GUI Failures of In-Vehicle Infotainment: Analysis, Classification, Challenges, and Capabilities

Daniel Mauser
Daimler AG
Ulm, Germany
daniel.mauser@daimler.com

Alexander Klaus
Fraunhofer IESE
Kaiserslautern, Germany
alexander.klaus@iese.fraunhofer.de

Konstantin Holl
Fraunhofer IESE
Kaiserslautern, Germany
konstantin.holl@iese.fraunhofer.de

Ran Zhang
Robert Bosch GmbH
Leonberg, Germany
ran.zhang@de.bosch.com

***Abstract*—With the growth of complexity in modern automotive infotainment systems, graphical user interfaces become more and more sophisticated, and this leads to various challenges in software testing. Due to the enormous amount of possible interactions, test engineers have to decide, which test aspects to focus on. In this paper, we examine what types of failures can be found in graphical user interfaces of automotive infotainment systems, and how frequently they occur. In total, we have analyzed more than 3,000 failures, found and fixed during the development of automotive infotainment systems at Audi, Bosch, and Mercedes-Benz. We applied the Orthogonal Defect Classification for categorizing these failures. The difficulties we faced when applying this classification led us to formulating requirements for an own classification scheme. On this basis, we have developed a hierarchical classification scheme for failures grounded on common concepts in software engineering, such as Model-View-Controller and Screens. The results of the application of our classification show that 62% of the reports describe failures related to behavior, 25% of the reports describe failures related to contents, 6% of the reports describe failures related to design, and 7% of the reports describe failures to be categorized. An outlined capability of the results is the support for fault seeding approaches which leads to the challenge of tracing the found failures to the correspondent faults.**

Keywords-domain specific failures; GUI based software; in-vehicle infotainment system; failure classification; fault seeding.

I. Introduction

This article focuses on classifying failures found and fixed during the development of automotive infotainment systems. As the research was conducted as part of a funded research project, we had the unique chance to analyze failure data collected by both car manufacturers and suppliers. The developed classification was awarded as best paper on the Fourth International Conference on Advances in System Testing and Validation Lifecycle [1] and invited for an additional journal publication.

In modern automotive infotainment systems ("Infotainment" is a combination of "information" and "entertainment"), the graphical user interface (GUI) is an essential part of the software. The so-called human machine interfaces (HMI) enable the user to interact with the system functionality, such as the radio system, the navigation, or the tire pressure monitoring system. According to Robinson and Brooks [2], a GUI "is essential to customers, who must use it whenever they need to interact with the system". Additionally, they "found that the majority of customer-reported GUI defects had a major impact on day-to-day operations, but were not fixed until the next major release" [2].

GUI-based software, especially in the automotive domain, is becoming more and more complex [3] - often, documents with more than 2,000 pages are written to describe all the functionality [4]. The reasons are (a) the growing number of functions, which form more and more complex systems, as well as (b) increasing variability due to more adaptive and customizable interaction behavior.

When testing GUIs, sequences of system interactions are performed and the system reaction is compared to the specified reaction in each step. It is obvious that not all possible combinations of user inputs can be tested. Thus, it is necessary to focus testing activities on certain failure types. To be able to (a) choose strategies accordingly, (b) adjust test case development or (c) guide failure recognition, the following questions need to be answered: What types of failures are to be expected in GUI based software today? Is it possible to build a classification of these types? What

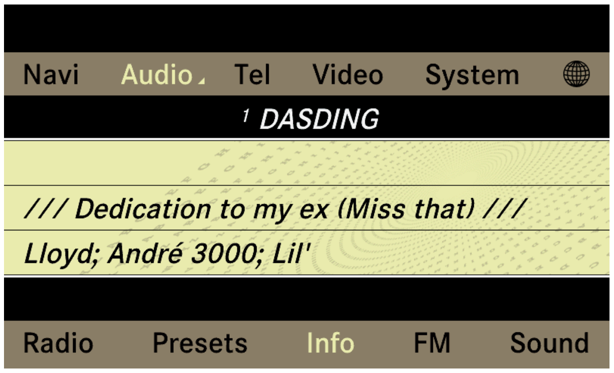


Figure 1. Example of a graphical user interface of the Mercedes-Benz infotainment system COMAND.

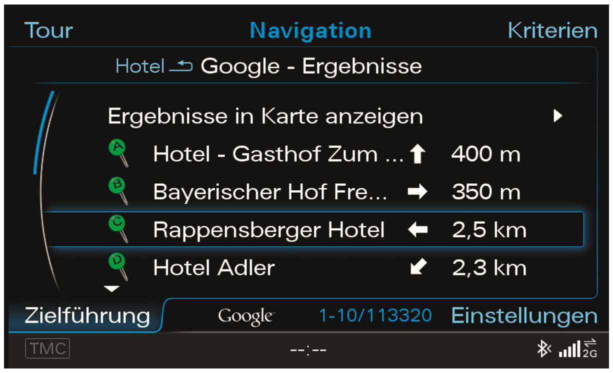


Figure 2. Example of a graphical user interface of the AUDI infotainment system MMI [6].

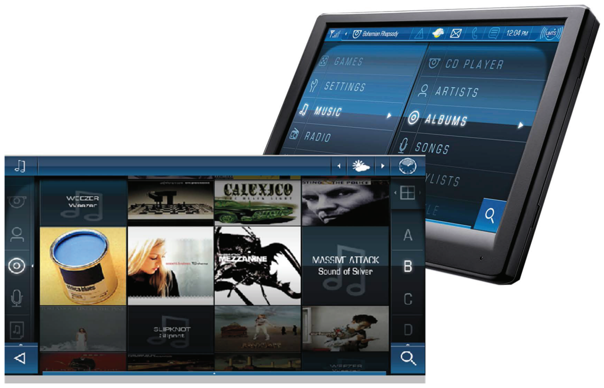


Figure 3. Example of a graphical user interface of the Bosch Multimedia Reference System (MRS).

are frequent failures in current GUI software? Which are common, which are rare?

Our context is the quality assurance of GUIs for automotive infotainment systems. As these are built into a car, the situation is different from that of desktop software. There is no convenient possibility to upgrade the system or to buy a new release, which means that manufacturers need to assure quality in the first release. Additionally, when the system does not work correctly, drivers may get distracted from driving. Therefore, special attention has to be paid to find and fix defects during development. The interaction with the system is different from that of desktop GUIs [5]. A common interaction device is the central control element (CCE).

Based on typical examples, the structure and the interaction concept of automotive infotainment systems are described in the following. In the Mercedes-Benz COMAND system shown in Figure 1, the GUI consists of a menu at the top of the screen, where all available applications, e.g., navigation or audio, can be accessed. Each application consists of an application area in the middle of the screen, where the actual content is displayed (here: information about the radio station and the song being played) and a sub-menu for content-specific options at the bottom (here: “Radio”, “Presets”, “Info”, etc.). The GUI is operated via the CCE, allowing the user to set the selection focus by rotating or pushing the CCE in one direction, and to activate options by pressing it.

