Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 264 (2010) 37–52
www.elsevier.com/locate/entcs

# Feature Interaction Aware Test Case Generation for Embedded Control Systems

Malte Lochau, Ursula Goltz[1,2,3]

*Institute for Programming and Reactive Systems*
*TU Braunschweig*
*Germany*

**Abstract**

The growing number and increased coupling of functionality in embedded control systems, e.g. in the automotive domain, leads to complex networks of interacting features and a wide range of variants. Hence, today's software development processes must include systematic approaches to analyze the functional correctness of system specifications and implementations. Model-based testing is of particular importance for embedded software systems as the test cases can be performed at the real system under test and failures arising from the interaction of software and hardware can be discovered. As features are usually designed in a modular and isolated way, unexpected and undesired behavior caused by unintended interferences of insufficiently synchronized and even contradicting features concurrently active in the system often remains undetected which may lead to serious safety problems. To overcome such feature interactions at system integration level is tedious as it leads to a voluminous number of unmanageable test cases. We describe a model-based approach for efficiently generating test cases that particularly aim at feature interaction analysis. We first characterize feature interaction in a formal way based on a rigorous functional architecture model, and describe how to detect potential feature interactions. For test case generation, behavioral models like Statecharts and according coverage criteria can be used as usual, but only those models are integrated into the test model, that contribute to one of the features under consideration. This leads to a trade off between comprehensive test coverage to find possible flaws caused by interacting features, and yet still a reasonable number of test cases. The steps of the approach are illustrated be means of a case study from the automotive domain.

*Keywords:* Model-based Specification and Testing, Test Sequence Generation Methods, Feature Interaction, Embedded Control Systems, Automotive Domain.

## 1 Introduction

### *1.1 Motivation*

Embedded software systems for controlling and monitoring mechanical and electrical processes become more and more ubiquitous. In the automotive domain, software is already the crucial part of electronic control unit networks [4]. The

[1] Thanks to all authors of [13] for contributing to the case study used in this paper
[2] Email: lochau@ips.cs.tu-bs.de
[3] Email: goltz@ips.cs.tu-bs.de


doi:10.1016/j.entcs.2010.12.013

set of features, i.e. complex functionality recognizable by the driver, is realized by cooperating software functions with shared access to hardware devices such as sensors and actuators. Various interacting components are to be integrated building reactive systems with high inner complexity and intensive interactions with the environment. In general, in the automotive domain the set of features is fixed and the data elements and signals used at runtime are known when designing the system. Therefore, rigorous modeling approaches for the functional architecture and the components' internal behavior are a crucial foundation to handle the complexity of interfering features in their different operating modes [3]. Especially in the context of safety critical systems, formal analysis methods are crucial for ensuring the functional correctness of the system specification and implementation. Besides verification techniques such as model checking, model-based testing is essential for embedded control system validation as test cases can be applied to the real system under test considering the effects of the software system when interacting with the hardware of the target platform.

In this paper, we describe a model-based approach for test case generation for embedded control systems that explicitly takes feature interactions into account. The approach aims at test case generation for systematic detection and analysis of potentially undesired behavior resulting from interference of "orthogonal" features sharing some system artifact. Compared to test suites necessary for considering the full integration of all features in the system, the number of test cases needed can be reduced significantly. Based on a functional architecture specification we adopt Statechart-like behavioral models, i.e. STATEFLOW automata for test case generation. The approach is illustrated by means of a case study from the automotive domain.

### *1.2 Model-based Development and Test Case Generation*

Model driven software engineering becomes more and more important to cope with the level of complexity of todays software projects. Modeling languages like those bundled in the UML (Unified Modeling Language) are used to represent the relevant properties, both structural and behavioral, of the system under development. Considering the development steps of embedded software systems, modeling formalisms with different levels of abstraction are applied providing views on essential artifacts on the system, e.g. for feature / requirements specifications, designing the functional architecture and the logical behavior of the functions, and finally implementation specific configurations for code generation. In the automotive domain model-based, tool supported development practices for controlling systems are prevailing and found their way into standards like AUTOSAR [1].

In addition, model-based formal analysis methods can be applied to ensure functional correctness, either by formal verification, e.g. model checking, or by model-based testing. Testing approaches in general are not capable of ensuring the absence of failures, i.e. proving the functional correctness. The application of systematic tests aims at finding failures in the implementation by investigating whether the system outputs differ from those predicted by the test case, i.e. for a particular use

case. The major benefit of testing is that test cases generated automatically from behavioral models can be applied to the real system under test by means of executable and repeatable scenarios. Model-based testing pursues the automation of test design, i.e. test case selection and generation from a model of the system under test. The number and complexity of test cases can be controlled by parameterizing the algorithm, e.g. stating which parts of the system are to be tested. Coverage criteria determine the granularity of the test case selection procedure, thus allow for reliability assumptions according to degree of coverage chosen.

### *1.3 Feature Interaction*

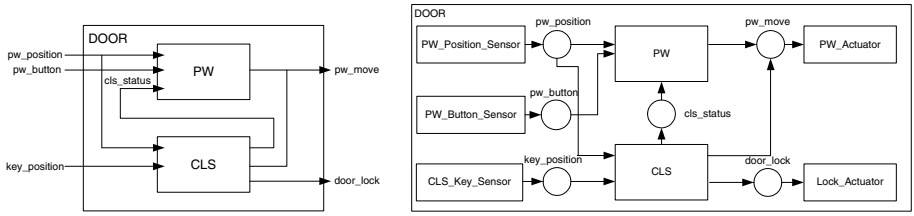
The growing number and coupling of functionality in automotive control systems leads to a complex network of interacting features realizing customers' requirements. Furthermore, the multitude of variants concerning possible feature combinations requires for a flexible integration and removing of features. Nevertheless, today's common development, testing, and integration practices still consider each individual feature as isolated modules, thus neglecting potential interactions of features within the system and with the environment. As a consequence, unexpected and undesired behavior caused by unintended combinations of insufficiently synchronized features from different parts of the system remain undetected. At worst, such *feature interactions* [7,12,10] can lead to serious safety problems, e.g. when ABS interferes with ESP (Electronic Stability Program) thus obstructing the slow-down process [4]. A similar phenomenon arises in case of component reuse when a particular feature interaction is required but missing, e.g. when safety critical feature fail to put a *veto* on the activation of another feature that would threat its safe execution.

