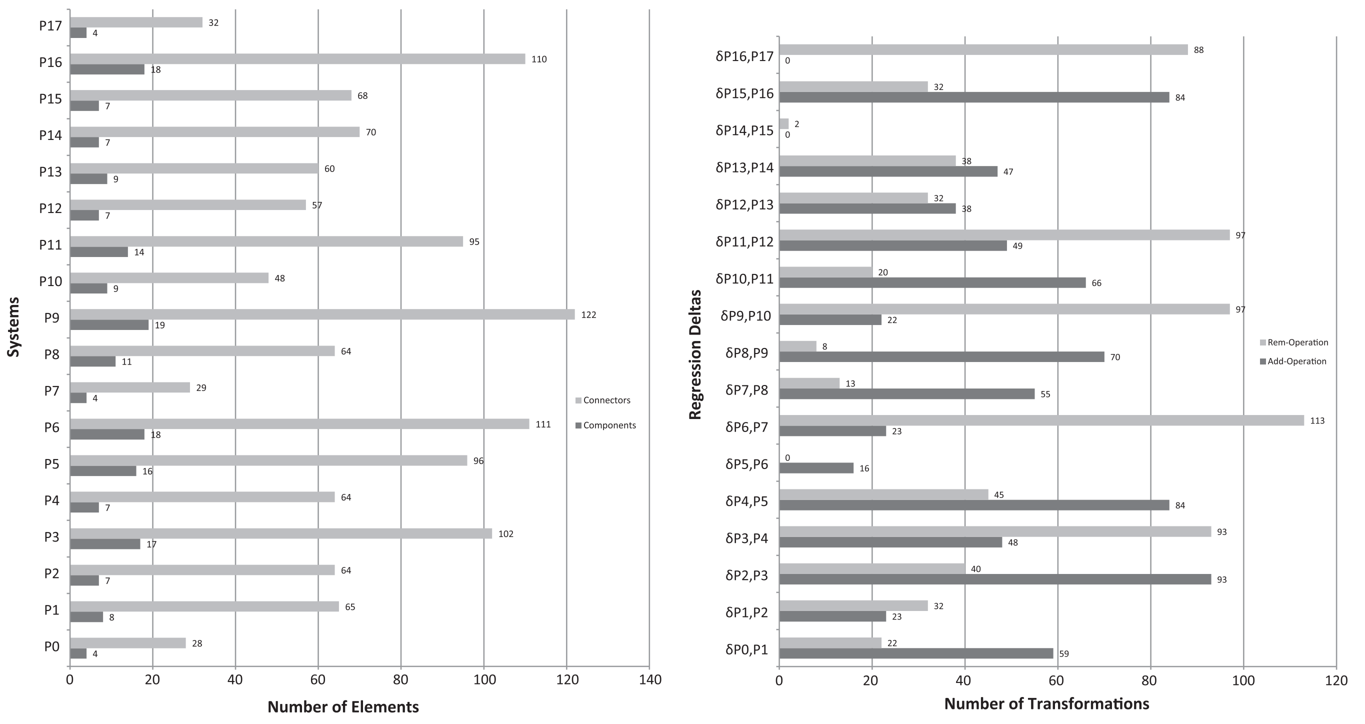
Threats to reliability. We collected and presented our data in a structured way and used standardized UML-based behavioral and structural models for test model specifications. We also described the foundations of delta-oriented test modeling and the incremental testing approach. Therefore, the data and the results should be replicable by other researchers as well. We documented the complete case study test models in Lity et al. (2012) thus being available for study replication.

The reliability of the approach and the results of the case study depend on the experience and system knowledge of the test engineer. The modeling of different MSC test models, e.g., for the initialization of the incremental testing approach may result in varying reductions of the testing effort. Again, this drawback exists in general when defining test models and test cases manually by a test engineer and is not specific to our approach.

In the following section, we present the results of the application of our incremental testing approach to the BCS case study separated according to the defined research questions.

8. Discussion of case study results

In this section, we present the results of the evaluation of our incremental model-based testing approach and sample tool chain based on the application to the BCS case study (cf. Section 7) constituting a large-scale component-based system from the automotive domain. As already mentioned, the BCS case study comprises over



(a) Overview architecture model size

(b) Overview regression delta size

Fig. 12. Overview of BCS case study delta model size and regression delta size.

11,000 system variants. Due to the commonality among the different system variants, testing each variant in isolation may result in the repetition of testing steps, e.g., for each system common test cases are to be generated and executed anew. By using a sampling strategy (Oster et al., 2011; Lochau et al., 2011) for determining a representative subset of the system variants under test based on the combinatorial selection criteria *pairwise feature coverage*, we obtained 17 representative systems under test (*P*1–*P*17). In our previous work, we evaluated the adequacy of this sampling strategy for covering the test model variants of a complete product family (Oster et al., 2011). For the BCS case study, we obtained a *model element coverage* of 95.5% for all product variants after applying the sampling for pairwise feature coverage. We further added a core system (*P*0) to the sample set as the starting point of the incremental testing process. As a criterion for choosing the core system *P*0, we require it to constitute one of the smallest possible variants, thus realizing (1) the mandatory functionality common for all system variants and (2) for alternative functionality one of the possible alternatives is arbitrary chosen. This way, we obtain an initial set of test artifacts for the core system with high reuse potentials for any subsequent variant under test.