Another well-known in-vehicle infotainment system is the “MMI” (Multi Media Interface) developed by Audi. Figure 2 shows a GUI of the MMI with an example of a navigation program. In contrast to COMAND, the main menu options of the Audi infotainment system are located in the four corners of the screen. In the middle area of the screen, the information and the menu options of the navigation program are displayed. To operate the GUI, a physical interactive component called MMI-Terminal is used, which consists of a central button allowing rotary and push operations, as well as four push buttons around the central button. Analogous to COMAND, the MMI enables the focus selection and the operation confirmation of the GUI.

Besides the above described ones, Bosch has introduced another in-vehicle infotainment system called “Multimedia Reference System” (MRS), which is based on an open source platform. Compared with infotainment systems of Mercedes-Benz and Audi, the MRS focuses on a full touch solution. Figure 3 exemplifies the MRS with a view of the main menu and with a view of the albums. The top of the screen is the area displaying the status of applications, such as E-mail, phone and weather report. The left border and right border are used for hot keys related to several frequently used functionality.

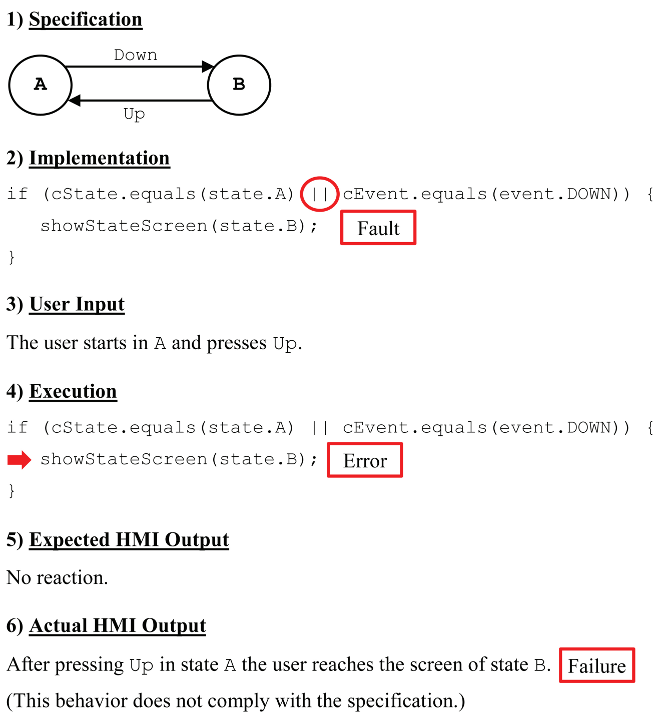
This article is structured as follows. In Section II, we discuss related work and stress the need to create a new classification scheme. In Section III, we describe how we applied the scheme that has been identified as most appropriate in an empirical study. In Section IV, we present our approach to develop the classification. The scheme itself is detailed in Section V. Section VI discusses the results based on the defined requirements. Section VII presents concluding remarks and future research directions.

II. Related Work

In the literature, various types of defect classifications can be found. However, many of them lack practical usage and empirical data in the form of distributions of defects into the scheme, and thus it is hard to tell whether they are a valuable addition. Other schemes for classification are used frequently, or at least once. For our study, we concentrate on those latter ones, and discuss why they are not fully suited for our means. As described above, our context is black-box testing of a GUI for automotive infotainment systems.

A. Definitions

First, we have to clarify the distinction between different terms for “defects”. The IEEE [7] released a standard for defect classification, which also includes a scheme for



1) Specification

$$\begin{tikzcd} A \arrow[r, shift left, "Down"] & B \arrow[l, shift left, "Up"] \end{tikzcd}$$

2) Implementation

```
if (cState.equals(state.A) || cEvent.equals(event.DOWN)) {
   showStateScreen(state.B);   Fault
}
```

3) User Input

The user starts in A and presses Up.

4) Execution

```
if (cState.equals(state.A) || cEvent.equals(event.DOWN)) {
➡ showStateScreen(state.B);   Error
}
```

5) Expected HMI Output

No reaction.

6) Actual HMI Output

After pressing Up in state A the user reaches the screen of state B. Failure

(This behavior does not comply with the specification.)

Figure 4. Example of the distinction of a fault, an error and a failure.

distinguishing between defects and failures. A defect is "an imperfection or deficiency in a work product that does not meet its requirements or specifications", while a failure is "an event, in which a system or system component does not perform a required function within specified limits" [7]. Therefore, when a defect is present, and we perform GUI testing, we can observe failures. They are caused by defects in the code, but since we test by using the GUI, and not the code (i.e., black-box), what we can observe is the behavior. This is why we do not create a defect but a failure classification scheme. The missing consistency of precise terms within the related work is complicating its conflation. According to [7], e.g., the terms anomaly, error, fault, failure, incident, flaw, problem, gripe, glitch, defect, and bug are often used synonymously. There is no need for our work to define all related terms. Here, "defect" is used as a collective noun. In accordance with IEEE [7], the usage of the terms fault, error and failure is based on Jean-Claude Laprie [8]: "A system failure occurs when the delivered service deviates from fulfilling the system function, the latter being what the system is aimed at. An error is that part of the system state, which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a fault." Figure 4 shows a concise example in the HMI context for clarifying the distinction between the terms.

According to [9], faults cause errors and errors cause failures. However, not every fault is the reason for an error and not every error is the reason for a failure. Hence black-box testing can lead to a sophisticated task because some faults cause failures only in very particular situations. In addition, failures that are caused by faults and lead to errors can cause other faults – resulting in a propagation of faults. Additionally, no one-to-one correspondence of failure and faults can be assumed. One failure can be symptom for more than one fault, one fault can cause more than one failure. The analyzed reports are based on the results of black-box testing. Thus, only failures were detected and documented within these reports. They contain no information about the faults – the root of the failures.

B. Defect Classification Schemes

IBM created the so-called Orthogonal Defect Classification (ODC) [10] in the early nineties. Since then, many companies have applied this approach. It consists of several attributes, such as *triggers*, *defect types*, *impact*, and others. A GUI section is included in the ODC extension V5.11 [10]. It contains *triggers*, such as *design conformance*, *navigation*, and *widget / GUI behavior*.

Another scheme, which contains several categories for GUI-related issues, was proposed by Li et al. [11]. It consists of 300 categories and is based on the ODC, but adapted for black-box testing. It contains, e.g., categories for a *GUI in general*, and for *GUI control* [11]. However, this scheme contains many categories that refer to highly specific GUI elements and therefore lacks in abstraction levels. For example, there are categories for a *Textbox*, *Dropdown list*, or a *Title bar* that are not applicable to systems that do not contain those. The scheme also contains categories for *interaction of various menus* or *display styles* [11]. There is no further differentiation, e.g., there may be an unexpected reaction of the system, or there may be no reaction when using a menu. This scheme is created for regular desktop software, as it also classifies keyboard- or mouse-related faults. In order to adapt this schema, a large number of categories would have to be exchanged. As there are no further abstraction levels, only few common aspects remain which limits the potential of general conclusions.