## 2 Preliminaries

For the following discussions we consider a simplified formal system specification model. The function oriented, component-based approach is inspired by state-of-the-art engineering methodologies for architectures and behavioral specifications of software intensive embedded control systems as proposed for instance by the methodology of the AUTOSAR standard [1] for the automotive domain.

### *2.1 Functional Architecture*

We define a *functional architecture* to be a quadruple $\mathcal{F} = \langle\, \mathcal{C}, \mathcal{S}, \mathcal{A}, \mathcal{V} \,\rangle$ consisting of the logical building blocks, i.e. the set of *components* $\mathcal{C}$ of the software system. The interface of a component $C \in \mathcal{C}$ defines the input values consumed by the processes as parameters, and the output values produced as the results of their internal functions' computations. We use this abstract notion of (shared) *values* to reason about potential data/control dependencies between components omitting the actual communication paradigm used. Values $V \in \mathcal{V}$ can therefore be interpreted as signals, shared variables, etc. *Sensor* components $\mathcal{S} \subseteq \mathcal{C}$ and *actuator* components $\mathcal{A} \subseteq \mathcal{C}$ mimic the role of sensor and actuator hardware devices at the functional level,




(a) Connector View (b) Value View

Fig. 1. Door System – Functional Architecture

i.e. sources of sensor values or sinks for values controlling actuators, respectively. As we omit hierarchical component structures, i.e. *compositions*, each $C \in \mathcal{C}$ is assumed to be *atomic* and can be defined as $C = \langle\, \mathcal{I}, \mathcal{O}, R \,\rangle$, where $\mathcal{I} \subseteq \mathcal{V}$ denotes the *input* values read by the component, and $\mathcal{O} \subseteq \mathcal{V}$ denotes the *output* values changed by the component. Thus, $\mathcal{I} \cup \mathcal{O}$ states the *Interface* values of $C$. The *Runnable* $R$ refers to the behavioral specification of the internal processes implementing the set of *functions* realizing the component's functionality. In case of $C \in \mathcal{S}$, $\mathcal{I} = \emptyset$ holds and $\mathcal{O} = \{O_S\}$ is the measured value delivered by the sensor. For $C \in \mathcal{A}$, $\mathcal{O} = \emptyset$, holds accordingly, and $\mathcal{I} = \{I_S\}$ is the value affecting the actuator. The internal behavior specification $R$ of sensors/actuators can be ignored. In component-based architecture specifications the notion of *connectors* is introduced to indicate a dependency between components denoted as *uses*-relations. For components $C_1 \in \mathcal{C}$, and $C_2 \in \mathcal{C}$ in a functional architecture $\mathcal{F}$ we have connectors leading from output values $V \in \mathcal{O}_1$ of $C_1$ to input values $V' \in \mathcal{I}_2$ of $C_2$, iff $V = V'$, hence for each $V \in \mathcal{O}_1 \cap \mathcal{I}_2$.

As a running example, we refer to a simplified case study *car door controlling system* in the following (see [13] for details). The functional architecture $\mathcal{F}_{DS}$ shown in Fig. 1 consists of controlling components $PW \in \mathcal{C}$ for a power window and $CLS \in \mathcal{C}$ for a central locking system. The representation in 1(a) uses *connectors* (arrows) to illustrate the data flow between components and the environment, against what in 1(b) the *values* (circles) and the components' accesses to them are made explicit. Sensor component $PW\_Position\_Sensor \in \mathcal{S}$ measures the vertical position of the window, $PW\_Button\_Sensor \in \mathcal{S}$ delivers the status of the power window button, and $CLS\_Key\_Sensor \in \mathcal{S}$ provides the position of the key. The actuator $PW\_Actuator \in \mathcal{A}$ moves the power window up and down, and $Lock\_Actuator \in \mathcal{A}$ locks and unlocks the door. Values in $\mathcal{V}$ are used to pass control data between the components. The power window button can be either pressed "up" or "down" forcing the power window actuator to move the window to the desired direction, until (a) the button is released, or (b) the window has reached the highest/lowest position indicated by the position sensor. The door can be locked and unlocked by turning the key accordingly. Turning the key to the lock position twice in a row activates the central locking, hence not only locking the door, but also moving the window to the highest position.