In the following, we present the results of the evaluation addressing the research questions defined in Section 7.1. Due to the successful evaluation of the delta-oriented component testing approach already obtained in our previous work (Lochau et al., 2012; Lity et al., 2012), we limit our presentation to the evaluation results of the delta-oriented integration testing approach.

RQ 1: Application of delta-oriented modeling for consistent test model specification. For the architecture models of the different system variants, we specified one core architecture model, i.e., the architecture model of the core system *P*0 and 25 deltas transforming the core model to the architectural model variants. Therefore, we mapped a system variant directly to a set of deltas. We refer to Lity et al. (2012) for a comprehensive documentation of the BCS test models for component as well as integration testing. The size of each corresponding architecture model, i.e., the number of components and connectors is depicted in Fig. 12(a). The variant size ranges between the maximum of 141 architecture elements in system *P*9 and the minimum of 32 elements in the core variant *P*0. Here, a model size over 80 elements indicates that the corresponding system variant contains the comprehensive functionality of *Status LEDs*. An architecture variant comprises 10 components and 71 connectors on average.

Summarizing, we specified the component state machine test model variants, as well as the architecture model variants of the BCS case study as defined in Section 4 on the basis of the delta-oriented modeling approach. The system variants comprise test model variants of different sizes, thus, scalability issues arise. Our experiences have shown that the modeling approach is intuitively applicable for modeling component and integration test models and that the resulting core models and delta modules are still manageable for larger models arising in the context of large-scale systems under test.

RQ 2: Application of delta-oriented modeling for concisely capturing and modularizing commonality and variability. Based on our incremental testing approach, we derive architecture model regression deltas to explicitly capture the test model changes when stepping from a system variant P_i to the subsequent system variant P_{i+1}. The degree of test artifact reusability among different variants under test is indicated by the amount and type of delta operations in the corresponding regression delta. Fig. 12(b) gives an overview of the regression delta size resulting between the members of the representative subset of system variants under test. We define the regression delta size by the number of architecture elements affected by add and remove operations. We found a maximum of 146 delta operations between system *P*11 and *P*12, whereas the minimum of two operations arises between system *P*14 and *P*15. Here, the presence/absence of the *Status LED* functionality is one crucial factor for the size of the regression deltas. Comparing Fig. 12(a) and (b), we are able to reason about commonality among particular variants based on the operations occurring in the respective architecture model regression delta. On average,

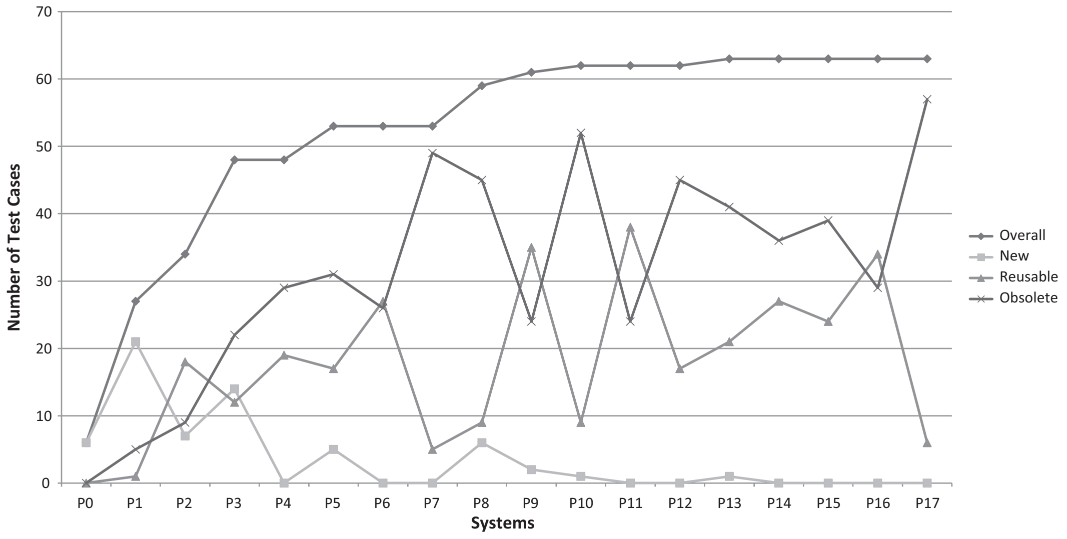


Fig. 13. Overview of test case categorization.

an architecture model regression delta comprises 46 additions and 45 removals of architecture elements.