Børretzen and Dyre-Hansen [12] created a scheme that is also based on the ODC. They target industrial projects. A single GUI fault category is included, but not further segmented. The rationale for this is that, although "function and GUI faults are the most common fault types", they are most often not severe, and thus, not as critical as other categories [12]. This seems to be a contradiction to what was stated in the introduction, but the criticalities of certain types of defects are subject to the application domain. In the beginning, in our application domain they are very critical, and therefore, we focus on them to assure software quality.

Hewlett-Packard created a scheme based on three categories: *origin*, *type*, and *mode* [13]. *Origin* refers to where the defect was introduced; the *type* can, e.g., be *logic*, *computation*, or *user interface*. The *mode* refers to why something has been entered: *missing*, *unclear*, *wrong*, *changed*, or *better way*. This last category, *mode*, is an interesting detail for classifications, as it not only allows a deeper hierarchical structure, but also allows distinguishing different kinds of defects of one type. However, the scheme created by Hewlett-Packard does not distinguish the various types of GUI-related failures, and, thus, does not enable us to categorize our defects.

Another well-known scheme was developed by Beizer [14]. The main categories are "requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture, and testing" ([14], p. 33), each having three levels of subcategories. The scheme is very detailed, but does not contain GUI-related categories. An adaptation of this scheme for GUI contexts was created by Brooks, Robinson and Memon [15]. The authors emphasize that "defining a GUI-fault classification scheme remains an open area for research" [15]. They simplified Beizer's scheme to create a two-level classification and added a subcategory for GUI-related issues, "to categorize defects that exist either in the graphical elements of the GUI or in the interaction between the GUI and the underlying application" [15]. However, since there is only one category specifically for GUIs and since we focus on GUIs, it is not possible to use this scheme for our purposes. Adapting it would result in the same effort as creating a separate one.

There also exists a fault classification scheme for automotive infotainment systems [16]; however, this scheme is based on network communication and can thus not be used for classifying software based GUI failures. This scheme differentiates between hardware and software, but does not differentiate further. It also has many categories not usable in our context, and does not include different GUI-related categories. Ploski et al. [17] studied several schemes for classification, including approaches not presented here. Since there were no matching schemes, we do not present them here.

Another approach was created by the IEEE [7]. However, this approach lists a number of attributes to be filled out for each defect and is not expedient for reaching our goals. This is due to the purpose of the standard to "define a common vocabulary with which different people and organizations can communicate [...] and to establish a common set of attributes that support industry techniques for analyzing software defect and failure data" [7]. This is much broader than what we want to achieve. However, the examples of defect attribute values in the standard contain a *mode* section with the values *wrong*, *missing*, and *extra* [7]. We adapted this *mode* section, and expanded it where necessary. The results will be presented in Section V.

The classification schemes available do not meet our requirements. Since we employ black-box testing of GUIs, we cannot use any code-related categories or schemes. We focus only on GUI-related failures. The schemes presented in [13][14] and [16] do not have GUI-related categories and because of this, they cannot be used by us. Others ([12][15]) have GUI-related categories, but still do not match very well to our purposes. The scheme presented in [11] has many GUI-related categories, but for desktop software. Due to the differences between desktop and automotive infotainment GUIs, we did not adapt it because we would then have had to either delete or change most of the categories.

As the trigger aspect listed in the ODC ([10]) was identified as the most appropriate existing scheme we found, an experimental application of the ODC was conducted, and the results are presented in Section III. For now, we just state that the ODC in the current state cannot be employed for our purposes perfectly. Since using or adapting other schemes does not lead to savings in effort (no differentiated GUI categories to use, most categories not applicable), we created our own failure classification scheme. After describing the approach we used, the categories of our scheme are explained in Section V.

III. Empirical Pre-Study

As stated in Section II, the ODC [18] [10] seems to be the most appropriate scheme to classify failures in GUIs for automotive infotainment systems. It provides eight attributes, such as *triggers*, *defect types*, *impact*, and others, describing pieces of information concerning a defect from different points of view. The ODC is intended to facilitate the entire bug tracking process including reproducibility (opener section) and fixing (closer section). The information of the ODC describing how the defect has been produced can be specified in the opener section using so called "trigger" categories. According to [18], a trigger is "the environment or condition that had to exist for the defect to surface". As this paper focuses on classifying failure types, this trigger section is most relevant for our purposes. Originally, the ODC included not primarily GUI related categories, such as "Logic/Flow" or "Concurrency". With the extension v5.11, these have been extended for graphical user interfaces introducing the values *design conformance*, *widget / icon appearance*, *screen text / characters*, *input devices*, *navigation*, and *widget / GUI behavior* [18]. See Table I below for an explanation for the trigger values.

A. Design

For this research, we analyzed databases of existing failure reports. The data was collected during the development of state-of-the-art automotive infotainment systems. The testers executed the System Under Test (SUT) manually, based on specification documents, and used failure reporting tools to keep records of anomalies. The reports were handed over to the developers, who then rechecked and fixed the software. In this context, failures are defined as mismatch between the SUT and an explicit GUI specification, which can be observed while operating the system. Any implicit requirements, such as general standards or guidelines, are not subject of the study. Only reports that were accepted as failures by both testers and developers were considered. Failures not referring to the GUI were sorted out. Examples for such failures are hardware errors or display flickering.

As a threat to the validity of this application of the ODC has to be mentioned, that we want to use the classification for dynamic testing purposes. However, in this section, we also selected entries for the *design conformance* value. One can argue that reviewing the design documents is not a dynamic testing activity. However, the ODC is not meant solely for testing, and a real application has to be done for the whole life cycle. When we simulate the usage of the ODC, we have to account for this, even when we do this in the aftermath of quality assurance activities. Additionally, a deviation between the design documents and the realization in a system cannot be found until the application is implemented and the implemented design can be reviewed and compared to the design documents.

B. Execution

For this study, Audi, Bosch, and Mercedes-Benz provided failure data. Hence, the analyzed reports represent a broad variety of contexts, as they stand for different infotainment systems (Audi MMI, Mercedes-Benz COMAND, and several projects developed at Bosch), different steps in the development process involved (e.g., module, system, and acceptance test), as well as different test strategies, test personnel, and test environments. As preparation to the analysis, the reports were exported to an Excel document. Testers sometimes recognize several anomalies at once but register those only in a single report. Therefore, reports that describe more than one failure have been split up in one line for each failure. Redundant reports that describe exactly the same failure as already considered ones were removed. After that, more than 3,000 reports remained to be analyzed. One third of the reports were used as training data to construct the failure classification, which was then fine-tuned using the remaining reports as test data. The following information per report was relevant for the analysis:

Table I
TRIGGER VALUES OF THE ODC EXTENSIONS V5.11.

