Dominik Holling

Defect-based Quality Assurance with Defect Models

Dissertation

Technische Universität München

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Defect-based Quality Assurance with Defect Models

Dominik Holling

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Bernd Brügge, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Alexander Pretschner
2.	Univ.-Prof. Dr. Lionel Briand, Universität Luxemburg

Die Dissertation wurde am 31.05.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.09.2016 angenommen.

Acknowledgements

My gratitude goes to

- Prof. Pretschner for having me in your research group for the last seven years. Starting with my bachelor's thesis, you have provided your continuous support and helped me grow personally throughout my master and particularly my Ph.D. student years. I treasure our invaluable discussions and your guidance of my research and industry endeavors. I admire your honesty, fairness, righteousness and desire to advance (computer) science. I aim to do my part in the form of this thesis and hope to continue our rewarding discussions in the future.

- Prof. Briand for his feedback on my research and my thesis as well as the fruitful collaboration between the research groups in Luxembourg and Munich.

- ITK Engineering for having been my industrial partner throughout my Ph.D. research. Thank you for providing industrial problems, many invaluable hours of discussion with employees and evaluation systems to perform scientific studies. Matthias Gemmar and Bernd Holzmüller in particular always have had a deep interest in my topic and were able to provide precious feedback on the industrial applicability. ITK Engineering also gave me the opportunity to present my work at various industry meetings, demonstrating their belief in knowledge transfer between academia and industry.

- Tobias Wüchner for the endless discussions about our research while having a glass of wine at our shared apartment for the last five years.

- My colleagues at the chair of software engineering for being excellent sparring partners for ideas. Your encouragement, criticism and support have helped me excel in my research and yielded a lot of fun at work. Your proofreading of my various drafts has improved their quality essentially and I am proud to call you friends in addition to colleagues.

- My family and friends for your unparalleled moral support in my research and the creation of this thesis. I am hugely grateful to the faith you have put in me over the years. My greatest gratitude goes to Hannah as the provider of my inspiration and strength. Without your never ending support and patience, this work would not have been possible.

Zusammenfassung

Beim Durchführen von Qualitätssicherungsaktivitäten sind Software-ingenieure mit häufigen und wiederkehrenden Fehlern konfrontiert. Zur Detektion dieser Fehler verwenden diese typischerweise entweder individuelles Wissen und Erfahrung oder Testselektionsstrategien mit inhärentem Wissen und Erfahrung, um Testfälle zu erstellen. Das Problem bei einer derartigen Testfallerstellung ist die Abhängigkeit der Testfälle vom implizit verwendeten individuellen oder inhärenten Wissen. Allerdings gibt es zu dieser Art der Fehlerdetektion anekdotenhafte Evidenz bezüglich ihrer Effektivität. Der Kernbeitrag dieser Arbeit ist ein systematischer Ansatz der fehlerbasierten Qualitätssicherung mit Fehlermodellen, der implizit verwendetes Fehlerwissen und -erfahrung in formalen Fehlermodellen erfasst. Durch Operationalisierung können (semi)-automatische Testfall- / Checklistengeneratoren abgeleitet werden, die exakt die erfassten Fehler effektiv und effizient detektieren.

Das Fehlermodelllebenszyklusframework strukturiert Aktivitäten für die Integration von fehlerbasierter Qualitätssicherung mit Fehlermodellen in bereits existierende Qualitätssicherungsprozesse. Das Lebensyklusmodellframework enthält Aktivitäten, um implizites Fehlerwissen zu erheben und klassifizieren. Als Teil der Methodikanwendung werden die Fehler dann in Fehlermodellen beschrieben und diese Fehlermodelle operationalisiert. Schlussendlich werden die Operationalisierungen bewertet und durch einen unterstützenden Prozess gewartet.

Um die definierte fehlerbasierte Qualitätssicherungsmethodik und die Strukturierung durch das Fehlermodelllebenszyklusframework umfassend zu bewerten werden Instanziierungen der Aktivitäten Erhebung und Klassifizierung sowie der Beschreibung und Operationalisierung vorgestellt und evaluiert. Die Instanziierung der Erhebung und Klassifizierung ist die kontextunabhängige Methode DELICLA. Basierend auf den Resultaten von DELICLA werden Beschreibungen und Operationalisierungen von Fehlermodellen auf allen Testebenen (8Cage, OUTFIT und Controller Tester) vorgestellt und bezüglich Effektivität und Effizienz evaluiert. Diese Beschreibungen und Operationalisierungen stellen den zweiten Kernbeitrag dar. Zusätzlich können anhand dieser Evaluierungen generische Bewertungskriterien für Operationalisierungen von Fehlermodellen und zudem ein Framework für deren Wartung erstellt werden.

Abstract

When performing quality assurance, software engineers are confronted with common and recurring defects. To detect these defects, they typically exercise their knowledge and experience to create test cases or use test selection strategies encapsulating knowledge in experience. For such test selection, there is at least anecdotal evidence of its effectiveness. The problem is its usage of tacit knowledge leading to engineer-dependent test cases or test cases lacking a rationale as to why exactly they are effective. This thesis proposes a systematic and (semi-)automatic approach using defect models for defect-based quality assurance as its major contribution. By capturing defect knowledge and experience of software engineers or inherent to test selection strategies in defect models, they are made explicit and described formally. Operationalizing defect models yields (semi-)automatic test case / check list generators directly targeting the described defects for their effective and efficient detection.

The defect models are accompanied by a defect model lifecycle framework to structure the integration of defect-based quality assurance into existing quality assurance processes. The lifecycle framework first contains activities to capture the tacit defect knowledge and experience by eliciting and classifying to arrive at an explicit library of common and recurring defects. The activities of formal description of defects in defect models and the operationalization of the defect models represent the method application activities actually leading to the (semi-)automatic detection of the described defects. Finally, the activities of assessment and maintenance enable the evaluation of effectiveness and efficiency in a continuous support process.

To comprehensively assess defect-based quality assurance based on defect models, we provide instantiations of the elicitation and classification as well as description and operationalization activities in the lifecycle framework. For the elicitation and classification activities, we present and evaluate the context-independent instantiation called DELICLA. Based on the results of DELICLA, we describe and operationalize defect models on all levels of testing (i.e. 8Cage, OUTFIT and Controller Tester) and demonstrate their effectiveness and efficiency in detecting the described defects. These descriptions and operationalizations yield the second major contribution. In addition to the activity instantiations, we give rise to generic assessment criteria for operationalizations and provide a framework for the maintenance of defect models.

Contents

1

Introduction

When developing software, software engineers are prone to make the same mistakes repeatedly leading to common and recurring defects in the system they develop. Performing as field study to gather evidence for these common and recurring defects multiple industries (see Chapter 4), we even found them to be present across project, organizational and even domain contexts. We treated defects as common, if their appearance frequency in the study was higher than other defects and, therefore, they had a higher likelihood to be present in the systems. Analogously, recurring refers to them not only appearing in one version of the system or one system, but occur in different version of the same system and among different systems.

In the field study, we also examined how these common and recurring defects are detected and found some test cases to be created based on personal knowledge and experience (1) and re-using encapsulated knowledge and experience in existing test selection techniques (2).

To directly target the common and recurring defects, software engineers typically use error guessing or explorative testing (see Section 2.1.2) for the selection of test cases. For error guessing and exploratory testing, there exists some guidance (see tours in Section 2.1.2), but no clear test selection strategy. These techniques rely on the knowledge and experience as well as understanding by the system of the software engineer (1). Thus, they lead to engineer-dependent test cases with varying defect-finding ability and, therefore, effectiveness. Furthermore, these techniques may sometimes be naturally applied by software testers without being required and/or documented.

In industry, limit testing (also referred to as boundary value analysis [122]) is one technique often used for test selection as there is at least anecdotal evidence of its cost-effectiveness. “Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not” according to Myers and Sandler [117]. Lewis explains the higher payoff by stating defects “tend to congregate at the boundaries. Focusing testing in these areas increases the probability

of detecting" [98] them. Limit testing explicitly uses knowledge and experience of past defects related to relational operators and selects tests at the boundaries of the specified input domain of the system (2). Since defects related to relational operators appear to be common and recurring, employing limit testing yields test cases targeting these common and recurring defects. For this reason, it is typically included in test plans and catered to by test management. Apart from limit testing, there are several other input domain-based techniques (see Section 2.1.2) explicitly using knowledge and experience of past defects (e.g. combinatorial testing) with at least anecdotal evidence of their effectiveness.

Combining these two approaches to test case selection, we see a high potential in defining a systematic and comprehensive approach for test case selection based on knowledge and experience of software engineers and encapsulated in existing test techniques. This approach selects test cases to directly target defects based on captured knowledge and experience or by examining and operationalizing existing defect-based techniques. When designing such an approach the questions of capturing the involved knowledge and experience of software engineers as well as existing defect-based techniques and its operationalization are to be answered.

To capture the knowledge and experience employed in the creation of test cases based by software testers, knowledge management in software engineering is a good starting point. Knowledge management is a cross-cutting discipline in software engineering in order to mitigate such knowledge and experience gaps and to support organizational learning [134]. When performing error guessing or exploratory testing, software engineers use tacit knowledge and "are not fully aware of what they know" [134]. They assume that the knowledge and experience used are common and other software engineers to find the same/similar defects. However, this is not the case and knowledge management enables the dissemination of knowledge in four activities [134]:

1. Acquiring (new) knowledge
2. Transforming knowledge from tacit or implicit into explicit knowledge
3. Systematically storing, disseminating and evaluating knowledge
4. Applying knowledge in new situations

To capture the knowledge and experience contained in existing techniques, their test case selection strategy must be analyzed. For this detailed analysis, the aforementioned umbrella term defect has to be refined into fault and failure, which are defined according to Laprie et al. [93] (see further Section 2.1.1). While a failure is a deviation of expected and actual behavior, a fault is the actual reason / mistake causing the deviation. An existing technique for the creation of test cases in software testing based on the underlying faults has been described by Morell [116] in 1990 and is called

fault-based testing. Fault-based testing aims to “demonstrate that prescribed faults are not in the program” [116]. Morell argues “that every correct program execution contains information that proves the program could not have contained particular faults” [116]. In more general terms, fault-based testing increases the confidence of certain faults not to be in the program or gives evidence of them to be present.

The definition of Morell inherently challenges the definition of a good test case provided by the pertinent literature [17, 117]. Since the early beginnings of software testing, a test case was defined useless, if “it does not find a fault” [117]. Conversely, this means that a good test case detects a fault. Extending this argumentation, there would be no good test cases for a correct system. However, one would like to have confidence in the correct functionality of system according to Morell. Thus, a better definition of a good test case is provided by Pretschner et al. [128, 129]. They define a good test case as one that “finds a potential fault with good cost-effectiveness” (see Chapter 3 for a detailed discussion). This definition is highly abstract and not directly operationalizable. Fault-based test case selection techniques inherently seem to select good test cases. Unfortunately, fault-based testing is hard to operationalize as the faults to target must be known prior to testing. However, basing the selection on the knowledge and experience of software engineers or existing techniques (e.g. limit testing and combinatorial testing) targeting common and recurring faults has the potential to select good test cases.

While fault-based testing directly targets certain underlying faults, it can be argued that limit testing does not directly target faults, but aims to exploit a certain set of faults to provoke failures. Given a failure provoked by a test cases using limit testing, only fault localization reveals the exact relational operator-related fault targeted. Thus, there must be a distinction between test case selection techniques directly targeting (types of) faults and methods to provoke failures. To make a clear distinction, we define a method to provoke failures based on the exploitation of a certain set of faults as failure-based testing. This is complementary to fault-based testing, where the fault is directly known after a failure occurs. The proposed approach must encompass both fault-based and failure-based testing and we re-use the umbrella term of defect and define defect-based testing to be the umbrella term for both. Since not only software testing has techniques to directly target faults, we define defect-based quality assurance as defect-based testing, static analysis and review/inspection. Note that, this term only includes analytical quality assurance as constructive quality assurance does not aim at defect detection, but defect prevention.

The major advantage of software testing over reviews/inspections is its ability to be automated. Automated testing systems used in test automation include “technologies that support automatic generation and/or execution of tests for unit, integration, functional, performance, and load testing” [74]. Test automation in software testing in the form of test case execution can completely be automated as testing can be performed whenever the system is changed and is seen as cost-effective [83]. In

practice, test automation is currently used for test execution and automatic adaption of harnesses, interfaces and test environments. Unfortunately, it is rarely used for the generation of test cases. The reason is the low defect detection rate of completely automatic tools, while they require great computational resources [136]. However, combining defect-based testing with automatic test case generation and re-use existing test automation has a high potential to automatically create good test cases. However, these test cases must be derived in a cost-effective and scalable way, which may not be possible fully automatically. Thus, semi-automatic test case generation also suffices. On the review/inspection side, check lists could potentially also be (semi-)automatically generated.