### 2.2 Behavioral Specification

Components $C \in \mathcal{C}$ of the functional architecture host processes by means of software artifacts realizing atomic functions of the system. We assume that the specification of these runnables $R$ of $C$ are given as hierarchical state machines, e.g. STATEFLOW automata. The STATEFLOW formalism [8] is a Statechart-like approach [9] that is widespread in embedded systems engineering as it is part of the MATLAB/Simulink tool set. Here, we consider a basic representation of STATEFLOW models with the most common features. A basic STATEFLOW model for a behavioral specification $R$ is given as $SF_R = \langle\, \mathcal{S}, \mathcal{T}, sub, S_0 \,\rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{T}$ is the set of transitions, $sub$ is the sub state relation, and $S_0 \subseteq \mathcal{S}$ is the set of default states. We omit explicit final states at this point. Besides basic states, STATEFLOW automata provide composite states, namely XOR states leading to hierarchical scopes of states, and orthogonal AND states introducing concurrent sub machines. Composite states $S \in \mathcal{S}$ contain direct sub states $S' \in \mathcal{S}$ that are related by the sub state relation $sub \subset \mathcal{S} \times \mathcal{S}$, thus $(S, S') \in sub$. Hence, $sub$ defines a tree hierarchy of composite states with basic states as leaves. The direct sub states of an XOR state are related by exclusive or, and the direct sub states of an AND state are themselves XOR states containing the orthogonal *regions* of the AND state. The outermost state of a STATEFLOW automaton, thus the root of the state hierarchy is an XOR state. For each XOR state $S$, there is exactly one direct sub state $S' \in S_0$ being the *default* state of $S$. Transitions $S \xrightarrow{E[C]/A} S' \in \mathcal{T}$ leading from a source state $S \in \mathcal{S}$ to a destination state $S' \in \mathcal{S}$ are labeled with complex *ECA rules* defining the behavior and communication in the system via shared values $\mathcal{V}$, where each of the three parts is optional. The E-part lists *events* whose occurrence in the system triggers the transition, and the C-part consists of *conditions*, i.e. guards to be satisfied for the transition to fire. As events, we assume *changes* to some value in the system, and conditions are defined by means of *expressions* over values. The A-part states actions to be performed when the transition is taken, i.e. value *reassignments*. The access and usage of values in ECA rules must match the interface specification of the surrounding component: (1) for values $v \in \mathcal{V}$ read in the E- or C-part, as well as in RHS of assignments in the A-part, $v \in \mathcal{I}$ must hold, and (2) for values $v \in \mathcal{V}$ in LHS of assignments, $v \in \mathcal{O}$ must hold, accordingly.

Again, consider the sample *door system*: The controlling components $PW$ and $CLS$ contain runnables $R_{PW}$ and $R_{CLS}$, each consisting of a single STATEFLOW model $SF_{PW}$ and $SF_{CLS}$ as shown in Fig. 2 and 3. Transitions are triggered by *conditions* over input values and react by changing output values of the components' interfaces as defined in Fig. 1. The three states of the power window denote the position of the window, i.e. up (closed), down (open), and pending (in between), which is represented by the values $pw\_position \in \{1, -1, 0\}$, and the same for $pw\_button$ and $pw\_move$, where 0 means "hold", accordingly. Depending on the button status and window position, the window is moved to the intended position by setting the actuator value $pw\_move$ until it has reached the uppermost/lowermost position. Being initially closed, the window can only be opened if the central locking is inactive, i.e. $cls\_status == 0$. The central locking reacts on the $key\_position$

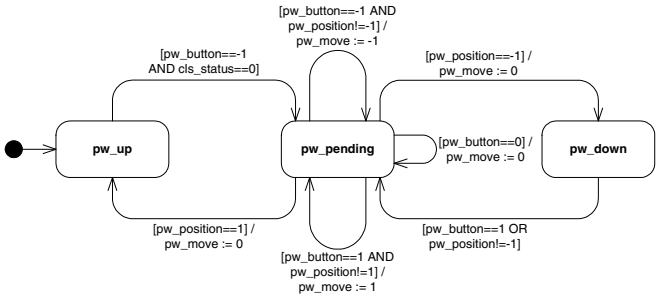



Fig. 2. Door System – Stateflow of Power Window

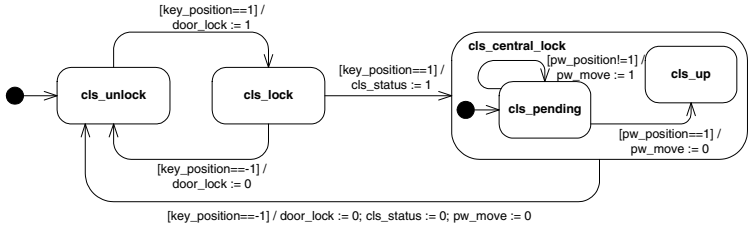



Fig. 3. Door System – Stateflow of Central Locking System

values 1 for "lock", −1 for "unlock" and affects the *door_lock* actuator value accordingly. When locked twice the central locking becomes active moving the window to the uppermost position and setting the status flag *cls_status* for notification of the power window feature.

When executed, the current state of STATEFLOW automaton $SF_R$ is a *configuration* $\sigma = \langle\ \mathcal{S}_\sigma, \nu\ \rangle$, where $\mathcal{S}_\sigma \subseteq \mathcal{S}$ is the set of active states, and $\nu : \mathcal{V} \to \mathcal{D}$ is an *interpretation function* mapping the current *values* to elements of some generic *domain* $\mathcal{D}$. A *step* of execution leads from one configuration $\sigma_k = \langle\ \mathcal{S}_{\sigma_k}, \nu_k\ \rangle$ to a subsequent configuration $\sigma_{k+1} = \langle\ \mathcal{S}_{\sigma_{k+1}}, \nu_{k+1}\ \rangle$ as a response to the stimuli in the system according to the operational semantics chosen. For comprehensive discussions, e.g. of resolving conflicting transition activations and non-determinism, we refer to [16]. A *run* of $SF_R$ is then defined to be (possibly infinite) sequence of steps $(\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots)$. We assume, that several processes within a runnable $R$ of a component $C$ are integrated in one state machine specification $SF_R$ by *orthogonal composition* at the outermost hierarchy level:

$$SF_R = SF_{R1} \oplus SF_{R2} = \langle\ \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_c, \mathcal{T}_1 \cup \mathcal{T}_2, sub', S_{0,1} \cup S_{0,2} \cup S_{0,c}\ \rangle$$

where $\mathcal{S}_C = \{S_{root}, S_{reg}\}$ adds a new XOR root state $S_{root}$ to $SF_R$ containing an AND state $S_{reg}$ whose regions are the former root states of $SF_{R1}$ and $SF_{R2}$. Relation *sub* is adapted accordingly to $sub'$ and the new root state $S_{0,c}$ is added to the set of default states of $SF_{R1}$ and $SF_{R2}$. Assuming that $SF_{R1}$ is located in a component $C_1$, and $SF_2$ in a component $C_2$, the component $C_R$ hosting $SF_R = SF_{R1} \oplus SF_{R2}$ can be constructed as:

$$C_R = C_1 \oplus C_2 = \langle\ \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, R\ \rangle$$

where $R$ refers to $SF_R$. Note that this basic construction can be refined depending on the interaction concepts used for implementation. For example, values that are

solely accessed by $C_1$ and $C_2$ can be removed from the interface values of $C_R$, e.g. constituting local variables $\mathcal{V}_R \subseteq \mathcal{V}$ hidden in $C_R$. In this case, $\mathcal{I}_R = \mathcal{I}_1 \cup \mathcal{I}_2 \setminus \mathcal{V}_R$, and $\mathcal{O}_R = \mathcal{O}_1 \cup \mathcal{O}_2 \setminus \mathcal{V}_R$, accordingly.