Value for the attribute "trigger"	Description
Widget / GUI Behavior	Concerned with the system reaction related to widget / GUI elements.
Navigation	Concerned with the system reaction related to navigating between screens.
Widget / Icon Appearance	Concerned with the layout / design of widget or icon elements.
Design Conformance	Concerned with the conformance of the design of the developed application with the design documents.
Screen Text / Characters	Concerned with the correctness of labels or other text elements.
Input Devices	Concerned with the system reaction related to using various input devices.

Table II
EXAMPLES OF THE ANALYZED FAILURE REPORTS.

ID	Title	Problem description
4711	Inserted music CDs are not played automatically	*Setup*: Any state *Actions*: Insert music CD *Observed result*: Nothing happens *Expected result*: System should display CD play screen *Reference*: R0026679 *Workaround*: Navigate to CD play screen manually
4712	Cell phone icon on call screen obsolete	*Setup*: Connect cell phone *Actions*: Navigate to Call screen *Observed result*: Placeholder icon for cell phones is displayed *Expected result*: Correct icon is displayed *Reference*: R0026672 *Workaround*: —

A *Report ID* provides unique identification for each report. In the *Title*, the testers describe the essence of the report. The *Problem description* is a detailed statement about (a) the required setup of the system under test, (b) the actions that lead to the failure, (c) the behavior or result that has been observed, (d) a description of what should have been displayed instead, and (e) how this failure could be bypassed. If failures were ambiguous or hard to describe, screen shots were added. Table II shows simple examples of reports.

C. Results & Discussion

In Table III, the percentages of the pre-study results are presented. As shown, of the more than 3,000 failure reports, we could classify more than 90% into the values suggested by the ODC extension. However, as we focus exclusively on HMI software testing, it was not possible to classify any reports to the *input devices* value. Input device reliability had been ensured in previous testing phases. We examined the failure reports manually to categorize them according to the trigger values mentioned above. During this process, we had the impression that the values in the ODC are not as disjunctive as expected: [18] states that an example of *widget / GUI behavior* is "help button doesn't work". However, when this button is pressed, one could argue that an attempt to navigate has been made. Thus, such a failure could also be categorized with the *navigation* value for the *trigger*. To be able to categorize such failures, we decided to use screens as a criterion; if the screen does change although it should not, or if the wrong screen is presented, or no new screen appears although it should, then we classified this as *navigation*, otherwise as *widget / GUI behavior*. More than 50% of all reports fall into these two values: we classified 18% as *navigation* failures, and 38% as wrong *widget / GUI*

Table III
RESULTS OF THE ODC APPLICATION.

Value for the attribute "trigger"	Distribution
Widget / GUI Behavior	38.0%
Navigation	18.0%
Widget / Icon Appearance	17.0%
Design Conformance	9.6%
Screen Text / Characters	9.2%
Input Devices	0%
—	—
Remaining reports not classified	8.2%

behavior.

We could classify 9.6% as *design conformance* and 9.2% as *screen text / characters*. Differentiation between these two values was not clear either. In this case, we had to use an additional criterion for separation: If a text is "wrong" and the text itself is known only at run time, then it is a *screen text / characters* failure. If a text is "wrong" and is known already at design time, then it can be found in design documents, and thus it is categorized under *design conformance*. This separation does not comply with the examples given in [18]. However, we did not consider the examples given for these values sufficient, as the example for *screen text / characters* is limited to the description "button mislabeled". Such a label is known at design time, and thus, the label has to be defined in the design documents. If the label is correct in the documents and wrong in the implemented system, the application does not conform to its design. This is a problem with *design conformance*, but it is listed under *screen text / characters*. Now, we could also differentiate whether the label is already wrong in the design documents or not and then had the possibility to categorize it. Nevertheless, this discussion shows that the values are not detailed enough as necessary for our purposes.

The last remaining value in the *trigger* section, *widget / icon appearance*, was used to classify about 17% of all reports. One additional problem we faced was that we also had to classify failures in relation to animations. Here, we decided to use the same criterion as with the *design conformance* and the *screen text / characters* values: Is the problem already known at design time or only at run time? The former are categorized as being a problem with *design conformance*, the latter were tagged with the *widget / icon appearance* value.

In summary, we state that the categorization following the ODC extension v5.11 was not satisfactory. Besides the difficulties with the values not being disjunctive enough for our purposes, which led to the usage of additional criteria, the distribution across the trigger values is rather imbalanced. Not a single entry could be categorized into *input devices*, because this is not in the focus of the testing activities we examined, so this cannot be counted as a weakness in the ODC. But for the remaining values, we have two categories with more than 9%, two categories with nearly 18%, and one category with more than 38%.

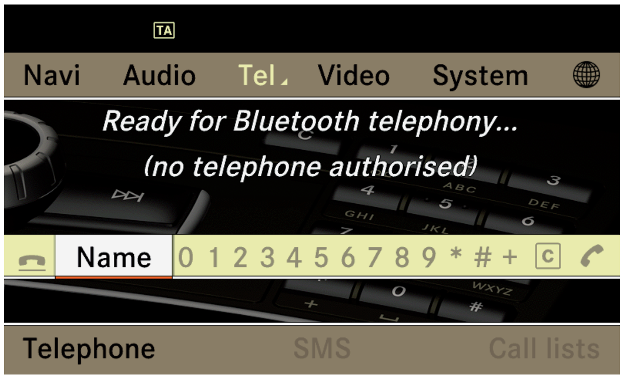
IV. APPROACH

The experiences described in Section II led us to the conclusion that it is more appropriate to create our own classification scheme, which would be more suitable for our needs. Following the lessons we had learned, we tried to include the additional categories we invented for using the ODC, and we posed requirements, for example to prevent categories growing as large as the *widget / GUI behavior* value. It should also be possible to use the ODC in combination with our taxonomy.

Therefore, a classification is needed that both gives a good overview and allows extension for comprehensiveness. Guidance is necessary to avoid universal categories with little information. In order to achieve those objectives, a hierarchical structure seems adequate: the lowest levels represent the actual failure class. Higher levels should summarize similar categories on the following level. By doing so, the impact of adding additional classes in the future should be mitigated, and different versions of the classification should be comparable at least at higher abstraction levels, such as "logic" or "design". Developing failure classes on lower levels has to be conducted thoroughly: On the one hand, classes have to be sufficiently abstract to satisfy the various analyzed contexts; on the other hand, they still have to be meaningful. As an indication of how many hierarchy levels have to be applied and whether one category could be subdivided reasonably or whether several categories should be combined, we defined the following requirements for the failure classes:

- To scale the scope of each classification level, an initial analysis of the data indicates the necessity to limit the percentage of the lowest level to 10% of the total numbers of failures.
- To develop a clear and easy-to-use structure, the number of categories on every level has to be a minimum of 2 and a maximum of 5.
- To ensure reproducibility, the assignment of failure reports should allow no ambiguity. Each failure class on the lowest level has to be disjunctive and well-defined.