In this thesis, we present a systematic and comprehensive approach to defect-based quality assurance. It consists of a generic formal containment vessel for defect knowledge called defect model with the capability of (semi-)automatic instantiation in software testing. To describe and operationalize defect models, we instantiate the four activities of knowledge management. To this end, the knowledge and experience of software testers and existing test selection techniques concerning past faults must be made (1) explicit by systematic description, (2) evaluated for its quality, (3) stored for application and (4) operationalized (semi-)automatically in the next project(s). Ultimately, this approach must be able to describe faults and failures for defect-based quality assurance and allow their (semi-)automatic operationalization for the creation of good test cases / review check lists.

1.1 Problem

A good test case finds a potential fault with good cost-effectiveness [128, 129]. This definition is abstract and not directly operationalizable. However, there exist test case selection techniques catering to this definition by selecting test cases directly targeting certain faults. Such techniques use the knowledge and experience either ad-hoc employed by the software engineers or systematically employed in existing test selection strategies (e.g. limit and combinatorial testing). Both ways of using knowledge and experience have at least anecdotal evidence of their effectiveness and are commonly used when testing software. For the usage of knowledge and experience of individual engineers, a field study of common and recurring defects across multiple industries (see Chapter 4) revealed defect-based testing to be present with subjects estimating 20% of test cases to be defect-based. However, test selection techniques based on the knowledge and experience of individual engineers lead to engineer-dependent test cases. The fault-finding ability and, therefore, effectiveness of these tests may vary. When using systematic test selection strategies targeting defects, the applicability of the strategy depends on the defects it detects and their likelihood of being in the system. For limit testing (also called boundary value analysis), Myers and Sandler [117] note a higher payoff than other tests and Lewis [98] explains the higher payoff by a higher

probability to detect a fault. This indicates a general cost-effectiveness of limit testing, which may not generalize in all cases and also for other test techniques encapsulating knowledge and experience. Thus, the encapsulated knowledge and experience must be examined to predict the likelihood of the targeted faults to be in the system and its cost-effectiveness. In sum, a systematic and comprehensive approach for test case selection based on knowledge and experience is currently missing. Such an approach would select test cases to directly target faults and failures and thereby have the ability to produce good test cases.

To investigate the employed knowledge and experience of software engineers, a field study gave the insight of defect-based quality assurance to be applied ad-hoc and unpredictably. Software engineers often performed manual tests and reviews of "what typically goes wrong" naturally and even if it was not part of the test plan. Although reportedly effective, this approach is problematic since it

(i) depends on the knowledge and experience of the respective software engineer,

(ii) may be costly due to manual test case creation or late defect detection and

(iii) may find the repeated mistakes of developers without a defined quality assurance process.

To investigate the encapsulated knowledge and experience in existing defect-based quality assurance techniques, the encapsulated knowledge and experience must be made explicit. Defect-based testing techniques (e.g. limit testing) are used in practice [40] and have at least anecdotal evidence of effectiveness including the ability to derive good test cases. However, their employment is problematic since it

(i) depends on the encapsulated knowledge and experience of the respective software engineer,

(ii) the cost-effectiveness to create the test cases and

(iii) the likelihood of the target defects to be contained in the system.

This thesis investigates how to capture the tacit knowledge and experience employed by the software engineers and encapsulated in existing defect-based quality assurance techniques (i), how to operationalize it to (semi-automatically) detect the detected defects (if not already operationalized) (ii) and how to integrate it into existing quality assurance (iii).

To capture the knowledge and experience employed by the software engineers and encapsulated in existing defect-based quality assurance techniques (i), knowledge management recommends the use of repositories for the storage of explicit knowledge (e.g. mind maps, use cases, glossaries or models) in the third activity of knowledge management [134]. Such repositories can then be accessed by software engineers in order to retrieve the required knowledge and enable its dissemination. Using the

disseminated knowledge leads to its manual operationalization and achieves the goal of being independent of personal knowledge and experience.

To perform (semi-)automatic operationalization of the captured knowledge and experience (ii), the requirements for the knowledge repository need to be extended. Automation of any kind requires the understandability of (part of) the repositories' contents by machines. As natural language understanding by machines is currently limited and an active field of research, the knowledge to be operationalized must be formalized to enable automation. Since scalability and cost-effectiveness may be involved when operationalizing the knowledge, (semi-)automatic operationalizations are also sufficient. A term used throughout literature related to fault knowledge is the term 'fault model'. It typically describes the creation of test cases targeting a specific (set of) fault(s). Although researchers have been describing fault models, the term has never been explicitly defined. Publications typically state "Our fault model is: fault X is present" or "We test our system in the following way to reveal the fault". While the first statement relates to the definition of fault-based testing [116], the second statement relates to methods to provoke failures or failure-based testing. Thus, the explicit and formal definition of defect models containing both fault and failure models yields a starting point to fill the repository (i) with knowledge capable of (semi-)automatic operationalization and its respective operationalization (ii).

The creation of the repository as defined above (i and ii) only yields a containment vessel with (semi-)automatic operationalizability. The problem of its integration into quality assurance (iii) persists and concerns the addition of knowledge (iiia), the transformation of knowledge to (semi-)automatically operationalizable knowledge and its respective operationalization (iiib) and the assessment of the operationalization as well as the maintenance of the knowledge repository (iiic). The addition of knowledge (iiia) regards the first and second activity of knowledge management concerned with the acquisition and transformation of knowledge from implicit to explicit. In the context of defect knowledge, the employed and encapsulated tacit knowledge of the software engineer and test selection strategy must be extracted and characterized. The transformation of this knowledge (iiib) to (semi-)automatically operationalizable knowledge represents a systematical storage in the third activity of knowledge management. This problem requires the careful evaluation of knowledge to be transformed, as formal descriptions are typically time-consuming, but yield the advantage of precision. The operationalizations represent the dissemination and application of the knowledge in new situations in the third and fourth activity respectively. The major problem with the operationalization of the knowledge is the context of operationalization as systems are different. Thus, the operationalizations must be assessed (iiic) representing the evaluation in the third activity in knowledge management. This assessment must ensure the (semi-)automatically created results are the same/similar to the manual results of the respective software engineer and created in a cost-effective manner. Maintenance of knowledge is not addressed by knowledge management directly, but is

required as changes in technology, organizational or domain affect the effectiveness of the approach.

In sum, there are the problems of how to capture the tacit knowledge and experience employed by the software engineers and encapsulated in existing defect-based quality assurance techniques (i), how to operationalize it to (semi-automatically) detect the common and recurring defects (if not already operationalized) (ii) and how to integrate it into existing quality assurance (iii). Addressing these problems allows going from an ad-hoc and unpredictable approach when employing knowledge and experience of software engineers or encapsulated in test selection strategies to a systematic and (semi-)automatic approach in defect-based quality assurance.

1.2 Solution

In order to address the problems above, this thesis presents a comprehensive and systematic approach to defect-based quality assurance based on defect models. We capture the knowledge and experience of software engineers or encapsulated in test selection strategies (addresses i). Describing it in formal defect models and operationalizing them yields effective, efficient and (semi-)automatic detection of the captured and described defects (addresses ii). To manage the repository of defect knowledge, defect models and operationalizations in an organization, the defect model lifecycle framework (addresses iii) provides activities for the extraction and characterization of defect knowledge, its description, operationalization and assessment as well as defect model maintenance. Thus, the defect model lifecycle framework gives a structure for the integration of defect-based quality assurance based on defect models in practice.

Capturing common and recurring defects in a defect repository (see Chapter 4) enables the strategic decision making to (semi-)automatically detect them or use other measures of quality assurance for their detection/prevention. For their (semi-)automatic detection, we introduce a formal generic defect model (see Chapter 3). The generic defect model is an abstraction of existing approaches involving fault models in literature and practice. It is able to describe faults as syntactic differences between desired and actual artifacts in fault models and failures as differences between desired and actual behavior in failure models. The operationalizations of defect models use this description and yield (semi-)automatic test case generators, which target the detection of the described defects. These test case generators have the ability to produce good test cases in case the defects they detect are likely to be present in the system.

To demonstrate the effectiveness and efficiency of the operationalization derived from the generic defect models, three operationalizations are created. While the description of defect models is generic, the (semi-)automatic operationalizations of defect models have a context characterized by variation points. These are the domain, test level and application of the operationalization. Variation points are inherent to the operationalization and are typically determined prior to implementation. The

variation point of domain is roughly defined to allow a context-specific instantiation fitting the project or organizational context. Examples of the domain are cyber-physical, embedded or IT systems. If the context requires further focus, the domain can be automotive, aerospace or medical for organizations focused on cyber-physical systems or finance, travel or government in the context of organizations focused on IT systems. For the refinement of test levels, defect models re-use the well-known distinction according to the target of the test. This distinction defines three test stages: unit, integration and system testing. Note that, although they are called test levels, static analysis may also be performed on each of these levels. At the lowest level, the variation point of application tailors the application area of an operationalization to a specific application. Thus, application-specific operationalizations are very limited in terms of the number of applicable systems and typically required only if the respective context is solely concerned with the development of one application. The number of operationalizations covering all possible instances of the variation points is unfeasibly large. Thus, we create a representative set of operationalizations in the domain of Matlab Simulink systems. Matlab Simulink is commonly used in the design and implementation of cyber-physical systems in the automotive, aerospace and medical domain. Each of the three operationalizations is application-independent, but on a distinctive test level. Thus, we are able to demonstrate effectiveness and efficiency in software testing and this domain.

The first operationalization on the unit testing level is 8Cage (see Chapter 5). 8Cage operationalizes an extensive library of fault models and a failure model for Matlab Simulink/Stateflow systems, which do not depend on the specification of the system [72]. The captured faults are: division by zero, over-/underflow, comparison and rounding, Stateflow faults and loss of precision. Particularly, faults in the first two categories lead to run time failures, which in turn can lead to violations of functional and non-functional requirements (e.g. safety). In addition, 8Cage allows to specify developer assumptions as fault models. The failures captured by 8Cage are signal range violations. Signal range violations are common in Matlab Simulink/Stateflow systems and lead to a variety of subsequent failures potentially violating functional and non-functional requirements. To create defect-based test cases, 8Cage performs a static check of the system to detect smells (potential faults similar to Lint) and subsequently aims to find evidence for the smell to be an actual fault. This evidence is provided by generating a test case using symbolic execution and executing it using the inherent robustness oracle to form verdicts. In sum, 8Cage enables the early detection of common and recurring faults in Matlab Simulink. This leads to expert relief concerning the static analysis, where these faults are typically detected. In general, the abstract concepts of 8Cage are portable to other programming languages/paradigms/methods, but current technology (e.g. symbolic execution) has not yet reached the required level of maturity yet to perform this porting.

The second operationalization on the integration test level is OUTFIT (see Chapter 6). OUTFIT operationalizes two failure models on the integration testing level for the detection of superfluous/missing functionality and the explicit test of central/upstream fault handling. OUTFIT integrates two directly connected components (i.e. outputs of the first are input of the second) and uses high coverage unit test cases to cover the integrated system. The test cases can either be re-used from previous unit tests or automatically created using symbolic execution. A manual inspection of structural coverage in each of the components produced by the tests then reveals the targeted faults. The selection of the components can be arbitrary for the superfluous/missing functionality failure model. For the explicit test of central/upstream fault handling, an upstream fault handler is chosen as second component to exercise all possible fault modes.

The third operationalization on the system level is Controller Tester (see Chapter 7). Controller Tester directly re-uses one failure model and its operationalization from literature. It abstracts the existing failure model and adds four additional ones to test control systems. From existing literature it is known that control systems typically have the requirements to be stable, respond within a certain time, have a minimal overshoot and stay within their specified bounds. When performing quality assurance activities, control system engineers aim to find worst-case scenarios which violate these requirements. Their test cases typically include special stimulations of the systems and external disturbances. However, the control system engineers typically choose only few test cases in these scenarios. Controller Tester uses the operational space of the control system as a search space and slices this space into scenarios to find the worst-case behavior of the control system in each of the scenarios. These search spaces are randomly explored and each exploration is ranked by using a formalized form of the requirements. Using the rank, a heat map is created to give an overall impression of requirement compliance and retrieve the worst case.