Summarizing, the definition of state machine as well as architecture model deltas allow for an explicit, modular specification of the common and variable parts between system variants. Here, each delta comprises one specific change operation, i.e., the addition, removal or modification of a model element, where for a better modularization, deltas are encapsulated, e.g., to transform the core to a specific variant, or to reuse the transformation for a particular variable functionality.

RQ 3: Application of delta-oriented modeling for pre-planned and precise selection of reusable test artifacts. For our experiments, we considered the *all-connectors* coverage criterion, i.e., each connector in the architecture model has to be appear in at least one integration test case. About 255,000 test cases are required for covering every single system variant of the complete BCS case study when testing every system in isolation. Covering only the system variants within the sample set still requires 382 test cases.

By applying our incremental testing approach, for each system under test a categorization of its test cases is obtained summarized in Fig. 13. Diamonds denote the overall number of test cases for the 18 representative systems under test. Those test cases are divided into new test cases presented by squares, reusable test cases denoted by triangles, and obsolete test cases depicted as crosses. Apparently, there is a strong increase of the overall number of test cases when considering the first four systems under test (*P*0–*P*3) corresponding to the number of fresh test cases due to component, connector, and signal combinations emerging for the first time. By exploiting the similarity among the several system architectures and, therefore, exploiting the existing reuse potentials, we have a smaller increase from *P*4 on. This ends up in the necessity of only one new test case for *P*13 sufficient for the last seven systems (*P*11–*P*17). Obviously, the more test cases are reusable among subsequent systems, the less test cases become obsolete and vice versa.

In addition, the architecture model size potentially influences both categorizations (cf. Figs. 12(a) and 13). For smaller system variants (*P*7, *P*10 and *P*17), there are less test cases classifiable as reusable resulting in the large gap between both lines. In contrast, there are more reusable test cases than obsolete ones among larger system variants (*P*9, *P*11 and *P*16). Here, the testing progress, i.e., the number of already applied system variant testing campaigns influences the test case categorization: the more test cases our overall set of test cases comprises, the more test cases are potentially reusable for a subsequent system variant. By using our testing approach, we defined for our experiments 64 test cases in total, where we required 4 new test cases on average, reused 18 test cases and categorized 31 test cases as obsolete.

Summarizing, as described above and more detailed in Section 5, the delta-oriented modeling approach enables a systematic reuse of test artifacts. In particular, this allows for (1) specifying reusable test models for component and integration testing, and (2) the automatic derivation of regression deltas encapsulating the differences between system variants and (3) efficient reasoning about the reuse of test artifacts among those variants.

RQ 4: Achieving a gain in efficiency from systematic test artifact reuse. In Fig. 14, the results of the evaluation of our incremental testing strategy against the retest-all testing strategy are shown. Diamonds denote the number of test cases selected per system, i.e., for every system the set of reusable and new test cases is to be applied anew. In contrast, for our incremental testing approach, squares denote the number of test cases to be (re-)tested on that variant. Here, we considered a retest selection criterion based on changes to interfaces of the corresponding component variant. First, we had to (re-)test the same number of test cases for the first two systems under test in both approaches. From there on, our approach exploits reuse potentials also for test results such that we had to selected constantly a decreasing number of test cases for test case execution and, e.g., no test cases for system *P*17 since all connectors have already been covered and the interfaces have not changed such that no more retest obligations occur. We identified three factors being responsible for the reduction of the retest set, namely (1) the change impact analysis used for the specification of retest obligations, (2) the test model size, and (3) the testing progress of our incremental approach. For instance, there is a large gap between both approaches at system *P*4 as (1) we required solely existing test cases (cf. Fig. 13) for its coverage and (2) less retest obligations occur in contrast to the retest-all strategy. In contrast, the small difference between both approaches for system *P*7 results from the small test model size and the small number of reusable test cases. System *P*16 is a relatively large system variant with a high number of reusable test cases (cf. Figs. 12(a) and 13), whereas a small subset of the reusable test cases is selected for retest in contrast to the retest-all strategy. Comparing our results to those of the retest-all approach, a significant reduction of the testing effort concerning test case execution was achievable for ensuring the same degree of test model coverage. In particular, the average number of test cases to be defined and executed per system for the retest-all