The development of the classification was influenced by the Bug Tracking Systems (BTS) in use, as they already allow to roughly categorize reports. However, as this classification is intended (a) to focus on GUI-related failures and (b) to be applicable not only to one system, we combined several report databases that use different failure categories with varying levels of abstraction. During the development stage analyzing one third of the reports, the classification had to be conducted manually. Once the basic structures had been established, the newly developed categories could be compared and systematically reviewed to match the existing ones. Unclear reports were reviewed and information required for classification was added. In the future, BTS



Ready for Bluetooth telephony...
(no telephone authorised)

Figure 5. Screen example: Telephone application.

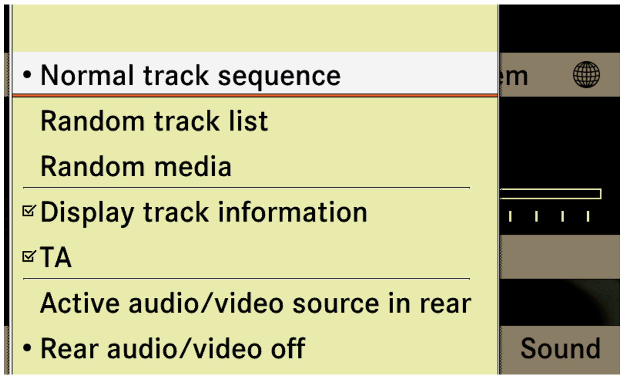


Figure 6. Screen example: Overlaying submenu.

might be able to provide these more detailed classes to make the classification during the GUI testing process more meaningful. Manual categorization would then no longer be necessary.

To determine the similarity of failures, the classification is based on concepts and patterns used in software engineering. For example, the top-level failure classes are *behavior*, *contents*, and *design*, according to the well-established Model-View-Controller [19] design pattern. The structure of the classification and the related separation criteria are presented in Section V.

V. Failure Classification

In this section, the GUI failure report classification is described. Table IV gives an overview of the entire classification, including the failure distribution. In Figure 7, the distribution of the most frequent classes is illustrated. As mentioned above, the top level follows the Model-View-Controller concept [19], proved to be an adequate abstraction for GUI-based software. This choice was made due to the authors' background as software developers. *Controllers* (here: *behavior*) abstract the observable behavior, indicating how input is processed. *Models* (here: *contents*) define all contents that are displayed by the system. *Views* (here: *design*) describe the layout and appearance of the contents to be displayed. As the SUT was tested as a blackbox, the MVC pattern is not intended to represent the actual software structure or to relate any failures to implemented software modules.

In order to avoid enforced classifications of reports to existing classes, a category "to be categorized" (TBC) was created. As for other categories, on the lowest level the TBC failure class is limited to 10% of the total number of failures. Classifying more failures than that limit as TBC would indicate that the definition of an additional failure class is necessary.

A. Behavior

The top-level failure class *behavior* contains all failure reports describing that stimuli to the SUT do not result in the specified output. In order to subdivide this failure class, common abstractions in GUI development were applied:

Screens [20][21] represent the current state of the GUI displayed. This state defines the options available to the user. Figure 1 shows the radio screen, where the current radio station and the song playing are displayed. The options provided allow users to change the waveband (FM option) or adjust the sound setting (Sound option).

The scope of screens is often a matter of system design. For example, in the COMAND infotainment system, similar to desktop applications, some of the options shown in the first place are general topics. Upon activation, a submenu is displayed on top of the remaining screen content (Figure 6). As the context of use remains unchanged, those menus are considered as part of the original screen, although they are not displayed all the time.

Screens are structured based on elementary GUI elements, so-called *widgets*. Widgets are either primitive (label, rectangle, etc.) or complex, meaning that they are compositions of primitive or again complex widgets. An example of widgets in Figure 5 would be the horizontal list in the top part. This list contains button widgets for all available applications, such as "Navi", "Audio", or "Tel" (i.e., phone). In terms of interaction logic, lists primarily manage the focus. Lists determine how their content can be iterated and what option is focused on (re)entering. If many options are available, e.g., when entering alphanumeric characters, the middle of the available options has to be focused on start. In cases of touch screens as input modality, lists would calculate the touch points of their containing entries depending on their visibility. Buttons consist of labels and/or symbols representing their function to the user. Additionally, buttons might define what actions have to be executed on pressing them and provide their visibility status on demand.

In this classification, the concepts of screens and widgets are used to differentiate micro behavior, which affects single elements on the display (e.g., iterating list entries), and macro behavior, which changes the entire context of use.

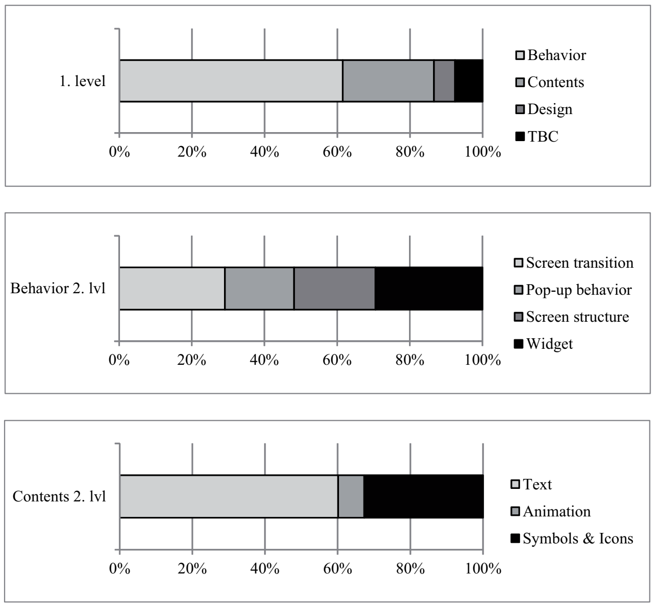


Figure 7. Failure distribution overview.

1) Widget Failures: The GUIs of the automotive infotainment systems analyzed mainly use various types of lists to present options to the user. To activate an option, those lists set a focus by having the user turn or push the CCE and press it once the desired option is focused. Potential failures might be that the wrong option is focused on start or that the focus does not change as specified. An example would be that every time the main menu is entered, the element in the middle should be focused automatically. A failure would exist if the first element would be focused instead. Those failures are considered as deficient *widgets focus* logic. The subcategories are:

- *initial*: the wrong option is focused when a list is (re-) entered.
- *implicit*: the focus has to be reset due to changing system conditions.
- *explicit*: the user resets the focus by turning or pushing the CCE.

For widgets, additional behavior is often specified. One example might be alphabetic scrolling to allow the user to jump to a subgroup of list entries starting with one specific letter. Those failures are considered as deficient *widget behavior*. Subcategories are:

- *missing*: specified behavior is not implemented.
- *wrong*: instead of the specified behavior, not specified behavior is implemented.
- *extra*: behavior is implemented but is not specified.