To give a structure to the planning of employment, employment and controlling of employment of defect-based quality assurance based on defect models, a defect model lifecycle framework is introduced. The lifecycle framework structures the systematic integration into existing quality assurance processes by providing activities tailorable to organizational/project contexts. The lifecycle framework consists of planning, application and controlling steps. Planning encompasses the elicitation and classification of common and recurring defects to enable strategic decision making about the application of defect models. Application includes the description and operationalization of defect models and represents the methodology presented in this thesis. In controlling, the suitability of the operationalizations is assessed and maintained (see Chapter 8). This enables to establish requirements for the proactive anticipation of changes in technology, organization or domain, including their impact on the employed defect models.

1.3 Contributions

The contribution of this thesis is the operationalization of the definition of a good test case / check list by defect-based quality assurance with defect models.

The formal definition of defects and defect models enabling systematic re-use of defect knowledge and (semi-automatic) defect-based quality assurance yields the first major contribution of this thesis. By investigating the encapsulated knowledge and experience in existing defect-based quality assurance techniques in a systematic literature survey, we are able to grasp the notion of defect-based quality assurance and generalize it in a generic defect model for quality assurance. As evaluation, we show that existing fault and failure models are instantiations of the generic defect model. We are the first to give the characterization of a fault and combine it with that of a failure to describe them in programming language/paradigm/methodology independent defect models. By operationalization, we are able to use them (semi-)automatically in quality assurance to generate defect-based test cases. Thus, we are able to not only capture defect knowledge, but to use it in tools for the detection of the captured defects. Categorizing these operationalizations, we are able to generalize them to generic operationalization scenarios. Thus, any defect model mappable to the generic defect model and any operationalization mappable to the generic operationalization scenarios then inherently creates defect-based (and potentially good) test cases. Even if the defect knowledge is not described in defect models and operationalized, quality assurance benefits from these defects as they may be detected in static analysis or prevented by the usage of standards and process improvement in constructive quality assurance.

The operationalization of defect models must not only yield good test cases, but must be able to produce them with good cost-effectiveness. As cost-effectiveness can only be determined by the usage of defect models in industrial projects after the introduction of defect-based quality assurance and the creation of industry-ready operationalizations, we evaluate effectiveness, efficiency and reproducibility of the operationalization to assess the quality of their created test cases. Thus, the demonstration of effectiveness, efficiency and reproducibility of the operationalizations in defect-based testing with defect models is the second major contribution of this work. To the best of our knowledge, we are the first to create three explicit operationalizations based on the generic defect model on the unit (8Cage), integration (OUTFIT) and system (Controller Tester) testing level. All operationalizations aim to (semi-)automatically detect defects as early as possible in the development process. They are able to perform this detection without a system-specific oracle by re-using existing oracles of generic requirements (e.g. robustness oracle of “no crash or exception”). All operationalizations are evaluated and demonstrated effectiveness, efficiency and reproducibility. In our context, effectiveness denotes the fault detection ability of the operationalization in real-world systems. Efficiency measures the resources (e.g. time) used to detect the aforementioned faults. Since all operationalizations use some form

of randomness to detect the defects, reproducibility measures the probability of an operationalization to detect a fault within several executions.

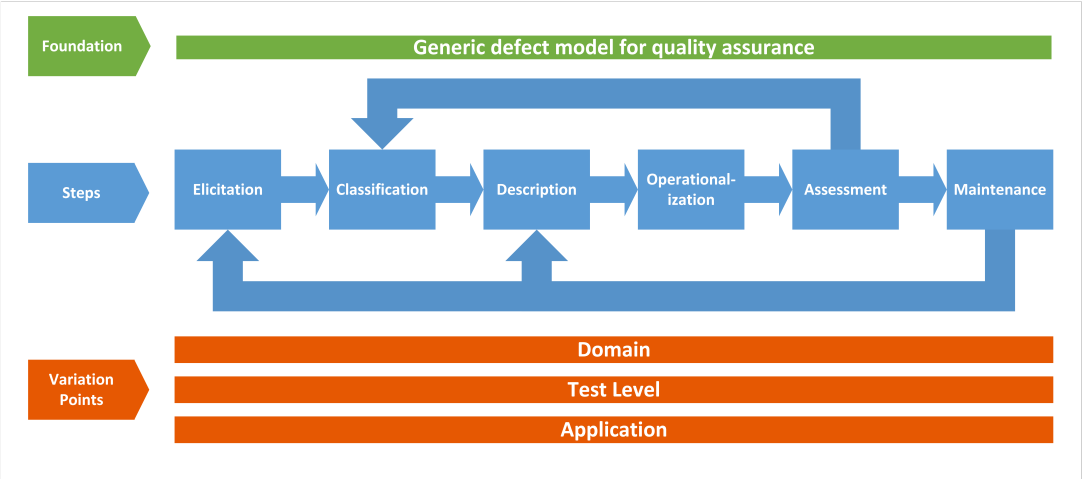
8Cage is the first explicit unit level operationalization of defect models. There exist other tools to detect the run time failures detected by 8Cage using abstract interpretations (e.g. Polyspace and Astrée). However, these tools typically have a run time spanning multiple days and require a manual fault investigation effort by an expert of multiple days. 8Cage is lightweight and performs a best-effort detection of run time failures within hours directly delivering evidence in the form of a test case to software engineers. To detect parts of the code potentially causing run time failures, there exist smell finding tools such as lint. However, these tools do not provide evidence for an actual fault to be present as 8Cage does, but rather return potential faults. There is typically a plethora of potential faults returned, which have to be manually investigated by the software engineers. The Design Verifier of Matlab uses the fault models of some run time failures implicitly and is limited to their automatic detection, but does not use their explicit description in fault models making it not extendable to other faults as 8Cage. Tools such as TPT and Reactis do not allow the automatic creation of defect-based test cases, but require their manual specification.

OUTFIT is the first explicit integration level operationalization of defect models. There exist approaches focusing on the creation of integration test cases targeting functional defects in the interaction of components partially in a program/domain/-paradigm specific manner. There also exist coverage and coupling-based approaches to integration testing that allow the automatic creation of test cases. However, both do not allow the (semi-)automatic creation of defect-based integration test cases targeting particular defects in the system without requiring any specification as OUTFIT does. TPT, Reactis or slUnit exist to perform functional software integration testing of embedded systems and could be used in OUTFIT for the generation of the coverage test suites.

Controller Tester is the first explicit system level operationalization of defect models. It borrows its methodology and one defect model directly from an automated testing of continuous controller approach in the literature. It aims to generalize and extend the approach by using failure models derived from the violation of quality criteria. Thereby, it automatically tests control systems in a variety of scenarios deemed relevant by several control system engineers as well as the predominant control system literature. Again, Design Verifier, TPT and Reactis are able to perform software system testing, but do not allow the automatic creation of test cases targeting defects in control systems by using defect models. However, they provide the ability to perform random or coverage-based testing, which can be manually combined with the respective quality criteria to represent part of the automated testing of continuous controllers approach.

The defect model lifecycle framework including its instantiation represent the third major contribution of this work. By enclosing the methodology within supporting processes, the lifecycle framework yields an instantiation of knowledge management

Figure 1.1: Overview of this thesis



and contributes the structure for a seamless integration into existing quality assurance processes. It structures the systematic defect-based quality assurance based on defect models by providing a framework for planning, method application and controlling activities including a description of their instantiation and tailoring. Particularly, it proposes a way to capture common and recurring defects in practice, their description in defect models and the operationalization to arrive at a repository of defects, defect models and defect model operationalizations. The planning in the lifecycle framework is mapped to the first two activities of knowledge and experience management as it acquires and transforms defect knowledge into explicit knowledge. The defect model methodology application then describes the defects in defect models and operationalizes the defect knowledge covering parts of the third and the fourth activity of knowledge and experience management. The controlling covers the evaluation part of the third activity of knowledge and experience management by assessment and performs maintenance in addition.

To elicit and classify common and recurring defects in practice for the creation of defect models, we provide a context-insensitive qualitative interview method (DELICLA) including a field study demonstrating its suitability and operationalizability of the results. In contrast to existing defect classification approaches, DELICLA deliberately chooses to employ a minimalistic/basic defect taxonomy to stay flexible for seamless adaptation to specific contexts and domains. This lightweight taxonomy enables the approach to be in tune with the expectations/prerequisites of our project partners. Thus, we are not generalizing our taxonomy to be “independent of the specifics of a product or organization” [32], but rather require adaptability to context. However, our taxonomy can be mapped to ODC.

1.4 Organization

This thesis is structured according to the defect model lifecycle framework [69] for systematic and (semi-)automatic defect-based quality assurance based on defect models shown in Figure 1.1. To give an overview of existing work and define the basics, Chapter 2 provides an introduction to testing, symbolic execution and the development of embedded systems using Matlab by Mathworks. The basis of the defect model lifecycle framework is the generic defect model for quality assurance (top section of the figure) in Chapter 3. This generic defect model represents an abstraction of defect-based quality assurance in literature and practice. Including its operationalizations, these represent a storage format in a defect knowledge repository that allows systematic and (semi-)automatic re-use of defect knowledge. Existing work is gathered using a systematic literature survey and shown to be an instance of the generic defect model.

The process to guide the integration of defect-based quality assurance based on defect models into existing quality assurance processes (middle section of the figure) has three major steps. In the first step (planning), defects are elicited and classified using qualitative interviews in Chapter 4 (activity 1 and 2 of knowledge management). In the second step (methodology application), three descriptions and operationalizations of defect models on the unit/integration (8Cage), integration (OUTFIT) and integration/system (Controller Tester) testing level are described in Chapters 5, 6 and 7 respectively (step 3 and 4 in knowledge management). In the third step (controlling), Chapter 8 proposes requirements for defect model assessment and maintenance of defect models. Both assessment and maintenance are the starting points of respective feedback loops. In case the operationalization is unable to detect the described defects or if the wrong defects are detected, the feedback loop allows going back to the classification, description and operationalization. In maintenance, issues such as defect models losing their effectiveness due to team/technology changes or organizational learning are discussed. This includes the re-evaluation of defect models and/or the creation of defect models for new technologies as it can be triggered by maintenance. Chapter 9 draws a conclusion, lists contributions and future work.

The method application step is accompanied by the variation points (lower section of the figure). These variation points are able to fundamentally categorize defect models by the domain they are applicable to, their test level and make the fine distinction between defect models for specific applications. As an example, 8Cage in Chapter 5 is applicable to the broad domain of embedded systems developed in Matlab Simulink on the unit test level, while ControllerTester in Chapter 7 is applicable to control system applications for embedded systems developed in Matlab Simulink on the system testing level.

2

Background

This chapter contains the background of this thesis. It gives a detailed introduction to quality assurance, symbolic execution and the development of embedded systems in Matlab Simulink (based on [72]). The introduction to quality assurance serves for the definition of terms used throughout this thesis. It also positions defect models and defect-based quality assurance in the field of quality assurance and specifically in fault-based test selection techniques. Related work specific to and generalized by the generic defect model including the defect-based quality assurance techniques it abstracts is located in Chapter 3 and specifically in Section 3.2. Quality assurance is divided into analytical and constructive quality assurance, where analytical quality assurance is separated into software testing and static analysis. As for the software testing part, this includes the definition of the basic terms of software testing, test levels and the relation to test case selection strategies. As for static analysis, basic definition of terms and methods are introduced. Symbolic execution is a key technology in the operationalization 8Cage (in Chapter 5) and OUTFIT (in Chapter 6) to create test cases going into a specific branch of a program and to generate high structural coverage use cases. As to introduce symbolic execution and the tool KLEE used in both 8Cage and OUTFIT, we describe both in the background chapter and reference it in the respective chapters of the operationalizations. A commonality of all defect model operationalizations of defect-based quality assurance in this thesis is their usage of Matlab Simulink. Matlab Simulink is a commonly and frequently used implementation language in the embedded systems domain since it is easy to understand for mechanical and electrical engineers. This section serves to form a common understanding of the development of embedded systems in Matlab Simulink throughout this thesis and is referenced in the respective chapters.