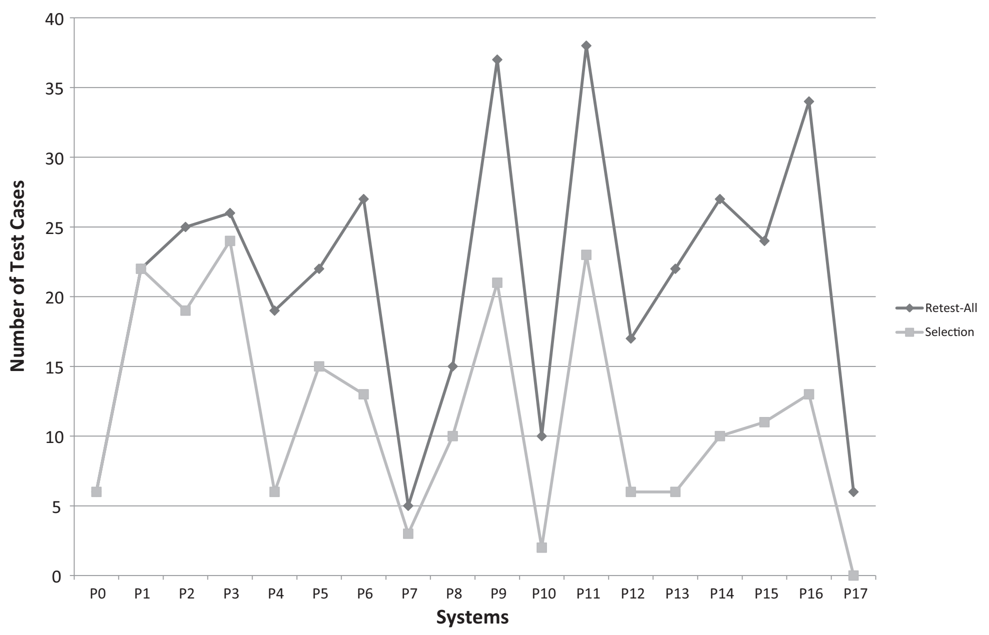


Fig. 14. Test artifact reuse evaluation results for the BCS case study.

criterion amounts to 21, whereas our incremental approach solely requires an average number of 12 test cases per system.

Summarizing, the positive results obtained in our previous work (Lochau et al., 2012; Lity et al., 2012) together with the evaluation presented here show that our incremental testing approach achieves a reduction of the testing efforts compared to testing every single system variant in isolation.

RQ 5: Achieving a gain in efficiency from explicit variability specifications. The application of the delta-oriented testing approach requires additional efforts compared to considering every variant in isolation. The entire delta-oriented test model is to be specified in advance comprising the core system and the set of deltas, whereas for the retest-all strategy all system test model variants can be modeled individually. The modularity capabilities of the delta modeling approach allows the specification of structural (architecture models) and behavioral (component state machines) common variations once for the entire family of product variants and their explicit reuse among the different variants. Thereupon, the derivation of regression deltas guarantees a structured and, therefore, a more efficient and accurate analysis of differences between system variants, whereas state-of-the-art model differencing (Treude et al., 2007; Kelter and Schmidt, 2008) approaches are costly and often imprecise. Based on the regression delta, obsolete test cases are directly identified. In contrast, we require additional efforts for the change impact analysis to select test cases for retest. We apply test model slicing (Kamischke et al., 2012) for retest analysis on the component level which has a polynomial complexity with respect to the model size (Kamischke et al., 2012). On the architectural level, we consider differences between component interfaces to determine sets of integration tests for retest on a subsequent variant. Those interface changes are likewise directly derivable from the regression delta.

Summarizing, our evaluation results show that to the additional effort imposed by the application of the delta approach is reasonable compared to the effort reductions achievable for test case executions gained from our incremental testing approach. In addition, the automated derivation of regression deltas leads to a more precise and efficient test case categorization compared to state-of-the-art regression testing approaches based on program/model differencing analysis (Treude et al., 2007; Kelter and Schmidt, 2008).

9. Related work

In this section, we present related work on variability modeling approaches, in particular for software architecture specifications, and on model-based testing approaches for variant-rich systems as well as for software architectures.

9.1. Variability modeling

Most modeling concepts for variability can be distinguished into three main approaches: annotative, compositional and transformational variability modeling (Schaefer et al., 2012). Annotative approaches use variant annotations, e.g., UML stereotypes in UML models (Ziadi et al., 2003; Gomaa, 2004) or presence conditions (Czarnecki and Pietroszek, 2006), to define which model elements belong to specific product variants. Since they contain all elements that can be contained in a possible variant, they are also called 150%-models (Schaefer et al., 2012). In the orthogonal variability model (OVM) (Pohl et al., 2005), a separate variability representation with links to the architecture model replaces direct annotations. In Loughran et al. (2008), the variability modeling language (VML) is proposed that specializes the ideas of OVM for architectural models. While annotative variability modeling allows fine-grained modifications, it relies on a monolithic product line representation.

Compositional approaches for modeling variability (Völter and Groher, 2007) capture variation by selecting specific component variants. Plastic partial components (Pérez et al., 2009) model component variability by extending partially defined components with variation points and associated variants. The Koala component model (van Ommering et al., 2000; van Ommering, 2005) is a first approach aiming at hierarchical variability modeling. In Koala, the variability of a component is described by the variability of its sub-components. The selection between different sub-component variants is realized by *switches* that are used as designated components. Hierarchical variability modeling for software product lines (Haber et al., 2011) aims at combining component variability with the component hierarchy to foster component-based development of diverse systems during architectural design. Compositional variability modeling allows a modular description of variability, but limits the impact of changes to the applied composition technique.