2) Screen Structure Failures: In this failure class, reports are clustered describing the logic for determining the widget objects the screens contain and what data they hold. In automotive infotainment systems, the availability of options

Table IV
THE DISTRIBUTION OF FAILURES.

1. level	2. level	3. level	4. level	distr.
TBC	-	-	-	*7.6 %*
Behavior (Σ: 61.5%)	Screen Transition (Σ: 17.9%)	missing	-	*5.8 %*
		extra	-	*2.9 %*
		wrong	-	*9.2 %*
	Pop-up Behavior (Σ: 11.7%)	missing	-	*3.6 %*
		extra	-	*3.2 %*
		priority	-	*0.5 %*
		wrong	-	*4.4 %*
	Screen Structure (Σ: 13.8%)	screen composition (Σ: 5.4%)	missing	*2.4 %*
			extra	*0.9 %*
			wrong	*2.1 %*
		options offer (Σ: 5.4%)	missing	*2.2 %*
			extra	*1.3 %*
			wrong	*1.0 %*
			order	*0.9 %*
		option gray-out (Σ: 3.0%)	missing	*1.6 %*
			extra	*1.0 %*
			wrong	*0.4 %*
	Widget (Σ: 18.1%)	Behavior (Σ: 14.7%)	missing	*5.1 %*
			extra	*0.9 %*
			wrong	*8.7 %*
		focus (Σ: 3.4%)	initial	*0.9 %*
			implicit	*1.5 %*
			explicit	*1.0 %*
Contents (Σ: 25.1%)	Text (Σ: 15.1%)	design time (Σ: 5.9%)	missing	*1.2 %*
			incomplete	*0.3 %*
			extra	*0.5 %*
			wrong	*3.9 %*
		run time (Σ: 9.2%)	missing	*2.2 %*
			incomplete	*1.1 %*
			extra	*1.0 %*
			wrong	*4.9 %*
	Animation (Σ: 1.8%)	design time (Σ: 0.8%)	missing	*0.4 %*
			extra	*0.1 %*
			wrong	*0.2 %*
			others	*0.1 %*
		run time (Σ: 1.0%)	missing	*0.4 %*
			extra	*0.1 %*
			wrong	*0.3 %*
			others	*0.1 %*
	Symbols & Icons (Σ: 8.2%)	design time (Σ: 2.9%)	missing	*1.5 %*
			extra	*0.2 %*
			wrong	*1.2 %*
		run time (Σ: 5.3%)	missing	*2.2 %*
			extra	*1.0 %*
			wrong	*2.1 %*
Design (Σ: 5.8%)	color	-	-	*1.0 %*
	font	-	-	*0.4 %*
	dimension	-	-	*0.7 %*
	shape	-	-	*0.4 %*
	position	-	-	*2.7 %*
	other	-	-	*0.6 %*

depends on numerous conditions, such as available devices (e.g., radio tuner available, connected mobile phones, etc.), the current environmental conditions (e.g., car is moving faster than 6 km/h), or even previous interactions (e.g., activating route guidance). These conditions affect whether options are displayed but cannot be selected (gray-out mechanism) or whether options are even listed at all. Therefore, two subcategories refer to option provision behavior. The first subclass is *option offer* which summarizes failures that refer to occurrence or order of options. The class is further differentiated as follows:

- *missing*: an option that should be displayed is not visible.
- *wrong*: an option A is displayed instead of option B.
- *extra*: an option is displayed but should not be visible.
- *order*: an option B is listed before option A but should be listed after.

The second option specific failure class contains failures that refer to their *gray-out behavior*, which again is further detailed as follows:

- *missing*: an option should be grayed-out but is available.
- *wrong*: instead of an option A an option B is grayed-out.
- *extra*: an option A is grayed-out but should be available.

The subclass *screen composition* clusters failures related to deficient setup of widgets on screen. Subclasses of this category are:

- *missing*: widgets that are specified are absent.
- *wrong*: the wrong widget is displayed.
- *extra*: an unspecified widget is displayed.

Screen structure failures are distinguished from the *widget behavior* category as follows: the former represents erroneous selection of widgets such as horizontal or vertical lists, whereas the latter clusters failures of widget behavior itself, such as the scrolling logic or widget state change.

3) Screen Transition Failures: As described above, screens represent one special usage context. The failure class *screen transition* clusters failures occurring when those usage contexts change, such as radio, players, or system setup. One indication of a screen transition is that the widget composition and the displayed options are replaced. With Figure 1 and Figure 5, a screen transition is demonstrated: First, the Radio screen is shown; by activating the option "Tel", the context changes to the telephone screen of the infotainment system. Subclasses of this category are

- *missing*: a screen transition that is specified does not take place.
- *wrong*: instead of screen A, screen B is displayed.
- *extra*: a screen transition that is not specified takes place.

4) Pop-up Behavior Failures: In automotive infotainment systems, messages are often overlaid over the regular screen (Pop-up mechanism). Those messages inform users about relevant events or changes of conditions. For example, those messages might state that the car has reached the destination of an active route guidance or that hardware has heated up critically. These messages might be confused with overlaying submenus described above as part of screens. The difference is that pop-up messages do not depend on the current system state and may occur any time, triggered by system conditions or events. Overlaying submenus are only displayed on particular screens and are triggered explicitly by user input. Pop-up behavior subcategories are

- *missing*: the pop-up is not displayed although the respective conditions are active.
- *wrong*: instead of pop-up A, pop-up B is displayed.
- *extra*: pop-up appears although the respective conditions are not active.

Additionally, with the pop-up mechanism the priority system is important: A pop-up with higher priority always has to be displayed on top of pop-ups with lower priority. Those failures are clustered in the subclass *priority*.

B. Contents Failures

The next top-level category is related to contents. The separation criterion is the type of the content: *symbols & icons*, *animations*, or *text*. In Figure 1, a text failure would be if the button for the "Audio" application was labeled incorrectly with "Adio". If the globe symbol in the upper right corner of the screen were a simple square as place-holder, this would be considered a symbol failure. Examples of erroneous animations might be if the focus highlight transition is faster than specified (wrong) or if the overlay menus are not faded in (missing). In this classification, we additionally distinguish content that is known at *design time* (e.g., the labels of available applications) and content that cannot be defined until *runtime* (e.g., displaying the names of available Bluetooth devices). Design time does include localization failures. Although this content depends on the language setting, the particular data is already defined and stored in a database. Characteristic for failures at runtime are patterns that define what data is needed for the content (e.g., title and artist of music on connected media) and how it is displayed (e.g., order, format, etc.). This explicitly includes how content might have to be shortened or reduced. Therefore, content runtime failures are close to the behavior category. As they are strongly related to the respective data to be displayed, we considered this a content category. For each of those content types, the following subclasses are defined:

- *missing*: Content that is specified is not displayed.
- *wrong*: Instead of the content that is specified other content is displayed.
- *extra*: Content that is not specified is displayed.