2.1 Quality Assurance

Quality assurance encompasses all software engineering activities to guide the creation and assessment of artifacts during the development and maintenance of a software

verification and validation

system. It spans from the creation of the first artifacts in requirements engineering to the last artifacts in acceptance testing and aims to assure (1) the satisfaction of customer requirements and (2) the correctness of all artifacts derived from the customer requirements. The former is commonly referred to as validation, while the latter is named verification. Quality assurance is divided into constructive and analytical quality assurance. Constructive quality assurance aims to assure quality by using processes, coding guidelines and process measures for the early detection or even prevention of defects. Analytical quality assurance aims to assure quality by performing an artifact-oriented assessment for the detection of defects in these artifacts. This thesis focuses on analytical quality assurance in the form of verification (although constructive quality assurance is discussed in Section 8.2). This encompasses the static and dynamic analysis/verification of artifacts. Static analysis includes any form of automatic or manual form of analysis of non-executable artifacts. The techniques of static analysis are the compliance checking to coding standards, the computation of metrics, the formal verification of properties of (parts of) programs and reviews/inspections. The predominant technique in dynamic analysis is software testing, which includes any form of execution of an executable artifact. Both forms of analytical quality assurance are introduced in the following.

constructive and analytic QA

static and dynamic analysis

2.1.1 Faults, Errors, Failures and Defects

While constructive quality assurance aims to avoid/mitigate some defects, analytical quality assurance aims to detect discrepancies between the specified and implemented system. Thus, the goal is to verify the requirements/specification and detect defects. The term defect is used throughout this thesis as an umbrella term for any fault, error or failure made in the process of designing or implementing a system. We refrain from using the terms bug, mistake or problem as their usage has been most ambiguous in literature and practice. However, the term failure, error and fault are clearly defined.

defect

failure

error

fault

According to Laprie et al. [93], a failure is a deviation of expected and actual behavior. An error is defined as the deviation of expected and actual state possibly leading to a failure. A fault is the actual reason / mistake causing the deviation in any artifact created during the development of hardware or software. In the light of these definitions, test case execution is only able to detect failures as internal states are hidden. Thus, additional effort is required before the defect can be removed after a test case has failed. This includes the reproduction of the failure (possibly in other scenarios), fault localization and debugging to remove the fault [119]. In static analytic quality assurance techniques, the fault is directly detected. In reviews/inspections it is often referred to as anomaly before being confirmed by a second reviewer/reader.

debugging

2.1.2 Software Testing

(Dynamic) software testing is "*dynamic* verification that a program provides *expected* behaviors on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain." [1]. Dynamic means that artifacts executable by a machine are required, which then require inputs or traces of inputs to be stimulated. Finite refers to one of the fundamental issues of testing. Even when looking at simple programs that add two 32-bit integer values, it instantly becomes clear that a complete/exhaustive test of all possible inputs (over 8 billion) is infeasible. Thus, a subset of all possible inputs must be found as a trade-off between residual risk and available resources. It is noteworthy that confidence in a program must be established using this subset. However, as Dijkstra described it, tests "can be used to show the presence of bugs, but never to show their absence" [1]. Test selection is the field of software testing aiming at the "suitable" selection of test cases and making it the core difference between different testing techniques [1]. The second fundamental issue of software testing is the determination of expected behavior also called oracle. After executing a test the verdict passed, failed or unknown must be clearly assignable to a test reflecting whether the system under test (SUT) behaved as expected by the specification or the user.

test selection problem

oracle problem

Levels of Testing

In general, software testing is divided into three levels of testing. On the first level, there is unit testing. "Unit testing refers to testing program units in isolation" [119]. A unit is not clearly defined and can be a function or method, but also a class in object-oriented programming or in the context of embedded system it can be a single Simulink subsystem. Thus, unit testing aims to test a single piece of functionality such as an addition of two integers or the popping of a value from a stack. On the second level, integration testing connects units or their aggregates of units called components and tests their interaction [119]. As different components are typically developed by different development teams in a divide and conquer style, integration testing particularly aims at detecting defects in the interfaces of the components. On the third level, system testing tests the fully integrated system on the deployment hardware. The test on the actual hardware is the key difference to the test of the fully integrated system now enabling end-to-end functional tests [38]. When executing system tests, particularly the fulfillment of non-functional requirements such as performance or crash recovery can be assessed [119]. When testing embedded software, the level of unit and integrations tests is usually referred to as Software-in-the-Loop (SIL) testing, whereas the deployment on the hardware is referred to as Hardware-in-the-Loop (HIL). In addition, there are software and hardware variants on the integration and system levels as one hardware system may run multiple software systems (e.g. an automotive ECU running multiple drive assistance systems).

unit test

integration test

system test

SIL

HIL

Test Case Selection

Finding the subset of all possible inputs such that a trade-off between residual risk and available resources is achieved is non-trivial. Test case selection has been tackled by researchers and practitioners alike in the past. Since the defect model methodology introduced in this thesis also presents a test selection strategy, this section gives an overview of existing test selection strategies. These strategies will be referenced by the next chapter when describing instantiations of defect models (see section 3.2).

All strategies for case selection are traditionally classified by the information available to the tester. "If the tests are based on information about how the software has been designed or coded" [1], they are referred to as white-box. "If the test cases rely only on the input/output behavior of the software" [1], they are referred to as black-box. However, this categorization of test methods is too coarse since there are more black-box than white-box strategies.

white-box test

black-box test

A more finely grained categorization is used in the Guide to the Software Engineering Body of Knowledge [1]. They distinguish seven categories of test case selection strategies (referred to as test techniques in [1]) based on "how test[s] are generated" [1]. These comprehensive categories and their respective strategies encompass all strategies described by Naik and Tripathy [119], Myers and Sandler [117] and Beizer [17]. Therefore, they are presented in the following with the additions made in the aforementioned literature to give a comprehensive overview.

Based on the Software Engineer's Intuition and Experience. In this category, the test cases are selected either (1) ad-hoc or with (2) exploratory testing. In ad-hoc test case selection, "tests are derived relying on the software engineer's skill, intuition, and experience with similar programs" [1]. These test cases must not necessarily target defects. In case they do, they are ad-hoc fault-based test cases (see below). Exploratory testing [151] uses manual dynamically designed and subsequently executed test cases that are created by navigating and learning the tested application. The navigation of the application typically follows a goal (also called tour) to detect certain defects. These defects may be deviations in the user interfaces (Supermodel Tour) or security issues (Saboteur tour).

ad-hoc test

exploratory testing

Input Domain-Based Techniques. These techniques rely on the specification to select test cases and include equivalence partitioning, pairwise testing, limit testing and random testing. Equivalence partitioning "involves partitioning the input domain into a collection of subsets (or equivalent classes) based on a specified criterion or relation" [1]. Any equivalence relation[1] can be used to partition the input space. A number of test cases are then selected from certain or all blocks of the partition. Pairwise testing belongs to the area of combinatorial testing and uses the pairs of input instead of all combinations of inputs [89]. Limit testing (also called boundary-value analysis [122]) chooses test cases "on or near the boundaries of the input domain of

equivalence partitioning

pairwise testing

[1] An equivalence relation is a binary relation $\sim$ on a set with reflexive ($a \sim a$), symmetric ($a \sim b \rightarrow b \sim a$) and transitive ($a \sim b \wedge b \sim c \rightarrow a \sim c$) properties

variables" [1]. It has the rationale (later introduced as defect model) "that many faults tend to concentrate near the extreme values of inputs" [1]. Random testing selects purely random test cases from the complete input space. The test selection effort for random testing is negligible and test automation is easily possible. A special form of random testing is fuzzing, which may use some directed approach to guide random testing [1]. This guidance may aim at the exploration/coverage of the system under test or its outputs [2, 17] among others. In case the guidance used in fuzzing targets specific faults in the systems, it falls into the category of fault-based techniques below.

limit testing

random testing

Code-Based Techniques. This category of techniques uses the code for test case selection and includes control/data flow-based criteria. Control-flow based testing aims at covering elements in the control flow graph of a program [1]. These elements can be statements, branches in the control flow graph or paths through it [17]. In addition, there are criteria regarding the conditions/decisions taken in the graphs such as condition, multiple condition and modified condition/decision criteria [117]. Data flow criteria annotate the control flow graph "with information about how the program variables are defined, used, and killed (undefined)" [1]. The criteria define elements to be covered to include all-definitions, all-uses, all-computational-uses and all-predicate-uses of variables in the program.

control-flow testing

data-flow testing

Fault-Based Techniques. "Test cases specifically aimed at revealing categories of likely or predefined faults" [1] are selected with fault-based testing techniques. Error guessing and mutation testing are two commonly cited techniques in this category [1, 119]. Error guessing creates test cases "to anticipate the most plausible faults in a given program" [1] based on earlier faults and the tester's knowledge and experience. As implied by the word guessing, this test selection technique is typically applied ad-hoc and unsystematically. Mutation testing performs test suite assessment by using fault injection on the original program [44]. Every fault-injected original program is called a mutant and killed, if the test suite is able to detect the injected fault. Mutation testing has one major underlying assumption called the coupling hypothesis. It states that injecting simple syntactic faults will lead to the detection of more complex/real faults [44]. The creation of a systematic and (semi-)automatic approach to select test cases targeting certain defects in systems is the central topic of this thesis positioned in this category of test selection techniques. Chapter 3 describes the generic defect models for defect-based quality assurance and gives a detailed representative selection of existing/related works in fault-based testing techniques including those above. It also shows how fault-based and failure-based techniques are instantiations of the generic defect model.

error guessing

mutation testing

defect model position

Usage-Based Techniques. Test cases based on the usage of a program are selected with usage-based techniques. These include operational profiles and user observation heuristics. Operational profiles allow the selection of use cases based on the expected usage of a functionality to infer future reliability. Markov chains can be used as underlying models for the usage probabilities and test cases are typically selected on

operational profiles

user observation heuristics

the system testing level [1, 152]. User observation heuristics can be used to discover "problems in the design of the user interface" [1] and "are applied for the systematic observation of system usage under controlled conditions in order to determine how well people can use the system and its interfaces" [1].

Model-Based Testing Techniques. Model-based testing uses "an abstract (formal) representation of the software under test" [1] or its environment for the selection of test cases. The technique is inherently automatable as test cases can be generated from the model and executed on the system under test without manual effort. There are several techniques to create the model and perform the test generation/execution described by Utting et al. [147] in a taxonomy.

Techniques Based on the Nature of the Application. Tests based on the nature of the application include tests that are specific to one domain of software engineering. These include object-oriented, web-based, concurrent and embedded testing. In each of these domains different faults are to be detected, which are particular to that domain. For example, inheritance faults are only to be present in object-oriented software whereas HTML layout faults are only possible in web-based software.

2.1.3 Static Analysis

Methods of static analysis perform verification on a respective artifact without executing it (even if it is executable). They include checking of suspicious usage/coding standards (e.g. lint [41] or MISRA-C [114]), calculation of metrics (e.g. cyclomatic complexity [110]), formal proofs (e.g. Hoare logic [68], model checking [33], abstract interpretation [36]) and reviews/inspections (e.g. walkthroughs [75] and Fagan inspections [52]). This section gives an introduction to each static analysis method.

Suspicious Usage/Coding Standards

lint

Programs such as lint [41] analyze the source code of a program and report suspicious usage of the programming language. The findings of lint are usually called smells, as they show potential defects, which may have been taken care of in another part of the program or may never occur in the environment the program is deployed in.

misra-c

MISRA-C [114] is a coding guideline in the automotive domain. It aims to avoid run time failures by disallowing language constructs of C and provides best practices. Checking for compliance is performed on the code and deviations are reported.

Software Metrics

Other than code coverage metrics, that are provided by software testing, there are a plethora of metrics, which can be calculated statically. These metrics aim to optimize non-functional requirements such as performance or maintainability. For instance,

cyclomatic complexity

a large valued of McCabe's cyclomatic complexity [110] indicates a very complex function/program, which may benefit from modularization. Other measures like

inheritance depth and method size for object-oriented programs [31] indicate the maintainability of the program.

Formal Proofs

To show that a program outputs the correct data for all its inputs, a formal proof is required. Hoare logic [68] uses a system of deductive rules for the proof, while theorem provers are based on higher-order logic [90]. After transforming the program into a model (e.g. a finite state machine (FSM) or Kripke structure), model checking [33] can provide a proof or counterexample for a given (typically temporal logic) property of the model. Abstract interpretation semantically abstracts from the source code using the notion of abstract objects to mitigate the undecidability problem. By using a generalized form of the program, it aims to give answers to "questions, which do not need full knowledge of program executions or which tolerate an imprecise answer" [36]. If a property can be proven or a counterexample can be found in the abstraction of the program, it can be proven/disproven in the actual program by concertizing.

hoare logic

model checking

abstract interpretation

Reviews/Inspections

Reviews and inspections represent manual reading techniques performed on human-readable artifacts. The IEEE Standard 1028 for software reviews and audits defines a review as "a process or meeting during which a software product is examined by a project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval" [75]. There are several types of reviews including a formal review type referred to as (Fagan) inspection.