For instance, when integrating the power window component and central locking component into a door system composition $C_{DS}$, the value $cls\_status \in \mathcal{O}_{CLS} \cap \mathcal{I}_{PW}$ can be removed from the interface of $C_{DS}$ thus becoming a local value of $C_{DS}$.

The basic STATEFLOW constructs introduced are sufficient for behavioral modeling of control systems, but can as well be enriched as usual, see e.g. [9,16].

### *2.3 Feature Network*

A feature is realized by a chain of related system artifacts that cooperate in some way in order to affect the *environment* in a way that is recognizable by the *user*. The environment is supposed to be either a hardware component of the system, or it is located outside of the system and therefore affected indirectly. A *feature chain* originates from the environment whose certain properties are recognized by means of sensor data. The data processing for calculating a system reaction according to the controlling tasks to be realized is done by a collection of interacting functions, finally orchestrating some actuators to cause the desired impact on the environment. Typically, features of embedded control systems realize different kinds of feedback control loops. Formally, a *feature* can be defined as $f = \langle\, \mathcal{S}_f, \mathcal{R}_f, \mathcal{A}_f \,\rangle$, where $\mathcal{S}_f \subseteq \mathcal{S}$ are the sensors $S \in \mathcal{S}_f$ delivering values $O_S$, i.e. stimuli and measured data relevant for the feature, $R \in \mathcal{R}_f$ are *Runnables*, i.e. interacting functions that implement $f$, and $\mathcal{A}_f \subseteq \mathcal{A}$ are the actuators $A \in \mathcal{A}_f$ to be influenced by the feature by adjusting $I_A$ in order to obtain the desired behavior. The interconnections between these artifacts results from the connectors of their components leading to a network of interrelated features.

The sample *door system* consists of two features: the power window and the central locking system:

$$\begin{aligned} f_{PW} = \langle\ & \{PW\_Position\_Sensor, PW\_Button\_Sensor\}, R_{PW}, \\ & \{PW\_Actuator\}\ \rangle \\ f_{CLS} = \langle\ & \{PW\_Position\_Sensor, CLS\_Key\_Sensor\}, R_{CLS}, \\ & \{PW\_Actuator, Lock\_Actuator\}\ \rangle \end{aligned}$$

### *2.4 Feature Interaction Detection*

In general, two or more features potentially interact with each other if they access at least one shared system artifact. Depending on the abstraction level, such *points of interaction* can be sensors/actuators, components, runnables, values, etc. used for the *realization* of more than one feature. According to our system specification model, we characterize feature interaction by means of *shared values*. Two features $f_1$ and $f_2$ *interact* if:

(i) Both features use at least one shared value $v \in \mathcal{V}$.

(ii) At least one feature assigns to $v$.

The second condition implies that $v$ cannot be a sensor value which fits to the intuition of feature interaction. We refer to this definition as *obvious* feature interaction as it also includes interactions explicitly intended such as simple passings of status flags from one process to another within the system. An *oblivious* feature interaction of $f_1$ and $f_2$ arises if:

(i) Both features use at least one shared value $v \in \mathcal{V}$.

(ii) Both features assign to $v$.

Hence, $f_1$ and $f_2$ might modify $v$ concurrently either affecting an internal control value, or the behavior of an actuator. Again, such interactions are often introduced by design and are synchronized properly, e.g. via control flags like *cls_status* in the sample *door system*. Nevertheless, such patterns must be exhaustively validated to rule out unintended behavior. If both features become activated simultaneously, i.e. orthogonal STATEFLOW (sub-) automata realizing different features that change the same value, contradicting forces can be injected to an actuator. A special case arises for two distinct actuators affected by different features that have contradicting effects on the environment. To detect such constellations, further knowledge concerning the environment may be necessary. In the *door system* example, both kinds of feature interactions are present as will be examined in the following section.

For detecting feature interactions in a set of features $f_1, f_2, \ldots, f_n$ realized by a system $\mathcal{F} = \langle\ \mathcal{C}, \mathcal{S}, \mathcal{A}, \mathcal{V}\ \rangle$, we define a *feature dependency graph* as $G_{FD} = \langle\ N_F, E_D\ \rangle$, where $N_F$ is the set of *nodes*, one for each feature $f_i$ of the system, and $E_D \subseteq N_F \times \mathcal{L} \times N_F$ is the set of *edges* connecting nodes whose features share at least one system artifact of $\mathcal{F}$ referred to by *labels* $L \in \mathcal{L}$. In the most general case, this is not restricted to a specific set of artifacts, and edges are undirected. In section 3.2, we will consider values $v \in \mathcal{V}$ as shared artifacts, and edges to be directed to denote read-write-dependencies.

### *2.5 Test Case Generation*

A system model as introduced can be used as a formal system specification for various purposes, e.g. formal analysis and automated code generation. As the model reflects the functional requirements of the system, it can also be used as a *test model* for deriving *test cases*, i.e. sample sequences of stimuli and the expected reaction of the system to be validated for the system implementation.