Transformational approaches represent variability by transformation of a base architectural model. In the common variability language (CVL) (Haugen et al., 2008), elements of the base model are substituted according to a set of pre-defined rules. In Jayaraman et al. (2007), graph transformation rules capture the variability of a single kernel model comprising the commonalities of all systems. In Hendrickson and van der Hoek (2007), architectural variability is represented by change sets containing additions, removals or modifications of components and component connections that are applied to a base line architecture. Delta modeling (Clarke et al., 2010; Schaefer et al., 2010) is a modular approach to represent system variability via transformations. A diverse set of systems is represented by a designated core system and a set of system deltas explicitly specifying changes to the core system in order to obtain other system variants. Delta modeling has been used for component-based systems in Delta-MontiArc (Haber et al., 2011, 2011).

9.2. Model-based testing of variant-rich systems

Model-based testing is well-suited for testing variant-rich systems such as software product lines (Olimpiew, 2008). Based on an annotative 150% statechart test model, Cichos et al. (2011) present an approach for coverage-driven SPL test suite generation with subsequent product subset selection for test suite execution. Also Lochau and Goltz (2010) and Lochau et al. (2011) use annotative statechart test models for the detection of feature interactions and apply this for the definition of an SPL test coverage metric. Weissleder et al. propose an approach for automatic test suite derivation based on reusable UML state machine test models and OCL expressions (Weißleder et al., 2008). Szasz and Vilanova integrate variability into UML state machines using composition operators (Szasz and Vilanova, 2008). Reuys et al. (2005) present an approach for the generation of reusable test cases on the basis of activity diagrams. Oster et al. determine a representative subset of products under test based on pairwise feature combinations and use this set for model-based SPL testing (Oster et al., 2010, 2011, 2011), whereas Perrouin et al. (2010) consider T-wise combinations. We refer the reader to Oster et al. (2010) for a summary of model-based SPL testing approaches as well as to Tevanlinna et al. (2004), Engström et al. (2008), da Mota Silveira Neto et al. (2011), and Lamancha et al. (2009) for surveys on SPL testing in general. However, no regression-based derivation and execution of test suites between product variants under test, that is presented in this work, has been introduced before.

Apart from variability in space, various model-based approaches were proposed to efficiently test different versions of a system evolving over time. Those approaches are usually based on principles and notions of regression testing (Engström et al., 2008) and aim at supporting and improving the precision and efficiency of change impact analysis and test case categorization. Farooq et al. (2010) present an approach and its tool support for regression testing of evolving software systems. They exploit the dependencies between class diagrams and state machines to analyze the impact of changes for the selection of retestable test cases. In contrast to our approach, the authors discard obsolete test cases, whereas we potentially reuse such test cases for subsequent testing campaigns. Furthermore, our approach exploits changes in the intra- and inter-component behavior for change impact analysis. In Naslavsky et al. (2009), test cases are generated from sequence diagrams and class diagrams such that during test case generation traceability links from model elements to test cases are established. When models are modified, a change impact analysis on the model is performed. According to the changes in the model, the linked test cases are classified into obsolete (if inputs have changed such that the test case is no longer executable), retestable (if they are not obsolete and traverse modified model elements) and reusable (if they are not obsolete and not retestable, hence, do not need to be re-executed). While this classification is similar to our approach presented in this paper, we do not rely on explicit traceability links between test cases and model elements. Briand et al. (2009) present a comprehensive methodology to categorize test cases defined as message sequences charts to support safe regression testing for new system versions. In contrast to our delta-oriented approach, where changes are explicitly encapsulated as part of a variable test model, their approach is based on formal mappings between UML design models and test cases which allows to trace system changes to compare different versions and the resulting change impact on existing test cases.

9.3. Model-based testing of software architectures

When testing variable software architectures, such as product line architectures, there are some challenges and opportunities due to the inherent system variability (Muccini and Hoek, 2003). Muccini and van der Hoek discuss those challenges and opportunities with respect to existing single-system testing techniques and describe how component testing, integration testing, conformance testing, and regression testing can be adapted to product line architecture testing (Muccini and Hoek, 2003). In the context of evolving systems, Harrold (1998) analyzes the potentials of regression test selection techniques and regression testability on the basis of formal specifications of software architectures. Bertolino et al. (2000, 2003) present an approach for test plan derivation for conformance integration testing based on labeled transition systems (LTS). They specify the dynamics of a software architecture by an adaptation of labeled transition systems focusing on relevant behaviors for test case generation. Muccini et al. (2005, 2006) adapt this approach for regression testing of evolving systems, where retest obligations arise due to the differences among the behavioral architecture specifications. For the behavioral architecture specifications, they compose the component LTS, whereas our approach uses a set of MSCs for the behavioral specification. da Mota Silveira Neto et al. (2010) introduce a regression testing framework for product line architectures, where regression testing decisions are performed on the basis of architectural similarities between product variants. In contrast to our approach, they exploit the knowledge about architecture source code and further remove obsolete test cases instead of potentially reusing them in subsequently testing campaigns.