Ad-hoc Code Reviews. The most basic form of a review is an ad-hoc code review. This code review is typically performed by the developer to localize a fault after the program has failed, to check whether cloned code was correctly adapted to the new situation or to sign off code before checking it into a repository. It is triggered dynamically and target as well as purpose are left to the developer. A form of continuous ad-hoc review is the agile technique of pair programming [153], where two developers work as a pair. In pair programming, one developer is the driver writing code and the other is the navigator constantly reviewing the driver's code.

pair programming

Walkthroughs. A walkthrough is "a static analysis technique in which a [developer] leads members of the development team [...] through a software product, and the participants ask questions and make comments about possible anomalies, violation of development standards, and other problems" [75]. As an example, a walkthrough of a piece of code has the developer of the code present the control/data flow and the other team members comprehend/comment the decisions made.

Technical Reviews. Technical reviews can be performed on any human-readable artifact (including code) in a formal target-oriented way described in the IEEE Standard 1028 for software reviews. The generic process for formal reviews includes seven steps

and requires the documentation of the review process. A review may be done without preparation and completely within one meeting.

Inspections. Inspections were first described by Fagan [52] to perform the most formal review out of any aforementioned technique. The inspection process has six steps and requires the offline preparation of all participants before the meeting. Additionally, it also considers the removal of the defect part of its process. Thus, an assigned moderator must also inspect the rework. For the preparation of the meeting, the instructions for the inspectors can either not be provided (ad-hoc reading), provided as a checklist (checklist-based reading) or require the inspector to perform certain tasks in certain roles (perspective-based reading) [15].

inspection reading techniques

2.2 Symbolic Execution

Symbolic execution was introduced by King [87] as a method of program verification. By substituting the concrete input values in a program's execution by mathematical symbolic values, conditions for each path through the program can be created. By exploring each path through the program concerning its input/output relation, the correctness can be established. King shows such verifications to be possible "in a simple PL/I style programming language" [87]. However, loops cause the number of potential paths to increase to infinity and decidability hinders the exploration of all possible paths. When exploring a program with symbolic execution, each possible path is described by so-called path conditions (PCs). The path conditions are initialized as TRUE (i.e. satisfied) at the entry point of a program (e.g. the main function). Symbolic execution then executes instruction after instruction of the program while keeping track of the modifications to the symbolic input values. Once a branching instruction occurs, the logical expression of the branching is added to one PC and its negation is added to another PC. These two PCs then describe the if and else branch and both paths are executed symbolically thereafter. Execution is terminated when reaching an error or the termination of the program. Upon termination, a solution to the path conditions yields a test case for the respective path. One major issue in symbolic execution are loops. Loops are broken down to if and else branches where the if branch loops back to a previous block of instructions. How often the if branch is to be taken is inherently undecidable.

symbolic value

path condition

Listing 2.1 shows an exemplary program to demonstrate symbolic execution. In the first step of symbolic execution, the variables argc and argv are made symbolic. Thereafter, the instructions generated by the printf statement are executed with no change to the symbolic variables or PCs. The subsequent branching add a PC where argc is larger than 2 and one where it is smaller. Following the first PC, printf does the same as before and the program terminates. The PC $argc > 2$ is easily solved by choosing any value in the set $2 > argc >= 32767$. The program also terminates with only printf for the negated PC $argc <= 2$ and any number of the set $-32768 > argc >= 2$ will

Listing 2.1: Source code for the symbolic execution example

```
1 #include <stdio.h>
2
3 main ( int argc, char **argv ) {
4 {
5   printf("Symbolic execution test!\n");
6    if (argc > 2) {
7        printf("Success!\n");
8    } else {
9        printf("Try again!\n");
10   }
11   return 0;
12 }
```

suffice. Note that, the value of argc (i.e. the argument count) passed by the operating system will always be positive.

The original recommendation by King [87] was to use a table of symbolic values and path conditions for larger programs in a manual symbolic execution. Due to recent improvements in constraint solving [42], automatic symbolic execution for programs in well-established programming languages was made feasible. Today, symbolic execution frameworks for C/C++, C# and Java exist [26]. The most prominent examples of currently maintained symbolic execution frameworks are KLEE [25] and CREST [24] for C/C++, PEX for C# [144] and Java Path Finder with its symbolic execution extension [3, 148] for Java. The major cost-effectiveness measure for all approaches is the coverage achieved by their generated test cases. Although all symbolic execution frameworks use tricks to accelerate the solving of the path conditions, the results obtained still show scalability to be the major problem for symbolic execution. As an example, KLEE's symbolic execution abilities are even unsuitable for some of the non-complex GNU coreutils [25].

klee

Listing 2.2: Command for running KLEE

```
1 klee     --simplify-sym-indices \
2          --max-sym-array-size=4096 \
3          --max-instruction-time=30 \
4          --watchdog \
5          --max-time=5000 \
6          --max-memory=1000 \
7          --optimize \
8          --only-output-states-covering-new \
9          --max-memory-inhibit=false \
10         --search=random-path \
11         --search=nurs:covnew \
12         --use-batching-search \
13         a.out
```

KLEE performs symbolic execution on the LLVM bitcode level of abstraction. LLVM is "a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs by providing high-level information to

llvm bitcode

compiler transformations at compile-time, link-time, run-time, and in idle time between run" [94]. To compile C/C++ source code to LLVM bitcode, the clang[2] compiler is provided by the LLVM developer group. As a default, clang compiles to binary formats, but given the argument `-emit-llvm` produces the desired bitcode. KLEE can then perform symbolic execution on clang's output. Throughout this thesis, we execute KLEE on the output of clang using the command line in Listing 2.2. One of the features of LLVM bitcode is the independence from the processor architecture and operating system as long as LLVM and clang binaries are available. This eliminates the need for cross-compilation and allows the compilation and execution of LLVM bitcode on completely different machines.

2.3 Development of Embedded Systems in Matlab Simulink

Matlab Simulink [109] is a development and runtime environment for embedded systems software. It aids in model-based architecting, designing, implementing and testing. Simulink models use a data flow driven block-based notation similar to wiring plans. Blocks execute functions such as arithmetic, accumulative, limiting and other operations to the data they are given. Each block has a specific number of inputs and outputs which are connected to other outputs and inputs. Each connection represents a data flow. Special I/O blocks let data flow into and out of the model. The notation helps mechanical/electrical engineers by giving well-understood model notations. Originally designed as a pure simulation environment for differential equations, code generation form the model was added for direct deployment of the implemented system. Thereby, programming efforts for continuous and hybrid systems can be performed by the engineers and seamless engineer comprehensibility is ensured.

Such a model-based approach is widely employed for automotive drive trains, aerospace engine control and brushless motor controllers, among others. For building complete software systems, Matlab allows abstraction by encapsulating so-called subsystems. These subsystems can be used in a different model, allowing their composition in levels. The lowest level is the unit level, constituting a model without any subsystems, using only built-in blocks of the Simulink library. Each layer above the unit level represents an integration level where two or more unit or integration levels are composed into components. The uppermost level is the system level consisting of all unit and integration levels. Most embedded systems are based on the input-processing-output model [60] as (1) input calculations are required to go from raw sensor values to the values needed for processing, (2) processing performs the calculations and is monitored by a supervisor and (3) output calculations are required to drive actuators towards the calculated values. Thus, typical high-level components of embedded systems are input calculation, processing/monitoring and output calculations. A particularity of these high-level components is their connectivity, as outputs of one component are inputs

matlab model

matlab component

[2] `http://clang.llvm.org/`

to the next. This is similar to a pipe-and-filter architecture [137] where filters do not need what they are connected to and can be executed in parallel.

When developing embedded software, the development cycle usually starts with the implementation of units. Once a unit is finished, it can enter unit testing by a tester. To perform first tests, Matlab provides a simulation environment to directly perform tests on the Model-in-the-Loop (MIL) test level. The tester will typically perform a black-box MIL and SIL (see Section 2.1.2) functional test according to the unit's specification. Units are then composed into components and integration testing is performed on certain (logical) combinations of components until the system level is reached. System testing is performed on the SIL as well as on the black-box HIL (see Section 2.1.2) level, where the environment of the system is simulated. These environment models are typically a problem in practice requiring further research as they may not be available/correct/complete etc. In a last step, static analysis using abstract interpretation (see Section 2.1.3) is performed on the production ready system to detect run time detectable failures. If such failures are detected, a trained expert needs to perform fault localization and present the result to the respective developer(s). When the faults are removed, the development cycle restarts with unit and integration tests.

MIL

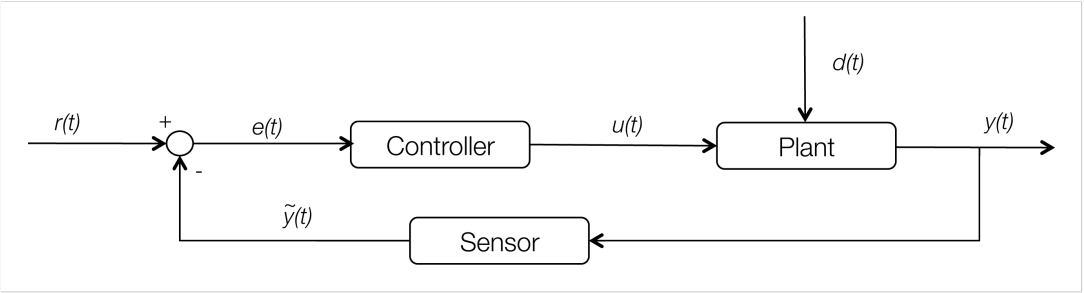
2.3.1 Control System Design

The systems developed in Matlab can be transformative systems or even have simple state as is the case with raw value conversions in the input calculations part. However, the major advantages of Matlab Simulink lie in the design of control systems by modeling differential equations. Control systems are an integral part of our everyday life. They range from steady temperature control in ovens to highly reactive electronic stabilization programs (ESP) in commercial vehicles and precisely rotating brushless motors. Originally implemented in continuous mechanical or electro-mechanical forms, today's controllers are mostly implemented using discrete microprocessors. The advantages are improved versatility and configurability towards a plethora of usage scenarios. In particular, one does not need to build an ESP controller for each vehicle model, but rather for one or multiple vehicle product lines.

The two essential parts of any control system are a controller on the one side, and a plant or process with a sensor on the other side (see Fig. 2.1). All controllers in this thesis are given a single desired value over time ($r(t)$) as reference-variable input signal and are supposed to drive ($u(t)$) the process to this value. In case the process's single control variable output signal or actual value over time ($y(t)$) is fed back to the controller ($\tilde{y}(t)$), the control system is referred to as closed-loop, and open-loop otherwise. A closed-loop control system enables the controller to correct or adapt to the changes detected in the process. In this thesis, only closed-loop systems are considered. However, defect models can also be applied to open-loop control.

controller
plant
closed loop
open loop

Figure 2.1: Closed-loop control system



reference-value response
disturbance response

For closed-loop control systems, there are two fundamental responses to test: reference-value and disturbance response. Reference-value response is observed if a desired value is given to the controller and it can drive the process without any environmental influences. Disturbance response is observed if environmental influences ($d(t)$) such as opposing forces, cooling or heating effects are present [59]. Testing each response separately allows control system engineers to make adjustments particularly pertaining to the considered response. However, this test is not always possible. In reality and particularly when non-linear, the controller is exposed to both desired value changes over time and environmental influences at the same time, thus leading to a hybrid of both behaviors.

As an example, let the process be an oven and the actuator its heating element. Turning a knob to a desired temperature would set the desired value for the controller. A controller would then switch the heating element on until the desired temperature value is reached. Unfortunately, the temperature in the oven does not stay constant at the desired value due to cooling effects. Thus, the controller has to turn on the heating element once the actual temperature value differs from the desired temperature value. When the knob is turned to a different temperature, reference-value response is exhibited. When the door of the oven is opened causing a substantial temperature drop, disturbance response is exhibited. An interesting aspect of the disturbance response is when to react to a change in temperature. Switching on the heating element when only a slight change in temperature is detected may not be energy-efficient. Thus, there may exist a dead band requirement demanding not to switch on the heating unless the temperature has changed by a certain percentage.