Stimuli are often called *events*, i.e. relevant, hence visible "impulses" occurring at a certain point in time and with a certain ordering to the interfaces of a (sub-) system under test. According to our system model abstractions, we assume such events to affect, i.e. change *values* in the system. Depending on the hierarchy level the tests are applied at, the values considered in test cases originate from interfaces of single functions, components, and lastly the whole system. In the last case, the stimuli are mainly related to sensor values, thus the test cases emulate the environment, against what at a lower level the tests also imitate events on local

values caused at some point within the system and recognized by the interface of the system artifact under test. A *test case* constructed from a behavioral specification $R$ can be defined as $t = \langle\ (\ e_1, e_2, \ldots, e_n\ ), \Phi\ \rangle$. Depending on the test level, $R$ might be either a basic function, or it is composed out of different components/compositions, up to an integrated model for the overall system. Events $e_i$ are changes of *values* $v \in \mathcal{V}$ that are inputs of $C_R$, hence affecting $\nu(v)$ and therefore triggering transitions and corresponding reactions in the STATEFLOW model. For a complete test case $t = \langle\ (\ e_1, e_2, \ldots, e_n\ ), \Phi\ \rangle$, the reaction of $R$ is a *run* $\Phi = (\sigma_0, \sigma_1, \ldots, \sigma_n)$. When $t$ is applied to the system under test, a failure arises if the system behavior differs from the behavior in $\Phi$ predicted for the test case. When performing black box tests, only the output values of $C_R$ are investigated and to be compared to those predicted in the configurations $\sigma_i$ of the *test oracle* $\Phi$. Otherwise, when performing white box tests, the complete details of states and values in the configuration chain of $\Phi$ can be taken into account. For example, a test case for the sample *power window* component may be:

$$t\_pw = \langle\ (key\_position := 1, key\_position := 0, key\_position := \text{-}1), \Phi\ \rangle$$

where the predicted end configuration in $\Phi$ is $\{door\_lock == 0, cls\_status == 0\}$. Further techniques like *fault modeling* can be used to inject incorrect or unexpected inputs to check whether the system steps to an error state [5].

A *test suite* constructed from a behavioral specification $R$ is given as $\mathcal{T} = \langle\ \{t_1, t_2, \ldots, t_m\}, \Psi\ \rangle$, thus a set of $m$ test cases $t_i$ and a coverage criterion $\Psi$ to be considered for the behavioral model of $R$, e.g. path coverage. The number $m$ of test cases to be generated to fulfill $\Psi$ depends on the rigorousness of the criterion, the size of the model (number of states, transitions, etc.), the number and domain of interface values of $R$, etc. The construction principles proposed in the following section aim at reducing $m$ when testing possible feature interactions by only composing those behavioral specifications to test models $R$ that potentially cause feature interactions.

# 3 Test Case Generation for Feature Interaction Analysis

## *3.1 Test Case Generation from Stateflow Models*

In general, model-based test suite generation consists of four steps: (1) building the test model, i.e. the system specification, (2) validating the test model, e.g. by simulation, (3) generating test cases concerning adequacy criteria, and (4) executing test cases. The adequacy criterion adopted heavily depends on the test model: for control/data flow oriented representations, i.e. flow graphs and automata-based approaches, graph algorithms can be used, e.g. for path coverage, and for source code oriented criteria, statement/expression coverage, def-use-analysis, etc. can be applied. Here, we consider the specification of the functional architecture $\mathcal{F}$ and the related behavioral STATEFLOW models $SF_R$ of the components' runnables to define the test model of the system under test. Features $f_i$ refer to the functional

requirements to be realized by the system, hence to be validated by test cases generated from the test model. The specification of a feature as introduced in Section 2 can be used to trace to the system artifacts involved in the realization of the related functional requirement, and therefore to be addressed for model-based test case generation. For Statechart-like formalisms like STATEFLOW, various approaches for model-based test case generation were proposed, e.g. in [5,2,6,11]. In general, the common basic ideas can be summarized as follows:

**Transformation into a flow graph:** Test case generation algorithms, e.g. for path coverage are primarily based on test models constituting basic data/control flow graphs. Therefore, Statechart-like formalisms leading to behavioral specifications in terms of hierarchical automata are to be transformed (*flattened*) first to be applicable as test models [5,11]:

(i) XOR states are removed by adding additional transitions. Beginning with the innermost XOR states, this is done until all regions of an AND state solely contain basic states. For example, in the model for the central locking system in Figure 3, the XOR state *cls_central_lock* would be removed, and the "unlock" transition would be added to each former sub state.

(ii) AND states are removed by constructing the Cartesian product of the sub states of all regions, i.e. resolving orthogonality by *interleaving*, see e.g. [5].

(iii) The steps (i) and (ii) are performed repeatedly bottom-up until the root state is reached, thus *sub* is "flushed".

**Adequacy Criteria Application and Test Case Selection:** In general, adequacy criteria, above all coverage criteria define measures to justify the effectiveness of a test suite in terms of its potential to reveal faults. Such criteria then indicate the point in time when to stop testing, i.e. when test case collections generated are "rich" enough for reliable correctness claims for the system. As a consequence, the criterion chosen implies the test generation methods used and the test model needed for the according algorithms.

For example, when applying path coverage, each possible transition sequence in the flow graph gives a test case, where the transitions are mapped to events (value assignments) matching the trigger of this particular transition. Especially for reactive control systems with non terminating control loops, the number of paths to be covered is unbound as the length of paths are potentially infinite. As a solution, an appropriate upper bound $k$ for the maximum path length is to be chosen in a way similar to the principles of *loop coverage* criteria [5]. Furthermore, model checking techniques can be applied to select distinguishing test sequences, see e.g. [14,15]. The *test oracle* to be generated contains at least the expected output, either solely for the final step, or for the complete trace of the test, and can be extended to exhaustive predictions of the internal system configurations supporting comprehensive debugging. For the application of techniques such as *fault modeling* [5] the test oracle may also define the error state to be reached. When dealing with data oriented criteria, control paths are to be covered not only once by an arbitrary value matching the transition conditions, but rather the whole domain of possible

values are to be taken into account, see e.g. [11] for details.