10. Conclusion

Software architecture specifications allow coping with the complexity of variant-rich large-scale software systems by a hierarchical decomposition of the system into smaller components. In order to facilitate efficient quality assurance of variant-rich software systems, in this article, we have proposed to combine architectural-driven model-based testing principles and regression-based testing strategies. We have introduced delta-oriented architecture test modeling to achieve the systematic reuse of common component and integration test artifacts between different system variants. Second, we have applied delta-oriented test artifact reuse and regression test planning for a systematic evolution of variable test artifacts among incrementally tested versions and/or variants of a software system.

In future work, we will extend our approach to support further integration and component test coverage and retest selection criteria. In addition, we plan to extend the delta-oriented architecture testing approach to real-world ADL along the lines of the existing integration of delta modeling into Matlab/Simulink (Haber et al., 2013). We will investigate further criteria for supporting the selection of the core system and investigate the impact of choosing different core systems as well as different orderings for testing the subsequent variants on the efficiency of our proposed testing approach. For this, we plan to reformulate the sample selection problem as a multi-objective optimization problem comprising estimated testing costs and benefits values for variants under test. Thereupon, instead of using a sample-based preselection of representative variants under test, we plan to define criteria for a suitable ordering for testing the different system variants in order to (1) increase reuse potentials between subsequent variants under test and (2) to obtain an adequate test coverage of test model elements in any possible variant by means of delta-oriented coverage criteria. Based on the prototypical implementation, we are planning to evaluate our approach using more complex practical case examples.

References

Agrawal, H., Horgan, J.R., Krauser, E.W., London, S., 1993. Incremental Regression Testing. In: Proceedings of the Conference on Software Maintenance, ICSM '93, Washington, DC, USA. IEEE Computer Society, pp. 348–357.

Bertolino, A., Corradini, F., Inverardi, P., Muccini, H., 2000. Deriving test plans from architectural descriptions. In: Proceedings of the 2000 International Conference on Software Engineering, pp. 220–229.

Bertolino, A., Inverardi, P., Muccini, H.,2003. Formal Methods in Testing Software Architectures. In: SFM. Springer, pp. 122–147.

Briand, L.C., Labiche, Y., He, S., 2009. Automating regression test selection based on UML designs. Information and Software Technology 51 (1), 16–30.

Briand, L., Labiche, Y., Liu, Y., 2012. Combining UML sequence and state machine diagrams for data-flow based integration testing. In: Proceedings of the 8th European conference on Modelling Foundations and Applications, ECMFA'12, pp. 74–89.

Cichos, H., Oster, S., Lochau, M., Schürr, A., 2011. Model-based coverage-driven test suite generation for software product lines. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, MODELS'11, pp. 425–439.

Clarke, D., Helvensteijn, M., Schaefer, I.,2010. Abstract delta modeling. In: GPCE. Springer.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison Wesley Longman.

Czarnecki, K., Pietroszek, K., 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints.

da Mota Silveira Neto, P.A., do Carmo Machado, I., Cavalcanti, Y.C., de Almeida, E.S., Garcia, V.C., de Lemos Meira, S.R., 2010. A regression testing approach for software product lines architectures. In: SBCARS'10, pp. 41–50.

da Mota Silveira Neto, P.A., Carmo Machado, I.d., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R., 2011. A systematic mapping study of software product lines testing. Information and Software Technology 53 (5), 407–423.

De Nicola, R., 1987. Extensional equivalence for transition systems. Acta Informatica 24, 211–237.

Engström, E., Skoglund, M., Runeson, P., 2008. Empirical evaluations of regression test selection techniques. In: Proc. of ESEM'2008, pp. 22–31.

Farooq, Q.-u.-a., Iqbal, M.Z.Z., Malik, Z.I., Riebisch, M.,2010. A model-based regression testing approach for evolving software systems with flexible tool support. In: Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2010, ECBS'10. IEEE Computer Society, Washington, DC, USA, pp. 41–49.

Fraser, G., Wotawa, F., Ammann, P., 2009. Testing with model checkers: a survey, software testing. Verification and Reliability 19 (3), 215–261.

Genest, B., Muscholl, A., 2005. Message sequence charts: a survey. In: Fifth International Conference on Application of Concurrency to System Design, 2005 (ACSD 2005), pp. 2–4.

Gomaa, H., 2004. Designing Software Product Lines with UML. Addison Wesley.