Controllers and plants can be designed and tested as differential equations directly within Matlab Simulink. Such MIL tests are used to verify the functionality w.r.t. to the functional response of the controller within the Matlab Simulink runtime environment, which allows precise simulations. Note that, this thesis considers discretized continuous responses, possibly with modes of the control system (i.e. hybrid responses). Such modes of the control system are typically created in a special Stateflow block in a Simulink model, that allows the implementation of logic in graphical state machines.

control system design process

The process of designing a control system in Matlab Simulink consists of several steps.

Firstly, the process characteristics have to be determined and have to be designed on the model level. Subsequently, a suitable type of controller must be chosen. Thirdly, the selected controller must be adapted to the application by adjusting its parameters. At this stage, the controller is implemented as a continuous system in both the time and value dimensions. Matlab Simulink allows the simulation of continuous controllers and production-ready code derivation for targeted hardware platforms, which may also be performed using other tools. To be able to implement it as software, it must be discretized w.r.t. time and subsequently w.r.t. values in a fourth step. In a last step, discrete values may need to be converted to fixed point as some microprocessors lack hardware floating-point units (FPUs) and do not support hardware floating point calculations.

2.3.2 Control System Verification

A classic way of quality assurance for control systems is to examine the transfer function [59]. The transfer function describes the response of a control system to an input signal as a mathematical function in a black-box way. Given today's complex controllers, including feed forward and cascading controllers, the derivation of the transfer function is infeasible in the time given to test the controller. Thus, control system engineers commonly perform manual knowledge-based testing. Typically, they partition the input space of the desired and disturbance values into equal blocks [50]. In addition, a block is allocated to all boundary values, thus yielding a form of limit testing. One representative from each is picked as test input. To judge requirement conformance, the engineer will inspect the trajectories of the desired, actual and disturbance values and form a knowledge-based verdict of requirements fulfillment.

transfer function

3

Generic Defect Model for Quality Assurance

The central question of test case selection is "what is a good test case?". Since the 1970s, the answer to this question was "one that finds fault" [17, 117]. However, considering the hypothetical existence of perfect programs, there would be no good test cases for such programs. Albeit, one would like to execute test cases on a perfect program to ensure it fulfills its requirements. Such good test cases for a perfect program (or any other program for that matter) would then find potential faults [127]. This definition appears to be capturing the aim of testing, but does not yet capture any required resources. When performing test case selection in practice, factors such as creation, execution, fault localization and debugging effort are taken into account to decide whether or not to choose a test / use a method of test case selection. The optimal scenario is a test case which detects a potential fault while yielding no costs. Since this scenario is unrealistic in practice and typically different types of faults have different cost implications, the test cases must be cost-effective w.r.t the cost factors above. Thus, a complete definition of a good test case is "one that finds a potential fault with good cost-effectiveness" [127]. The same definition can be applied to static analysis (e.g. via a tool or check list in review/inspection).

good test case

The definition of good test cases and static analysis above is far too abstract to be directly operationalizable. However, test selection strategies (see Section 2.1.2) yielding good test cases in the above sense exist. This leads to the question of which test selection strategies are able to select good test cases. An abstract model of test selection was created by Weyuker and Jeng [150] to compare random testing to partition-based testing. Partition-based testing according to Weyuker and Jeng is an abstraction of equivalence partitioning in Section 2.1.2, where an arbitrary criterion can be used for partitioning. Thus, partition-based testing is not only part of the input domain-based test selection techniques, but other test selection techniques also inherently partition the input domain. Based on the test selection categories introduced in Section 2.1.2,

these include code-based and usage-based among others. When selecting test cases according to code-based techniques, the input domain of the program is partitioned according to the paths taken through the programs concerning the control or data flow. In usage-based techniques, the input is partitioned into blocks with respective usage frequencies and the number of test cases picked from each block depends on its usage frequency. Thus, Weyuker and Jeng deliberately choose partition-based testing for its generality. As random testing can be seen as the gold standard of testing with (almost) no test derivation effort, it is the obvious choice as a baseline test selection technique. Thus, any test selection technique with test derivation effort should be better than random testing to be worth the test derivation effort.

For comparison, they define a program P to have an input domain $\mathcal{D}$ of size d, m failure-causing inputs of $\mathcal{D}$ with $d >> m$ and a failure rate of $\theta = m/d$. For random testing, they perform the test selection based on a uniform distribution as related work does so as well and "using a uniform distribution seems most appropriate" [150]. For partition-based testing, they partition $\mathcal{D}$ into non-empty pairwise disjoint subsets called blocks $D_0 ... D_k$, where $0...k$ is the identifier of the block and $i \in 0...k$. Let d_i be the size, m_i be the number of failure causing inputs and $\theta_i = m_i/d_i$ be the failure rate of each block D_i. As quality criterion, they choose the probability to select / detect at least one failure-causing input with n test cases while at least one test case (i.e. input) must be selected per block. For random testing, this probability is expressed by $P_r = 1 - (1 - \theta)^n$. For partition-based testing, it is $P_p = 1 - \prod_{1 \leq i \leq k} (1 - \theta_i)^{n_i}$, where the number of test cases selected per block n_i must be larger than 1 and the sum of all n_i must be n $\left(\sum_{1 \leq i \leq k} n_i = n\right)$. Using their model and quality criterion, Weyuker and Jeng's result is that "partition-based testing can be better, worse, or the same as random testing" [150]. As an example, let d be 100, m be 8 and n be 2 to demonstrate the three cases. Firstly, partition-based testing is the same as random testing when $\theta_1 = \theta_2 = 4/50$ and $n_1 = n_2 = 1$. This is obvious as all failure-causing inputs are uniformly distributed in both blocks. Secondly, partition-based testing is worse than random testing when $\theta_1 = 8/99$ and $\theta_2 = 0/1$ and $n_1 = n_2 = 1$. In this case, one test case is wasted on a block with no failure-causing inputs. Thirdly, partition-based testing is better than random testing when $\theta_1 = 0/92$ and $\theta_2 = 8/8$ and again $n_1 = n_2 = 1$. This input space partition successfully captures all failure-causing inputs in one block and at least one failure causing input is always detected.

These results are interesting as they suggest a test selection technique to be effective only if it takes the distribution of failure-causing inputs into account. Thus, there must be a frequency of usage giving faults an inherent severity (e.g. usage-based or requirements-based test selection) of a *fault hypothesis*. As a result of selecting test cases by frequency of usage, the number of faults does not necessarily decrease, but the severe faults in the most used features are detected decreasing the system's failure rate in the field. A fault hypothesis is an assumption as to which inputs are (likely) failure-causing possibly derived from knowledge and experience. This assumption may

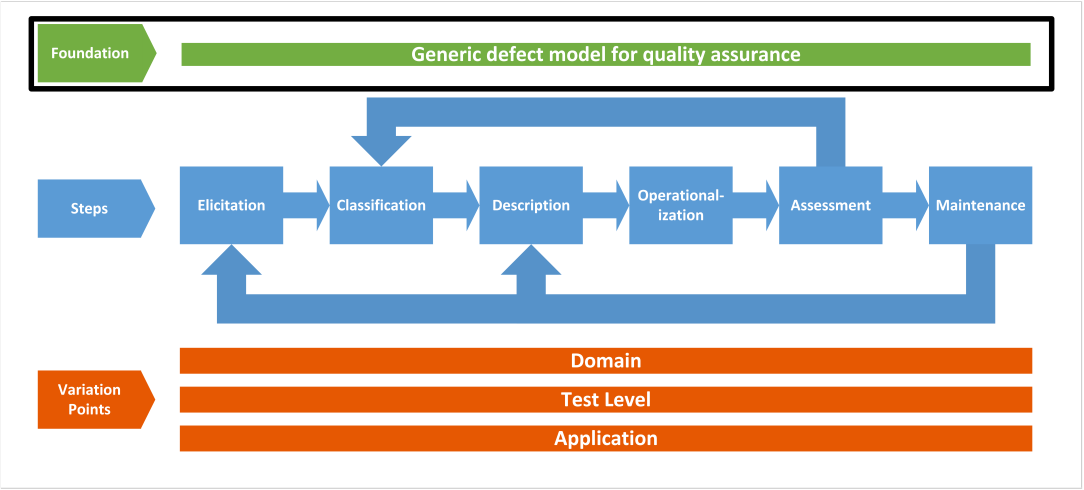
originate from knowledge and experience or may just be pure speculation/guessing. Albeit, it appears to be the central ingredient in several test selection techniques known to select good test cases. As an example, error guessing (see Section 2.1.2) inherently selects test cases based on the hypothesis of certain inputs to be (likely) failure-causing. Also, limit testing has the fault hypothesis or encapsulated knowledge of faults occurring at the boundaries of blocks and there is at least anecdotal evidence of its effectiveness. The same argument also applies to the static analysis techniques of reviews/inspections. These techniques can check lists or even defect taxonomies to partition human-readable artifacts based on a fault hypothesis. Thus, some test selection techniques appear to have encapsulated knowledge and experience yielding a fault hypothesis at their core.

An interesting aspect of input space partitions created by code-based techniques (i.e. control-flow/data-flow coverage) is the conclusion that they can be better, worse or the same as random testing according to the model of Weyuker and Jeng [150]. Thus, according to Weyuker and Jeng, the effectiveness of code coverage-based test selection criteria is questionable. However, the model of Weyuker and Jeng makes several inherent assumptions (e.g. uniformly selected test cases in random testing) and uses a debatable quality criterion affecting its generalizability.

Inozemtseva and Holmes [77] summarize previous studies concerning the effectiveness of tests selected based on code coverage criteria and perform their own study. The results have been mixed and including their own study, there are 5 studies finding a correlation between code coverage and test suite effectiveness, 2 that do not and 5 inconclusive ones. Thus, the effectiveness of test cases selected by code-based techniques remains disputed and it is questionable whether code-based techniques are able to derive good test cases. If the distribution of failure-causing inputs must be taken into account for a test selection technique to be effective and code-based techniques do not take this into account, this may explain the inconclusiveness.

Based on the considerations of the distribution of failure-causing inputs and the requirement of a fault hypothesis to create effective test cases, the following section provides a key to understanding the knowledge and experience encapsulated in some test selection strategies to select good test cases called defect models. They yield the foundations for the defect-based quality assurance approach presented in this thesis. We formally capture the notions of fault (1) and failure-provoking (2) test selection (i.e. test cases created with an underlying fault hypothesis) defining fault (1) and failure models (2) as the two kinds of defect models for fault-based and failure-based testing respectively. To arrive at generic defect models and gather related work, we perform a literature survey in the area of defect-based quality assurance explicitly using the term “fault model” . To evaluate the generality of our defect model, we show all findings in literature to be instances and give a notion to their operationalization. Finally, generic guidelines to the operationalization of defect models for the derivation of (semi-)automatic test case generators are given.

Figure 3.1: Position of the generic defect model chapter in this thesis



This chapter represents the formal foundations to capture defect knowledge for its (semi-)automatic operationalization shown in Figure 3.1 in the defect model lifecycle framework. It is based on previous work [128, 129] emerging from a cooperation between the supervisor and the author of this thesis.

3.1 Definition

Acknowledging the need for fault hypotheses in quality assurance, we aim to define a construct to capture these fault hypotheses and to operationalize them automatically in quality assurance. As this construct seems to be a model of the fault, we perform a literature survey in the area of quality assurance for the term "fault model". This survey yields that the term fault model is already commonly used by authors to express the notion of a fault hypothesis. Two common patterns are typically exhibited: the authors either state the faults in the SUT to be of a certain nature (e.g. "fault x, y and z are detected by our approach") or state how failures are provoked (e.g. "our approach uses data a to make the system fail"). However, the authors never state "a fault model is" and give it a definition. Thus, there is no comprehensive definition of the term fault model available. Considering the fault hypotheses above and the literature survey, a generic definition must encompass the fault hypotheses, must have all existing fault models as instantiations and provide a guide to their operationalization.

common patterns to describe fault models

The existing definition of Martin and Xie states that a fault model is "an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software [103]." Harris states that a fault model associates a potential fault with an artifact [64]. Both definitions are rather abstract and describe faults of any activity during the software engineering process or even during deployment. However, the aforementioned descriptions are too general as they encompass also process-related aspects. In contrast, the definition by Bochmann et al. [22] says

existing fault model definitions

that a fault model "describes a set of faults responsible for a failure possibly at a higher level of abstraction and is the basis for mutation testing". This definition is related to quality assurance and the first common pattern above. However, it is still too abstract and circular to encompass the fault hypotheses and have the existing fault models as instantiations. Thus, we will give a definition of fault model encompassing the first common pattern and failure model encompassing the second in the following. Both comprehensive definitions of fault and failure model form the generic defect model.