When embedding test case generation and application into an engineering process model, e.g. the V-model, different levels of abstractions and related system artifacts for test cases accompanying the development stages can be distinguished, e.g. function tests, i.e. testing single "atomic" functions of the system in isolation (e.g. moving the power window up), module/component tests, i.e. testing (sub-) functionalities (e.g. the complete power window feature), up to integration/system tests, i.e. testing the complete system functionality including the interaction of all features demanded in the requirements specification. Depending on the level of abstraction, the corresponding test model for test case generation is constructed by composing the partial models of the artifacts involved, i.e. the STATEFLOW models of (sub-) components to be integrated at that level. The corresponding test cases generated then propagate input stimuli of the resulting test model interface (see Section 2.2). The test case execution, i.e. the injection of the sequence of interface stimuli into the (sub-) system under test is realized by emulating the sources of value changes/events. For instance, the injection of sensor values, i.e. the simulation of the environment can be done by means of a hardware-in-the-loop setting.

### *3.2 Test Case Generation for Feature Interaction*

An exhaustive generation and execution of test suites for system tests is tedious as parallel composition of all components in the system leads to a flow graph with an exploding number of states (and therefore paths). In addition, mixing all features into comprehensive test cases makes it difficult to identify the sources of faults that lead to the failures observed during test application especially when caused by *oblivious* feature interaction. Therefore, an intermediate step between component tests and system tests is needed for specifically investigating the interaction of a limited number of features. The test model is then built by parallel composition of only those STATEFLOW models involved in the realization of at least one of the interacting features. First, this mixing can be done pairwise, then by combination of several interacting features, and finally the overall set of interrelated features, which is in the worst equivalent to system test generation. For a system specification $\mathcal{F} = \langle\, \mathcal{C}, \mathcal{S}, \mathcal{A}, \mathcal{V} \,\rangle$ realizing a set of features $f_1, f_2, \ldots, f_n$, the generation of feature interaction test cases can be done by the following steps.

#### *3.2.1 Feature Dependency Graph Construction*

As described in Section 2.4, we take the set of values $\mathcal{V}$ as the artifacts shared by features $f_i$ indicating a potential interaction for constructing $G_{FD} = \langle\, \{n_1, n_2, \ldots, n_n\}, E_D \,\rangle$, where nodes $n_i \in N_F$ represent features $f_i$, and edges $(n_i, v, n_j) \in V_D$ connect nodes of features $f_i$ and $f_j$, if they share a variable $v \in \mathcal{V}$. Edges $(n_i, v, n_j)$ are directed leading from a node of feature $f_i$ that assigns $v$ to a node of a feature $f_j$ that accesses $v$, either by assignment or reading. Consequently, $(n_i, v, n_i) \in E_D$ holds for each feature $f_i$ assigning $v$. For a feature $f_i = \langle\, \mathcal{S}_{f_i}, \mathcal{R}_{f_i}, \mathcal{A}_{f_i} \,\rangle$, we refer to $\mathcal{I}_{A_{f_i}}$ as the set of actuator values assigned by $f_i$. The set $\mathcal{O}_{S_{f_i}}$ of sensor values read by $f_i$ can be ignored in our construction as these

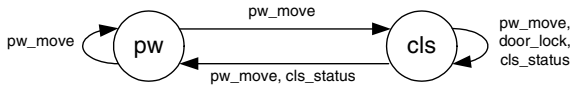



Fig. 4. Door System – Feature Dependency Graph

values are never assigned within the system and therefore can not cause a feature interaction. For runnables $R \in \mathcal{R}_{f_i}$ we refer to their components $C_R = \langle\, \mathcal{I}_R, \mathcal{O}_R, R \,\rangle$ as usual. We define the set of values $V_{f_i,c} \subseteq \mathcal{V}$ changed by at least one runnable of feature $f_i$ to be:

$$V_{f_i,c} = \mathcal{I}_{A_{f_i}} \cup \{ \bigcup_{R \in \mathcal{R}_{f_i}} \mathcal{O}_R \}$$

and the set of values (except sensor values) $V_{f_i,r} \subseteq \mathcal{V}$ read by at least one runnable of feature $f_i$ by:

$$V_{f_i,r} = \{ \bigcup_{R \in \mathcal{R}_{f_i}} \mathcal{I}_R \} \setminus \mathcal{O}_{S_{f_i}}$$

Algorithm 1 takes the features $f_i$ of a system specification $\mathcal{F}$ and the related sets $V_{f_i,c}$ and $V_{f_i,r}$ for $f_i$ as described above as inputs, and constructs the set of edges $E_D$ of the feature dependency graph $G_{FD}$ connecting nodes $n_i$ of features $f_i$. For

**Algorithm 1** Feature Dependency Graph Construction

```
1: for all f_i do
2:   for all v ∈ V_{f_i,c} do
3:     for all f_j do
4:       if v ∈ V_{f_j,c} ∪ V_{f_j,r} then
5:         E_D ← (n_i, v, n_j)
6:       end if
7:     end for
8:   end for
9: end for
```

each feature $f_i$, the set of values $v \in V_{f_i,c}$ potentially changed by that feature is considered. A possible interaction with a feature $f_j$ arises, if $f_j$ also accesses value $v$, either by reading or changing it, and an edge leading from $n_i$ to $n_j$ is added to $V_D$ labeled with $v$. Note that $(n_i, v, n_i) \in E_D$ also holds for this construction. Fig. 4 shows the feature dependency graph constructed for the sample *door system*. The nodes refer to the features power window (PW) and central locking system (CLS). Note that only those values at least changed by one of the features are present at the edges.

### *3.2.2 Feature Interaction Detection*

The feature dependency graph as constructed above explicitly reveals different degrees of *coupling* between features by means of shared values. For a $G_{FD}$, we denote the number of incoming edges of node $n_i$ labeled with variable $v \in \mathcal{V}$ by $|n_i|_v$. We consider the following cases for feature $f_i$:

(i) $|n_i|_v = 1$
  (a) $(f_i, v, f_i) \in E_d$, i.e. feature $f_i$ is the only feature changing value $v$, hence *no* feature interaction arises concerning $v$.
  (b) $(f_i, v, f_j) \in E_d$, where $i \neq j$, i.e. feature $f_j$ changes $V$ and feature $f_i$ reads $v$. Hence, there is an *obvious* feature interaction between $f_j$ and $f_i$.