This category might be confused with the screen structure failure class in the behavior sub-tree. For example, a failure report describing that the second button in the main menu is "Blind Text" instead of "Audio" could be categorized as either a *contents* or an *option provision* failure. If pressing the button still leads to a screen transition to the Audio context, the report is considered as deficient contents. If another context is displayed, for example the Telephone screen, it would be a deficient option provision.

C. Design Failures

The last top-level category clusters reports that describe design failures. This includes

- *color*: e.g., focus color is red instead of orange.
- *font*: e.g., text font is Courier instead of Arial.
- *dimension*: e.g., a button is higher or broader than specified.
- *shape*: e.g., a button should be displayed with rounded instead of sharp edges.
- *position*: e.g., a label of a button is centered instead of left-aligned.

As design failures were often described vaguely, a subcategory for *other* design failures was defined. Ambiguous descriptions were, for example, that wrong arrows, wrong Cyrillic letters, or a wrong clock were observed. As it became obvious early that a low percentage of reports were categorized as design failures, no additional work was done to clarify this category.

VI. Discussion

The requirements defined in Section IV were fulfilled for most failure classes. We intended to cover at least 90% of all defect reports analyzed. Only 7.6% of the reported failures had to be classified as "to be categorized". Furthermore, we intended to limit the percentage of the classes on the lowest level of the hierarchy to 10%. This could be achieved as well: with 9.2%, the largest category was *behavior* - *screen transition* - *wrong*. We intended to allow only 2-5 categories on each hierarchy level. This could not be realized for the design category (6 sub classes). However, due to a very small number of failures classified as design-related (5.8%), we did not consider it necessary to restructure this category.

The requirements further stated that the failure classes have to be disjunctive. Most of the distinction was clear during the classification process. In the available reports, there was no interference between logic and design failures. However, failures regarding design, which is determined by algorithms were not analyzed. For those cases, a new class within the logic sub-tree has to be defined. No ambiguity was noticed in terms of differences between content and design failures.

Most challenges were experienced in differentiating content and logic failures, especially in cases where several failures occurred at once. This was due to the fact that the systems had to be tested as a black-box and only the information displayed on the screen could be accessed. The following scenario exemplifies the key issues: let us assume all 5 buttons in the main menu line illustrated in Figure 1 are labeled as "Blind Text". This is definitely a content text failure. However, it has to be checked whether there are additional failure symptoms such as wrong, missing, or extra options provided, or a failure regarding the order of menu entries. Those additional failures have to be revealed by analyzing other button properties. In this case, it would have been checked which screen transition they trigger. However, several alternatives have to be considered:

- If pressing the second button in the menu line – which should be the Audio button – triggers the transition to the Audio context, no additional failure has to be reported.
- If pressing not the second but the third button triggers the transition to the Audio screen, an additional option provision (order or extra) failure has been revealed.
- If pressing a button labeled Telephone triggers the Audio screen, this could be a screen transition failure or a content text failure.

In the analyzed reports, this distinction was possible due to the given descriptions. However, problems might occur when applying the classification in the future. This is not only an issue with failure classifications, but an issue with reporting failures in general. We recommend bearing the ambiguity of symptoms in mind while testing and reporting. It is essential to provide the information that is needed to differentiate failure symptoms. A detailed classification, such as the one presented in this paper, might help to clarify the exact circumstances even in early phases.

Further, we answered the question raised in this paper what types of failures are frequent in current information systems. The results show that the majority (61.5%) are failures related to behavior. This demonstrates the complex macro and micro behavior in modern infotainment systems. Most of the failure reports are related to missing or wrong individual widget behavior (13.8%) as well as missing or wrong screen transitions (15.0%). The content category is the second largest top-level failure class (25.1%), with erroneous text being the biggest subcategory (15.1%). The majority (9.2%) is not known until runtime. Explanations are (a) that in most infotainment systems, information is mainly displayed textually and (b) that testing texts is easier for human testers than comparing symbols or animations in detail. Very few failures (5.8%) describe erroneous design. One explanation might be that design is hard to test manually. For example, it is a problem to differentiate shades of colors visually. In addition, most design errors are less critical and even might not be recognized by users. Therefore, testing design might not be of high priority to test planners. Hence, this data cannot be seen as definite evidence showing that design failures are indeed this rare. However, we addressed this limitation by analyzing failure reports representing a broad variety of contexts such as testing goals, test personnel, or test environments.

VII. Conclusion and Future Work

In this paper, we answered the question of what types of failures can be found in GUI-based software in the automotive domain today. A failure classification was developed and applied to more than 3,000 failure reports. Ultimately, each related fault concerning the reported failures was fixed during the development process. The reports were created during the development of modern automotive infotainment systems at AUDI, Bosch, and Mercedes-Benz. 61.5% of the reports describe failures related to high- and low-level behavior, 25.1% of the reports describe failures related to contents, and 5.8% of the reports describe failures related to design. We support not only the testers in creating detailed and clear reports, but also the entire GUI development process by pointing out pitfalls leading to gaps between the specification and the implementation. The classification indicates, which aspects need special attention in specification documents and might need to be described more explicitly than is common today. For roles responsible for the implementation of GUI concepts, this work points out aspects that might be ambiguous and require clarification.

Requirements were defined in order to guide the classification process and to avoid categories that are too general or too specific. These requirements proved to be effective for guiding the classes development process. General sections such as those suggested by the ODC extension v5.11 [18] could be avoided. However, as there are classes with less than 1%, the presented classification seems to be over detailed. In most cases, classes follow the missing/extra/wrong pattern suggested by the IEEE classification [7], which was applied with ease. In future applications, those minor classes might be ignored.

In future research, the suggested classification might be scaled by reducing the maximum percentages of lowest-level categories. Thus, some categories have to be differentiated further and additional failure classes have to be defined. Moreover, additional parameters such as "failure criticality", "predicted number of affected users", or "costs for testing" could be added to the classification. Those aspects are not in focus at the current stage and might influence the choice of test strategies significantly. Future failure reports should include information about those aspects. Extension of the failure reports would require intensive collaboration between testers, programmers, requirement engineers, and other participants in the development process. Extended reports together with the failure distribution might enable the derivation of prioritization factors. The usage of existing prioritization approaches, such as the techniques for selecting cost-effective test cases shown by Elbaum [22], is conceivable. One could then focus or prioritize testing on those types of failures that are most critical based on their frequency and these additional parameters. For this purpose, coverage criteria and prioritization techniques are currently being examined to check which of them, if any, can be used for our purposes. This classification could be applied to future automotive infotainment systems to analyze changes of the failure focus.