3.1.1 Prerequisites

One of the goals of a generic defect model for quality assurance is comprehensiveness w.r.t. all artifacts created in the description of the system. On one hand, this requires considerations of all executable artifacts related to testing activities. These are not only executable programs, but also models for model-based testing. On the other, it also requires non-executable human-readable artifacts related to review / inspection to be considered. The commonality of all artifacts is their description of behavior, which semantically creates an input/output relation. Thus, we refer to them as behavior descriptions (BDs) containing a semantic function mapping inputs to outputs. For every BD, we assume it is developed w.r.t. another behavior description (BD), a so-called specification. Syntactic representations of BDs form the set $\mathcal{B}$; syntactic representations of specifications form the set $\mathcal{S}$. Of course, $\mathcal{B}$ and $\mathcal{S}$ are not necessarily disjoint.

behavior description

specification

Given universal input and output domains I and O, both BDs and specifications give rise to associated semantics

$$[\![\cdot]\!] : \mathcal{B} \to (I \to 2^O).$$

Note that this semantic function is in concurrence with the suggestion of Mills [113] and Morell [116] with the extension that it supports non-determinism. For our purposes, it is sufficient to define it as partial function with $\forall b \in \mathcal{B}.\forall i \in dom([\![b]\!]).[\![b]\!](i) \neq \emptyset$ where $dom([\![b]\!])$ is defined as the domain of the behavior description. In case the behavior description does not produce an output for a given input i (i.e. the output is defined, but empty), the output is an explicit $\{\epsilon\}$.

semantics

A BD $b \in \mathcal{B}$ is *correct* w.r.t. or satisfies a specification $s \in \mathcal{S}$ iff $\mathrm{dom}([\![b]\!]) \supseteq \mathrm{dom}([\![s]\!])$ and $\forall i \in \mathrm{dom}([\![s]\!]) : [\![b]\!](i) \subseteq [\![s]\!](i)$. This is denoted as $b \models s$. To grasp a global notion of correctness, we assume the specification of a BD to be verified at a higher level and, therefore, to be correct.

correctness

3.1.2 Faults

The definition of fault models in the following relies "on the inner structure of a behavior description, and therefore can be associated with white-box testing strategies" [128]. Thus, we give a precise definition of fault before defining a fault model.

Faults are textual (or graphical) differences between an incorrect and a correct BD and affect the inner structure of the BD. An incorrect BD can be turned into a correct BD

fault

by doing replacements that obliterate the respective differences. Vice versa, by adding the differences to a correct BD, it can be turned into an incorrect one. The differences themselves relate to elements in the BD. For a textual program, a character, line, block or function may be replaced, while for a Matlab Simulink system lines and blocks may need a replacement. For requirements artifacts, single sentences, paragraphs or section may be replaced (e.g. in a clarification replacement).

fault class K

Each fault belongs to a respective fault class. "A fault class *K*, drawn from a set of failure and fault classes $\mathscr{K}$, is a description of what has gone wrong or is considered likely to go wrong in a (hypothetical) set of behavior descriptions. Examples include a characterization of incorrect parts of a behavior description that lead to "division-by-zero", "stuck-at-one", "null-pointer-dereferencing", "array index overflow", and so on" [128]. Thus, we can categorize the replacements above by the classes of faults they insert into BDs and characterize them by the transformation

transformation A_K

$$A_K : \mathcal{B} \times \mathcal{B} \to 2^{\mathcal{B}}.$$

This transformation takes an arbitrary correct BD j and its specification s and returns a set of incorrect BDs. The set of incorrect BDs is created by trying to inject every fault in class K one or multiple times into j. Thus, A_K is a fault injection operator, which creates all combinations of incorrect BDs w.r.t. K, j and s. This injection may not be possible if the syntactic element required to apply the transformation is missing. However, the set contains versions of j that have elements added or omitted and may not be compilable. The requirement that j be correct and s is its specification is expressed by

$$(j, s) \in \mathrm{dom}(A_K) \Rightarrow \forall i \in \mathrm{dom}(s) : [\![j]\!](i) \subseteq [\![s]\!](i).$$

transformation α_K

In order to be able to apply a single transformation of all transformations performed by A_K, we defined α_K as

$$\alpha_K : \mathcal{B} \times \mathcal{B} \to \mathcal{B} \text{ with } \alpha_K(j, s) \in A_K(j, s).$$

Note that, the sole injection of a fault does not mean that this causes a failure. There could be fault handling routines or fault masking, which prevents the actual behavior of the BD to be different to the intended one and creates an equivalent behavior description. However, let α_K only produce truly faulty BDs.

3.1.3 Failure-Causing Inputs

For failure models the definition relies exclusively "on the externally visible interface of a system, and can therefore be associated with black-box testing strategies" [128]. Thus, we give a precise definition of failures/failure-causing inputs before defining failure models.

failure

Recalling the definition in Section 2.1.1, a failure is the deviation between expected and intended behavior. Failures are caused by giving a failure-causing input to a BD

and detected when the resultant behavior is compared to the behavior defined in the specification s. In general, the cause of a failure is a non-empty set of faults, which may not be characterized before performing debugging/fault localization activities. All failures can also be categorized into failure classes. "A failure class K, drawn from the above set of failure and fault classes $\mathscr{K}$, is a description of what can go wrong while executing a (hypothetical) set of behavior descriptions, as opposed to what can go wrong in one specific behavior description. Examples include the semantics of behavior descriptions that lead to "division-by-zero", "stuck-at-one", "null-pointer-dereferencing", "array index overflow", and so on" [128].

For (hypothetically) characterizing the non-computable set of all failure-causing inputs, let

failure-causing input set φ

$$\varphi(j, s) = \{i : i \in \mathrm{dom}(s) \wedge [\![j]\!](i) \not\sqsubseteq [\![s]\!](i)\}$$

be the set of inputs leading to incorrect behavior with BD j developed according to specification s. The result of $\varphi(j, s)$ then is one block of an input space partition with two blocks, where the other block is the set $\{i \in I : i \notin \varphi(j, s)\}$. This input space partition then has one block of inputs leading to correct and one block of inputs leading to incorrect output(s).

3.1.4 Fault models

fault model

We define a fault model as (1) a descriptions of the difference between a correct BD and a BD that contains one instance of fault class K and (2) a resulting set of failure-causing inputs caused by instances of fault class K. More precisely, the application of α_K to (b, s) with $b \models s$ yields a b' with $b' \not\models s$ and b' contains one instance of fault class K. This definition concurs with the definition of a fault in Section 2.1.1 of a fault being the underlying reason for a failure. α_K (or only a fault pattern in its image) creates a respective induced failure domain, which is characterized by

$$\varphi_K(j, s) = \varphi(\alpha_K(j), s)$$

with $(j, s) \in \mathrm{dom}(\alpha_K)$. $\varphi(\alpha_K(j), s)$ then contains all inputs, for which α_K has modified the path through the BD j. For a program or system, it is the execution path. For the requirements and architecture, the workflows through a use case or the interactions between the components are the path respectively. As the modification and the underlying fault is known, fault models capture faults and allow the creation of test cases by characterizing the failure domain.

3.1.5 Failure models

failure model

We define a failure model as set of failure-causing inputs caused by a failure class K. More precisely, it is exactly the set $\varphi_K(j, s)$ with BD j developed according to specification s. As the specific underlying fault is unknown, failure models capture

methods to provoke failures. Thus, failure models inherently characterize the failure domain and allow the selection of failure-causing inputs for test cases. However, φ_K is generally not computable, which requires using its approximation for test case derivation.

3.1.6 Approximations

approximated fault / failure model

α_K or its induced failure domain $\varphi(\alpha_K(j), s)$ are typically not computable. This is due to the undecidability of program equivalence/non-equivalence and a possible infinite number of paths. Thus, approximations are required. For a given BD b and a specification s, let $\tilde{\alpha}_K$ describe an approximation of α_K, and let $\tilde{\varphi}$ and $\tilde{\varphi}_K$ describe approximations of the failure domain. They are approximations in that $\mathrm{dom}(\alpha_K) \cap \mathrm{dom}(\tilde{\alpha}_K) \neq \emptyset$ and $\exists b \in \mathcal{B} : \alpha_K(b) \cap \tilde{\alpha}_K(b) \neq \emptyset$. This formal definition is rather weak. The intuition is that $\tilde{\alpha}_K$ should be applicable to *many* elements from $\mathrm{dom}(\alpha_K)$ and that, for a *large class of BDs* B', the result of applying $\tilde{\alpha}_K$ to $b' \in B'$ coincides largely with $\alpha_K(b')$. Because $\tilde{\alpha}_K$ is an approximation, these induced partitions may or may not be failure domains w.r.t. the considered specification. For φ and $\tilde{\varphi}$, there are no restrictions and they may be completely different sets. An example is presented in the following section.

Example

Approximation $\tilde{\alpha}_K$ can be over-approximations that may contain fault-injected BDs (mutants) that do not necessarily contain faults or even are equivalent to the original BD; under-approximations that yield fewer mutants due to the omission of some transformations; or a combination of both. As an example, consider the class of off-by-one faults K, where a boundary condition is shifted by one value due to a logical programming mistake. For off-by-one faults an exemplary α_k transforms a relational operator into a different one or transforms the afterthought of a loop such that the loop is executed once too often.

To demonstrate over-approximation, $\tilde{\alpha}_k$ transforms the BD b with the fragment "`if (x<=50) { if (x==50) {`" into a set of BDs. This set includes the BD b', in which only the aforementioned fragment was transformed into "`if (x<=50) { if (x>=50) {`". b' is semantically equivalent to b and not faulty. Thus, the set of BDs created by $\tilde{\alpha}_k$ is larger than the set of BDs created by α_k (which by definition, contain faults of class k). To demonstrate under-approximation, one possibility is to limit $\tilde{\alpha}_K$ to consider only relational operators and not other operators (e.g. ++) for off by one faults. Then, the set of BDs created by $\tilde{\alpha}_k$ is smaller than the set of BDs created by α_k and there would indeed be further faulty BDs than created by $\tilde{\alpha}_k$.

For $\tilde{\varphi}$ an over approximation creates a larger set of inputs than actually cause failures. This can be the case when the fragment in b is unable to produce a failure (e.g. due to a guard), but inputs to reach the fragment are added to the set. In an

under approximation scenario of $\tilde{\varphi}$, actually failure-causing inputs are not added to the set. This is the case, if the path constraints to reach the fragment in b with symbolic execution (see Section 2.2) is not solvable with current constraint solvers.

Note that in the case of α above, the transformation is required as failure-causing inputs depend on the original relational operator and its replacement (e.g. replacing == with >= yields a different failure domain than replacing it by !=). In some cases of K only the image of $\tilde{\alpha}_K$ is required as the fault is independent of the transformation (i.e. its inverse image does not matter as the image will always cause a failure). This is the case for K being the class of faults leading to run time failures. As an example the fragments "`a = b / c`" and "`a = b + c`" may already fail, if (1) `a`, `b` and `c` have an integer data types and (2) a value of `0` for `c` in the first fragment and `MAX_INT` for `a` and `b`. Thus, operationalizations of defect-based quality assurance based on fault models do not always require $\tilde{\alpha}_K$, but are able to use the image of $\tilde{\alpha}_K$ as a form of fault patterns always leading to undesired behavior (see 8Cage in Chapter 5 for further examples only using the image of $\tilde{\alpha}_K$).

3.1.7 Effectiveness of Defect Models

Using defect models (i.e. fault or failure models) for test case derivation directly targets the fault or failure classes captured by these models. Defect models hypothesize what *could go wrong*. Thus, the captured classes of faults or failures are close to "what goes wrong". In this case, the derived test cases are likely to detect a fault and fulfill the first part of the definition of a good test case above. Intuitively, the better a defect models captures problems ("what goes wrong") in practice, the more effective it is.