(ii) $|n_i|_v > 1$
  (a) $(f_i, v, f_i) \notin E_d$, i.e. $f_i$ reads a value $v$ that is changed by interacting features. Thus, $f_i$ may be indirectly affected by a feature interaction of other features in the system.
  (b) $(f_i, v, f_i) \in E_d$, there is an *oblivious* interaction of $f_i$ with other features concurrently changing $v$.

We can now define the *set of interacting features* $FI_v$ concerning variable $v$, where $\{f_i, f_j\} \subseteq FI_v$ iff either case i(b) holds, or $(f_j, v, f_i) \in E_D$ and case ii(b) holds, respectively.

For example, in the *door system*, we have $|n_{PW}|_{pw\_move} = |n_{CLS}|_{pw\_move} = 2$, hence an *oblivious* feature interaction of the power window and the central locking system via concurrent accesses to the power window actuator and therefore $FI_{pw\_move} = \{PW, CLS\}$. In contrast, $|n_{PW}|_{cls\_status} = 1$ indicates an *obvious* interaction between both features by explicitly using the status flag.

### *3.2.3 Test Cases for Feature Interaction Analysis*

For a single feature $f_i = \langle\, \mathcal{S}_{f_i}, \mathcal{R}_{f_i}, \mathcal{A}_{f_i} \,\rangle$, a test model $SF_{f_i}$ can be constructed by orthogonal composition of all STATEFLOW automata in the set of runnables $\mathcal{R}_{f_i}$, and then transforming the composed STATEFLOW into a flow graph as described in Section 3.1. The interface for the resulting test model consists of $\mathcal{S}_{f_i}$, $\mathcal{A}_{f_i}$, and further globally visible values from the interface for $SF_{f_i}$.

The test model for interacting features $f_i \in FI_v$ and $f_j \in FI_v$ can now be constructed by the composition $SF_{f_i} \oplus SF_{f_j}$. For feature interaction aware test case generation, those transitions in the resulting flow graph are of interest, in whose ECA rules the shared variable $v$ occurs. For the creation of test cases $t = \langle\, (e_1, e_2, \ldots, e_n), \Phi \,\rangle$ that "provoke" a feature interaction, $t$ must contain events $e_i$ that trigger transitions caused by $f_i$ and accessing $v$, as well as events $e_j$ mapping to a transitions of $f_j$ also accessing $v$. Therefore, when constructing the transitions of the flow graph, annotations must be added carrying information about the feature the transition originates from. For comprehensive coverage of feature interactions, for test suites $\mathcal{T} = \langle\, \{t_1, t_2, \ldots, t_m\}, \Psi \,\rangle$ test cases must be selected that contain all different access constellations to $v$ and their orderings occurring in paths of the test model, i.e. read-write / write-read sequences (*obvious* interactions), and write-write sequences (*oblivious* interactions).

Again, consider the *door system*: a first simple test case for interacting features $PW$ and $CLS$ is shown in Table 1. Each row denotes an input event from the test sequence and the corresponding system *run*, i.e., the door is locked, and afterwards the window starts to move down released by a button stimulus, which

| **Input** | **PW Output** | **PW State** | **CLS Output** | **CLS State** |
|---|---|---|---|---|
| *key_position* := 1 | | *pw_up* | *door_lock* := 1 | → *cls_lock* |
| *pw_button* := −1 | | → *pw_pending* | | *cls_lock* |
| *pw_position* := 0 | *pw_move* := −1 | *pw_pending* | | *cls_lock* |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 1
First Test Case

| **Input** | **PW Output** | **PW State** | **CLS Output** | **CLS State** |
|---|---|---|---|---|
| *key_position* := 1 | | *pw_up* | *door_lock* := 1 | → *cls_lock* |
| *key_position* := 1 | | *pw_up* | *cls_status* := 1 | → *cls_pending* |
| *pw_position* := 0 | | *pw_up* | *pw_move* := 1 | *cls_pending* |
| *pw_button* := −1 | | *pw_up* | *pw_move* := 1 | *cls_pending* |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 2
Second Test Case

| **Input** | **PW Output** | **PW State** | **CLS Output** | **CLS State** |
|---|---|---|---|---|
| *key_position* := 1 | | *pw_up* | *door_lock* := 1 | → *cls_lock* |
| *pw_button* := −1 | | → *pw_pending* | | *cls_lock* |
| *pw_position* := 0 | *pw_move* := −1 | *pw_pending* | | *cls_lock* |
| *key_position* := 1 | | *pw_pending* | *cls_status* := 1 | → *cls_pending* |
| *pw_position* := 0 | **pw_move:=-1** | *pw_pending* | **pw_move:=1** | *cls_pending* |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 3
Third Test Case

conforms the test oracle. Note that the input stimulus of the window position sensor can be delivered continually by an environment model coupled to the power window actuator for closed control loop. Having detected the feature interaction $FI_{pw\_move} = \{ PW, CLS \}$, appropriate test cases for investigating conflicting accesses of $PW$ and $CLS$ to the power window actuator can be constructed. For this example, *oblivious* feature interactions arise from transitions $T_{pw\_move} \subseteq \mathcal{T}$ with $S \xrightarrow{E[C]/A} S'$, where action $A$ contains assignments $pw\_move := RHS$. Test cases for investigating this feature interaction must cover combinations of $t_{PW} \in T_{pw\_move}$ and $t_{CLS} \in T_{pw\_move}$ of all possible orders in the flow graph, where $t_{PW}$ refers to transitions of $SF_{PW} \oplus SF_{CLS}$ that originate from the Stateflow $SF_{PW}$ of feature $PW$, and $t_{CLS}$ accordingly from $SF_{CLS}$.