Another part of future work will be to analyze whether our defined classification scheme and the ascertained distribution of failures could be combined with fault seeding approaches, which are used to measure and predict reliability. One of the most popular fault seeding models is the hypergeometric model by Harlan D. Mills [23]. According to [24] and [25], fault seeding is based on seeding a known number of faults in a software program whose total number of faults is unknown. After testing the software, the comparison of the number of "found seeded faults" and "found indigenous faults" allows estimating the number of remaining faults. In the case of [23], estimation is realized by using the hypergeometric distribution. Related work like [26], [27] and [28] focus on the described principle "with the purpose of simulating the occurrence of real software faults" [29]. Andrews [30] shows that generated mutants are similar to real faults and consequently claims that mutation operators yield trustworthy results. Based on Andrews, the work of [31] is about adding numerous faults for each module by selecting mutation operators simultaneously applied to the source code. This results in a single high-order mutant that represents the faulty version of the system. The approach performs a "1/10 sampling" to limit the number of seeded faults. This means that 10% of the maximum number of the calculated faults – e.g., faults regarding the logical operators or any constants – are seeded into the system.

The knowledge of the classified distribution of faults could improve fault seeding approaches by considering the known fault rates during the seeding process. However, the analyzed reports in our work contain the description of failures and not faults. Due to the fact that the origin of a failure is always a fault and that fault seeding is based on faults and not failures, traceability between faults and failures (through errors) is necessary to obtain the benefits of a known classified distribution of failures. An enabler for these benefits could be failure proximity approaches, which identify failing traces and group traces to the same fault together. [32] regards "two failing traces as similar if they suggest roughly the same fault location" and assumes that collecting failing traces can support developers, respectively testers, in prioritizing and diagnosing faults. In conclusion, our classified distribution of failures could support the effectiveness of fault seeding approaches. Applications will follow.

Acknowledgment

The authors would like to thank Krishna Murthy Murlidhar, Sven Neuendorf, and Jasmin Zieger for their contributions. The research described in this paper was conducted within the project automotiveHMI[1]. The project automotiveHMI is funded by the German Federal Ministry of Economics and Technology under grant number 01MS11007.

References

[1] D. Mauser, A. Klaus, R. Zhang, and L. Duan, "GUI failure analysis and classification for the development of in-vehicle infotainment," in *Proceedings of the Fourth International Conference on Advances in System Testing and Validation Lifecycle*, 2012, pp. 79–84.

[1]http://www.automotive-hmi.org/

[2] B. Robinson and P. Brooks, "An initial study of customer-reported GUI defects," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*. IEEE Computer Society, 2009, pp. 267–274.

[3] S. Gerlach, "Modellgetriebene entwicklung von automotive-hmi-produktlinien." Ph.D. dissertation, AutoUni, Logos Verlag Berlin, 2012, pp. 2–6.

[4] C. Bock, "Model-driven hmi development: Can meta-case tools do the job?" in *Proceedings of 40th Annual Hawaii International Conference on System Sciences (HICSS 2007)*. IEEE Computer Society, 2007, pp. 287b–287b.

[5] L. Duan, "Model-based testing of automotive hmis with consideration for product variability." Ph.D. dissertation, Ludwig-Maximilians-Universität München, 2012, pp. 18–22.

[6] "Audi infotainment system MMI." [Online]. Available: https://www.audi-mediaservices.com [last visited: 25.02.2013]

[7] *IEEE Standard Classification for Software Anomalies*, IEEE Std., Rev. 1044-2009, 1994.

[8] J.-C. Laprie, "Dependability of computer systems: concepts, limits, improvements," in *Proceedings of Sixth International Symposium on Software Reliability Engineering*, Oct 1995, pp. 2–11.

[9] E. Dubrova, "Fault tolerant design: An introduction," Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden, Tech. Rep., 2008.

[10] R. Chillarege, "Orthogonal defect classification," *Handbook of Software Reliability Engineering*, pp. 359–399, 1996.

[11] N. Li, Z. Li, and X. Sun, "Classification of software defect detected by black-box testing: An empirical study," in *Proceedings of Second World Congress on Software Engineering (WCSE)*, vol. 2. IEEE, 2010, pp. 234–240.

[12] J. Børretzen and R. Conradi, "Results and experiences from an empirical study of fault reports in industrial projects," in *Proceedings of the 7th international conference on Product-Focused Software Process Improvement*. Springer-Verlag, 2006, pp. 389–394.

[13] R. Grady, *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., 1992.

[14] B. Beizer, *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.

[15] P. Brooks, B. Robinson, and A. Memon, "An initial characterization of industrial graphical user interface systems," in *Proceedings of International Conference on Software Testing Verification and Validation*, ser. ICST '09. IEEE Computer Society, 2009, pp. 11–20.

[16] M. Kabir, "A fault classification model of modern automotive infotainment system," in *Proceedings of Applied Electronics*, 2009, pp. 145–148.

[17] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring, "Research issues in software fault categorization," *SIGSOFT Software Engineering Notes*, vol. 32, no. 6, pp. 1–8, November 2007.

[18] IBM Research Center for Software Engineering, Extensions to ODC, 2002. [Online]. Available: http://www.research.ibm.com/softeng/SDA/EXTODC.HTM [last visited: 19.02.2013]

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*. Reading, MA: Addison Wesley, 1995.

[20] S. Stoecklin and C. Allen, "Creating a reusable GUI component," *Software: Practice and Experience*, vol. 32, no. 5, pp. 403–416, Apr. 2002.

[21] J. Chen and S. Subramaniam, "Specification-based testing for GUI-based applications," *Software Quality Journal*, vol. 10, pp. 205–224, 2002.

[22] S. Elbaum, G. Rothermel, S. K, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, September 2004.

[23] H. Mills, "On the statistical validation of computer programs," Federal Systems Division, IBM, Report FSC-72-6015, 1972.

[24] A. L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1411–1423, 1985.

[25] G. Schick and R. Wolverton, "An analysis of competing software reliability models," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 2, pp. 104–120, March 1978.

[26] M. J. Harrold, A. J. Offutt, and K. Tewary, "An approach to fault modeling and fault seeding using the program dependence graph," *The Journal of Systems and Software*, vol. 36, no. 3, pp. 273–296, March 1997.

[27] C. Artho, A. Biere, and S. Honiden, "Enforcer - efficient failure injection," in *Proceedings of the 14th international conference on Formal Methods*. Springer-Verlag, 2006, pp. 412–427.

[28] J. Voas, G. McGraw, L. Kassab, and L. Voas, "A crystal ball for software liability," *Computer*, vol. 30, pp. 29–36, June 1997.

[29] A. Marchetto, F. Ricca, and P. Tonella, "Empirical validation of a web fault taxonomy and its usage for fault seeding," in *Proceedings of 9th IEEE International Workshop on Web Site Evolution (WSE 2007)*, Oct. 2007, pp. 31–38.

[30] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 402–411.

[31] F. Belli, M. Beyazit, and N. Güler, "Event-oriented, model-based GUI testing and reliability assessment - approach and case study," *Advances in Computers*, vol. 85, pp. 277–326, 2012.

[32] C. Liu and J. Han, “Failure proximity: A fault localization-based approach,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 46–56.