Gupta, R., Jean, M., Mary, H., Soffa, L.,1992. An approach to regression testing using slicing. In: Proceedings of the Conference on Software Maintenance. IEEE Computer Society Press, pp. 299–308.

Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.,2011. Delta-oriented architectural variability using MontiCore. In: ECSA'11 5th European Conference on Software Architecture: Companion Volume. ACM, New York, NY, USA, pp. 6:1–6:10.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.,2011. Hierarchical variability modeling for software architectures. In: Proceedings of International Software Product Lines Conference (SPLC 2011). IEEE Computer Society.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., 2011. Delta modeling for software architectures. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, fortiss GmbH München.

Haber, A., Kolassa, C., Manhart, P., Nazari, P.M.S., Rumpe, B., Schaefer, I., 2013. First-class variability modeling in Matlab/Simulink. In: VaMoS.

Harrold, M.J., 1998. Architecture-based regression testing of evolving systems. In: Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA 1998), pp. 73–77.

Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A., 2008. Adding Standardized Variability to Domain Specific Languages. In: SPLC.

Hendrickson, S.A., van der Hoek, A., 2007. Modeling product line architectures through change sets and relationships. In: ICSE.

ITU, 1999. Recommendation Z.120: Message Sequence Chart (MSC). Haugen (ed.), Geneva.

Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H., 2007. Model composition in product lines and feature interaction detection using critical pair analysis. In: MoDELS.

Kamischke, J., Lochau, M., Baller, H.,2012. Conditioned model slicing of feature-annotated state machines. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development, FOSD'12. ACM, New York, NY, USA, pp. 9–16.

Kelter, U., Schmidt, M.,2008. Comparing state machines. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models, CVSM'08. ACM, New York, NY, USA, pp. 1–6.

Lamancha, B.P., Usaola, M.P., Velthius, M.P., 2009. Software product line testing – a systematic review. In: Shishkov, B., Cordeiro, J., Ranchordas, A. (Eds.), ICSOFT (1). INSTICC Press, pp. 23–30.

Lity, S., Lochau, M., Schaefer, I., Goltz, U., 2012. Delta-oriented model-based SPL regression testing. In: 3rd International Workshop on Product Line Approaches in Software Engineering, 2012 (PLEASE), pp. 53–56.

Lity, S., Lachmann, R., Lochau, M., Schaefer, I., 2012. Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study. Tech. Rep. 2012-07, Technische Universität Braunschweig.

Lochau, M., Goltz, U., 2010. Feature interaction aware test case generation for embedded control systems. Electronic Notes in Theoretical Computer Science 264, 37–52.

Lochau, M., Oster, S., Goltz, U., Schürr, A., 2011. Model-based pairwise testing for feature interaction coverage in software product line engineering. Software Quality Journal, 1–38.

Lochau, M., Schaefer, I., Kamischke, J., Lity, S., 2012. Incremental model-based testing of delta-oriented software product lines. In: Brucker, A., Julliand, J. (Eds.), Tests and Proofs, Vol. 7305 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 67–82.

Loughran, N., Sánchez, P., Garcia, A., Fuentes, L., 2008. Language support for managing variability in architectural models. In: Software Composition, Vol. 4954 of Lecture Notes in Computer Science.

Müller, T.C., Lochau, M., Detering, S., Saust, F., Garbers, H., Märtin, L., Form, T., Goltz, U., 2009. A comprehensive description of a model-based, Continuous Development Process for AUTOSAR Systems with Integrated Quality Assurance. Tech. Rep. 2009-06, TU Braunschweig.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.

Muccini, H., Hoek, A.V.D., 2003. Towards testing product line architectures. In: International Workshop on Testing and Analysis of Component Based Systems, pp. 111–121.

Muccini, H., Dias, M.S., Richardson, D.J.,2005. Towards software architecture-based regression testing. In: Proceedings of the Workshop on Architecting Dependable Systems, 2005, WADS'05. ACM, New York, NY, USA, pp. 1–7.

Muccini, H., Dias, M.S., Richardson, D.J., 2006. Software architecture-based regression testing. JSS – Special Edition on Architecting Dependable Systems 33.

Naslavsky, L., Ziv, H., Richardson, D., 2009. A model-based regression test selection technique. In: IEEE International Conference on Software Maintenance, 2009 (ICSM 2009), pp. 515–518.

Olimpiew, E.M., 2008. Model-Based Testing for Software Product Lines. George Mason University (Ph.D. thesis).

Oster, S., Markert, F., Ritter, P., 2010. Automated incremental pairwise testing of software product lines. In: SPLC'2010, pp. 196–210.

Oster, S., Wübbeke, A., Engels, G., Schürr, A., 2010. Model-based software product lines testing survey. In: Model-based Testing for Embedded Systems.

Oster, S., Lochau, M., Zink, M., Grechanik, M., 2011. Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations. In: FOSD'11.