A test sequence where both features manipulate *pw_move* must include pushing the power window button and turning the key to the lock position twice as shown in Table 2. This scenario covers the interaction between $PW$ and $CLS$ via the value *cls_status* what we identified as *obvious* before. The status flag hinders the window to move down in case of active central locking as intended. Furthermore, the power window system is able to track the state (position) of the window via the position sensor, although it is manipulated by an "unknown" artifact in the system. Now consider the test case in Table 3, where a different ordering of the events is injected. Note that in the last step, the system reaches a state, where both features try to manipulate the window actuator concurrently by contradicting forces, which covers the *oblivious* feature interaction constellation as described above. Hence, with this

test case an incorrect synchronization between $PW$ and $CLS$ can be discovered in the behavioral specification and therefore in the system implementation. The problem is that the status flag is only checked by the power window controller when being in the uppermost position, and this state has already been left in the test case, when the central locking became active. Note that the example also includes a variant of a "physical" feature interaction as the actuator for moving the window directly affects the environment by means of the window position, which is then influencing the window position sensor and therefore, again, the features PW and CLS.

Going a step further, the potential effects of "transitive", i.e implicit interactions of features are to be investigated. Having two feature interaction sets $FI_v$ and $FI'_v$ for different values $v$ and $v'$, and $FI_v \cap FI'_v \neq \emptyset$, then features from $FI_v$ may interact with features from $FI'_v$, although they do not share any value. For example, if $FI_v = \{ f_1, f_2 \}$ and $FI'_v = \{ f_2, f_3 \}$, hence $FI_v \cap FI'_v = \{ f_2 \}$, $f_1$ and $f_3$ may influence each other via $f_2$. Therefore, a corresponding test model is to be built by composing those of $f_1, f_2$, and $f_3$. This can be generalized by not only intersecting *pairs* of $FI$, but rather all possible combinations, which will naturally lead to the test model for the complete system integrating all features. To what extent this process shall be performed can depend on decision criteria such as the safety level of features involved, the degree of coupling, i.e. the number of shared values, etc.

## 4 Conclusion and Outlook

We described an approach for model-based test generation that explicitly take the detection and comprehensive investigation of features interaction into account, yet aiming at reduction of test cases needed compared to exhaustive functionality tests at system level. The approach is based on a rigorous modeling of both, the functional architecture as well as behavioral specifications using Statechart-like formalisms. As future work, different levels of "transitive" couplings are to be investigated in more detail, e.g. by adapting the notion of *obvious* and *oblivious* interaction accordingly, which can potentially lead to a further reduction of test cases. Also, adequate models for explicitly investigating feature interaction arising from causal dependencies in the environment can be integrated, e.g. for dealing with feedback loops as well as causal dependencies between sensor values such as temperature and daylight sensors. Finally, advanced specifications of behavioral models are to be addressed in test cases, e.g. timing constraints. Depending on the timing model used, i.e. synchronous vs. asynchronous step semantics [9,8], according approaches are to be applied. For instance, introducing clocks as variables and related tick events as additional inputs, transition conditions can be defined for timing constraints, which can be explicitly taken into account during test case generation and execution.

## References

[1] AUTOSAR Partnership, *Automotive Open System Architecture* (last visited Nov. 2009). URL http://www.autosar.org

[2] Bader, A., A. S. M. Sajeev and S. Ramakrishnan, *Testing Concurrency and Communication in Distributed Objects*, in: *5th International Conference On High Performance Computing, 1998. HIPC '98*, 1998, pp. 422–428.

[3] Bass, L., P. Clements and R. Kazman, "Software Architecture in Practice," Addison-Wesley Longman, Amsterdam, 2003, 2 edition.

[4] Bauer, H., K.-H. Dietsche and J. Crepin, "BOSCH Automotive Handbook," Robert Bosch GmbH, Stuttgart, 2000, 5 edition.

[5] Belli, F. and A. Hollmann, *Test Generation and Minimization with "Basic" Statecharts*, in: *SAC*, 2008, pp. 718–723.

[6] Bogdanov, K., M. Holcombe and H. Singh, *Automated Test Set Generation for Statecharts*, in: *FM-Trends*, 1998, pp. 107–121.

[7] Calder, M., M. Kolberg, E. H. Magill and S. Reiff-Marganiec, *Feature Interaction: A Critical Review and onsidered Forecast*, Computer Networks **41** (2003), pp. 115–141.

[8] Hamon, G. and J. Rushby, *An operational semantics for Stateflow*, International Journal on Software Tools for Technology Transfer (STTT) **9** (2007), pp. 447–456.

[9] Harel, D., *Statecharts: A Visual Formalism for Complex Systems*, Sci. Comput. Program. **8** (1987).

[10] Juarez-Dominguez, A. L., N. A. Day and J. J. Joyce, *Modelling Feature Interactions in the Automotive Domain*, in: *MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering* (2008), pp. 45–50.

[11] Li, L. and Z. Qi, *Test Selection from UML Statecharts*, in: *TOOLS '99: Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems* (1999), p. 273.

[12] Metzger, A., *Feature Interactions in Embedded Control Systems*, Computer Networks **45** (2004).

[13] Müller, T., M. Lochau, S. Detering, F. Saust, H. Garbers, L. Märtin, T. Form and U. Goltz, *A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance*, Technical Report 2009-06, TU Braunschweig (2009).

[14] Robinson-Mallett, C., T. Mücke, P. Liggesmeyer and U. Goltz, *Generating optimal distinguishing sequences with a model checker.*, in: *A-MOST*, 2005.

[15] Robinson-Mallett, C., T. Mücke, P. Liggesmeyer and U. Goltz, *Extended state identification and verification using a model checker*, in: *IST Special Issue on Model Based Testing*, 2006.

[16] von der Beeck, M., *A Comparison of Statecharts Variants*, in: *FTRTFT*, 1994, pp. 128–148.