Oster, S., Zink, M., Lochau, M., Grechanik, M.,2011. Pairwise feature-interaction testing for SPLs: potentials and limitations. In: Proceedings of the 15th International Software Product Line Conference, Vol. 2, SPLC'11. ACM, New York, NY, USA, pp. 6:1–6:8.

Oster, S., Zorcic, I., Markert, F., Lochau, M., 2011. MoSo-PoLiTe – tool support for pairwise and model-based software product line testing. In: VAMOS'11.

Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J., 2009. Plastic partial components: a solution to support variability in architectural components. In: WICSA/ECSA.

Perrouin, G., Sen, S., Klein, J., Le Traon, B., 2010. Automated and scalable *T*-wise test case generation strategies for software product lines. In: ICST'2010, pp. 459–468.

Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg.

Reuys, A., Kamsties, E., Pohl, K., Reis, S., 2005. Model-based system testing of software product families. In: CAiSE, pp. 519–534.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14 (2), 131–164.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.,2010. Delta-oriented programming of software product lines. In: SPLC. Springer.

Schaefer, I., Bettini, L., Damiani, F.,2011. Compositional type-checking for delta-oriented programming. In: Intl. Conference on Aspect-oriented Software Development (AOSD'11). ACM Press.

Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K., 2012. Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer 14 (5), 477–495.
Schaefer, I., 2010. Variability Modelling for Model-Driven Development of Software Product Lines. In: VaMoS, pp. 85–92.
Szasz, N., Vilanova, P., 2008. Statecharts and variabilities. In: VAMOS'08, pp. 131–140.
Szyperski, C., 2002. Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Tevanlinna, A., Taina, J., Kauppinen, R., 2004. Product family testing: a survey. ACM SIGSOFT Software Engineering Notes 29, 12–18.
Treude, C., Berlik, S., Wenzel, S., Kelter, U.,2007. Difference computation of large models. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE'07. ACM, New York, NY, USA, pp. 295–304.
Utting, M., Legeard, B., 2007. Practical Model-Based Testing. A Tools Approach. M. Kaufmann.
Völter, M., Groher, I., 2007. Product line implementation using aspect-oriented and model-driven software development. In: SPLC.
van Ommering, R., van der Linden, F., Kramer, J., Magee, J., 2000. The koala component model for consumer electronics software. Computer 33 (3), 78–85.
van Ommering, R., 2005. Software reuse in product populations. IEEE Transactions on Software Engineering 31 (7), 537–550.
Verner, J., Sampson, J., Tosic, V., Bakar, N., Kitchenham, B., 2009. Guidelines for industrially-based multiple case studies in software engineering. In: Third International Conference on Research Challenges in Information Science, 2009 (RCIS 2009), pp. 313–324.
Weißleder, S., Sokenou, D., Schlingloff, H., 2008. Reusing state machines for automatic test generation in product lines. In: MoTiP'2008.
Ziadi, T., Hélouët, L., Jézéquel, J.-M., 2003. Towards a UML profile for software product lines. In: Workshop on Product Family Engineering (PFE).

Malte Lochau has a postdoc position at the Real-Time Systems Lab of Prof. Andy Schürr at the TU Darmstadt. His research interests are software product line engineering, model-based testing and formal semantics. His research is part of the DFG project IMoTEP and in the DFG SFB 1053 MAKI.

Sascha Lity is a Ph.D. Student at the Technische Universität Braunschweig. He finished his Master of Science in 2011 and works as research assistant at the Institute for Programming and Reactive Systems since 2012. Main interests of his research are evolving software product lines and model-based testing of variant-rich systems. He is a member of the DFG project IMoTEP.

Remo Lachmann is a Ph.D. Student at the Technische Universität Braunschweig. He finished his Master of Science in 2012 and works as research assistant at the Institute of Software Engineering and Automotive Informatics since November 2012. His research interests focus on testing variant-rich systems as well as testing driver assistance systems.

Ina Schaefer is chair of the Institute of Software Engineering and Automotive Informatics at the Technischen Universität Braunschweig. She received their Ph.D. degree from the TU Kaiserslautern and worked as a postdoc at the Chalmers University of Technology in Gothenburg, Sweden. Her research interests are verification and testing methods for variant-rich and evolving software systems.

Ursula Goltz studied Computer Science at the Technical University of Aachen and graduated there with a diploma degree 1982. She received their Ph.D. degree from the TU Aachen in the year 1988. She worked as a scientific assistant at the TU Aachen from 1982–1985 and at the Institute on Methodological Foundations of GMD, St. Augustin from 1986–1992. After teaching activities at the Universities of Munich, Erlangen-Nürnberg, Mannheim and Bonn, she became professor for Programming at the University of Hildesheim in the year 1992. Since 1998 she is professor for Computer Science at the Technische Universität Braunschweig. There she is chair of the Institute for Programming and Reactive Systems. Her main research interests are specification and system design, reactive systems, concurrency, process algebras and semantics.