Formally, we can re-use the model of Weyuker and Yeng [150] at the beginning of this chapter for the considerations on effectiveness by comparing defect-based testing with random testing. Again, random testing is the gold standard of software testing as it requires (almost) no test derivation effort. For the comparison, we need to adjust the formulas of P_r to P_{rnd} and P_p to P_{defect} in Chapter 3 to defect models. P_r concerns randomly (uniformly) sampling n elements from the input space of a BD b written to specification s. Let an arbitrary fault class L give rise to the failure domain $\varphi_L(j, s)$. Then, the probability of causing at least one failure with n tests (with redrawal) resulting from possibly multiple faults of fault class L and under the assumption of uniformly distributed failure-causing input is [150]

$$P_{rnd}(b, s, n, L) = 1 - \Big(1 - \frac{|\varphi_L(b, s)|}{|\mathrm{dom}([\![b]\!])|}\Big)^n.$$

Note that, in the comparison, we use $\varphi_L(b, s)$ and not its approximation $\tilde{\varphi}_L(b, s)$ to get a clear hypothetical picture of the comparison. In reality, it is not computable and would make testing expendable. For defect-based testing in the model of Weyuker and Yeng [150], the sampling is performed using $\varphi_L(b, s)$ leading to the probability to

provoke at least one failure of

$$P_{defect}(b,s,n,L) = 1 - \Big(1 - \frac{|\tilde{\varphi}_L(b,s) \cap \varphi_L(b,s)|}{|\tilde{\varphi}_L(b,s)|}\Big)^n.$$

By analogy to Weyuker and Yeng [150], the effectiveness of a defect model for defect class L (i.e. fault class L or failure class L respectively) applicable to a behavior description b and a given specification s "iff random testing is significantly worse than defect-based testing"' [128]. Formally, this condition can be expressed as

$$P_{rnd}(b,s,n,L) \ll P_{defect}(b,s,n,L)$$

, or in an extended form as

$$\frac{|\varphi_L(b,s)|}{|\mathrm{dom}([\![b]\!])|} \ll \frac{|\tilde{\varphi}_L(b,s) \cap \varphi_L(b,s)|}{|\tilde{\varphi}_L(b,s)|}.$$

As given by intuition, "a defect model is effective if the average failure rate of the behavior description is far smaller than the average failure rate in the set $\tilde{\varphi}_L$" [128]. The same holds for the approximation $\tilde{\varphi}_L$ and modeling the above in the model of Gutjahr [62]. Note that, the number of test cases being drawn is obviously negligible, if the same number of test cases is drawn. Thus, we can assume n to be fixed in the following.

One consideration implicitly addressed above is the applicability of a defect model. Run time failures, such as arithmetic overflows are safely handled by floating point data types and access violations are not expected when programming purely in Java. Thus, "*Effective fault models* for a domain-, company- or technology-specific set of specifications $S \subseteq \mathcal{B}$ are defined using a (hypothetical) set of behavior descriptions $B_S \subseteq \mathcal{B}$ realistically written w.r.t. these specifications S" [128]. This grounds the reasons for the variation points of defect models to be exactly domain, test level and application in the defect model lifecycle framework (see Section 1.4).

To reason the number of specifications from S (including all accordingly developed behavior descriptions) for which the effectiveness of defect-based testing is significantly higher than or at least equal to random testing, the following equation is defined in [129]

$$n_S = \Big|\Big\{s \in S : \begin{array}{l} \big|\{b \in B_S : P_{prt}(b,s,L) \gg P_{rnd}(b,s,L)\}\big| \\ \quad \gg \big|\{b \in B_S : P_{prt}(b,s,L) \not\gg P_{rnd}(b,s,L)\}\big| \end{array}\Big\}\Big|$$

Thus, "a specific defect model is *effective* if this number is "high" for a given class of specifications S that define a domain of interest" [128].

Considering the number of specifications from S above, n_S must be far larger than the number of specifications from S for which the defect model is *not* effective to be an effective defect model.

$$n_S \gg \Big|\Big\{s \in S : \begin{array}{l} \big|\{b \in B_S : P_{defect}(b,s,L) \gg P_{rnd}(b,s,L)\}\big| \\ \quad \not\gg \big|\{b \in B_S : P_{defect}(b,s,L) \not\gg P_{rnd}(b,s,L)\}\big| \end{array}\Big\}\Big|.$$

This definition of defect models is based on [128, 129], who state that "intuition that a defect model is "better" if it is more generally applicable, that is, if many realistic behavior descriptions from a given domain potentially contain instances of the respective defect class." This result points to the relevance of certain faults in certain domains could be demonstrated using defect models ex post to quality assurance. This gives rise to the idea to use defect models for the classification of test cases as defect models naturally are able to give the target/purpose of a test case. However, this classification is not fully automatable.

Note that, using the definition of the generic defect model and all considerations, empirical demonstrations of effectiveness could already lead comprehensive effectiveness results for one specification by selecting only one accordingly developed behavior description. However, the approximation $\tilde{\alpha}$ and $\tilde{\varphi}$ are likely dependent on the behavior description used to obtain the results. Thus, a representative set of behavior descriptions must still be chosen for an empirical evaluation.

The considerations above yield effectiveness for defect models iff they detect any faults. This is not very realistic as faults endangering the safety may be more critical than simple typos giving a warning in a log file. To take the severity into account Pretschner [128] defines a fault classification with a respective cost function. This allows risk-based quality assurance using defect models and, if a classification of test cases using defect models is used, to assess consequences of any test case failing.

3.2 Instantiation

In the previous section, a generic defect model was defined to capture the fault hypothesis based on knowledge and experience and the result of the literature survey. To demonstrate the usefulness of the generic defect model, we show existing fault and failure models (i.e. the related work) to be an instance of it in this section. Particularly, we formalize existing defect models and, therefore, are able to reveal their encapsulated knowledge and experience concerning their described defects. If we can show existing fault and failure models to be instances, (a) the generic definition is capable of capturing fault hypotheses and (b) its operationalizations are able to perform defect-based quality assurance. To this end, our literature survey considers existing fault and failure models explicitly stated as such in the area of quality assurance. Of course, there are many more defect models in literature and practice. However, our aim is to survey a representative set and give an idea how defect model instantiations are mapped back to the generic defect model in different areas of quality assurance. Since some of the stated fault models are failure models according to our definition, these are introduced as failure models. Although literature concerning reviews and inspections does not contain the key words of fault or failure model, we include reviews and inspections in the instantiations to also describe instantiations in this area of analytic quality assurance. This list is not intended to be exhaustive of all possible defect

models, but aims to guide the casting of other defect models in literature and practice into our generalization.

The instantiations are described using the respective α, φ, and their approximations. $\tilde{\alpha}$ describes a fault as a (possibly higher order) mutant. $\tilde{\varphi}$ defines a set of failure-causing inputs. We assume that the BDs are correct w.r.t. their specification before the transformation $\tilde{\alpha}$ and incorrect afterwards as mentioned in the definition.

3.2.1 Stuck-at

The stuck-at fault model [111] is known for automated test pattern generation in the hardware industry. It assumes that a manufacturing defect is present in one or multiple logic gates or subcircuits such that regardless of their input, their output is always the same. The transformation α for stuck-at is the transformation of one circuit into another circuit where one subcircuit is replaced by 1 or 0. For our purposes, this replacement can equivalently be performed at the level of logical formulas f that represent equivalent circuits. For each application of α, that is, each element that is picked by α, φ then add the inputs $\{i : [\![f]\!](i) \neq [\![\alpha(f)]\!](i)\}$. Operationally, depending on the formalism used, a SAT solver is adequate to compute $\tilde{\varphi}$ for a specific circuit.

As an example, assume a function (a circuit) $f = (a \wedge b) \vee (b \wedge c)$. As one exemplary stuck-at-0 fault, α introduces a permanent output of 0 for the subformula (the gate) $b \wedge c$, that is, $\alpha(f) = (a \wedge b) \vee 0$. It is easy to verify $\tilde{\varphi}$ yields $(a = 0, b = 1, c = 1)$.

3.2.2 Division By Zero

Division by zero is a classic fault in many BDs. It typically happens if developers do not perform input sanitization (i.e. check for a value of 0 for the divisor) prior to a division, or when the value 0 for the divisor was not assumed possible in the BD's context. Let us concentrate on the former case (and this in itself is an example of how to under-approximate α by some $\tilde{\alpha}$). The transformation $\tilde{\alpha}$ removes the sanitization mechanisms (if they exist) from BD p. The resulting $\tilde{\varphi}$ is $\{i : [\![p]\!](i) \neq [\![\tilde{\alpha}(p)]\!](i)\}$ representing all inputs causing the divisor to be 0 at the point of the division.

For a BD p_d with input parameter i and $p_d = f_x(i)/f_y(i)$ developed according to specification s to divide two integers, let $\tilde{\alpha}(p_d) = f_x(i)/f_z(i)$ be the replacement of the function f_y including some sanitization mechanism by an f_z without sanitization. Then the result of $\tilde{\varphi}(p_d, s)$ is $\{i : [\![f_z^{-1}]\!](i) == 0\}$. Operationally, depending on the formalism used, a symbolic execution tool is an adequate tool to compute some $\tilde{\varphi}$ for a specific BD and a specific set of mutation operators.

3.2.3 Mutation Testing

While mutation testing aims at assessing test suites and targets small syntactic faults, mutation operators do describe fault models that we can use for our purposes (for

instance, Ma et al. provide several direct relationships between some mutation operators and faults [102] where the coupling hypothesis appears immediately justified). Mutation operators are intuitively captured by our transformation α—in fact, we see α as a reasonable higher order mutation operator. Since α is applied to a program, the general considerations of Section 3.2.2 with the resulting $\tilde{\varphi}$ of $\{i : [\![p]\!](i) \neq [\![\alpha(p)]\!](i)\}$ also apply here. Consequently, symbolic execution tools are promising for computing $\tilde{\varphi}$.

As an example, take a program p_m with specification s and input parameter x and $p_m = 1$ if $x < 10$ and $p_m = 0$ otherwise. Using a mutation that transforms $<$ to $\leq$, let $\alpha(p_m) = 1$ if $x \leq 10$ and $\alpha(p_m) = 0$ otherwise. Then $\tilde{\varphi}(p_m, s) = \{10\}$ for the input domain I as only the path for input 10 changed for p_m.

3.2.4 Finite State Machine Testing

In finite state machine (FSM)-based testing, typical fault models are based on output and transfer faults. As one typical example that easily generalizes, let us consider BDs in the form of deterministic Mealy machines $\mathcal{M}$ such that each $M \in \mathcal{M}$ is a sextuple $(\Sigma, \Gamma, S, s_0, \delta, \gamma)$ where Σ and Γ are input and output alphabets, S is the set of states, $s_i \in S$ is an initial state, $\delta : S \times \Sigma \to S$ is the transfer and $\gamma : S \times \Sigma \to \Gamma$ the output function.

Output faults occur when a transition yields a different output than specified in the output function. This deviation is the result of the transformation $\alpha_o : (S \times \Sigma \to \Gamma) \to 2^{S \times \Sigma \to \Gamma'}$ which models faults in the same way as in the stuck-at fault model (see Section 3.2.1: for a given transition, the correct output is mapped to another, incorrect output from a set $\Gamma' \supseteq \Gamma$). Analogously, transfer faults lead the FSM into a different state than specified in the transfer function, $\alpha_t : (S \times \Sigma \to S) \to 2^{S \times \Sigma \to S'}$ with $S' \supseteq S$ since the destination state of a transfer fault may be a new state not in the design of the original FSM [22]. In the following, we assume that the definitions of α_o and α_t are lifted to entire machines in the obvious way, that is, α_o and α_t are of type $\mathcal{M} \to 2^{\mathcal{M}}$. In the remainder of this paragraph, α refers to both α_o and α_t.

Finite traces $[\![M]\!] \in \Sigma^* \to \Gamma^*$ for a $M \in \mathcal{M}$ are pairs of (input, output) sequences that we assume to respect the transfer and output functions in an intuitive way (that is, they induce state changes that are captured by δ, and they model γ). φ then yields $\{i \in \Sigma^* : [\![M]\!](i) \neq [\![\alpha(M)]\!](i)\}$, which defines all those traces that are different in M and $\alpha(M)$ – these are the traces that exhibit faults.

Generally speaking, model checkers and dedicated algorithms on graphs are adequate tools for computing approximations $\tilde{\varphi}$.

Several related fault models have been described in the area of object-oriented testing [20] that model objects as finite state machines. One of them is sneak path, which describes that a message (i.e. a composite input) is accepted although it should not be. In the notion of an FSM, a sneak path is a an additional transition in the

transfer function and can be modeled by $\alpha_t : (S \times \Sigma \to S) \to 2^{S \times \Sigma \to S'}$ as described above.

Similarly, a trap door is the acceptance of an undefined message (i.e. a new letter in the alphabet), which causes the system to go to an arbitrary state. Intuitively, $\alpha_t : (S \times \Sigma \to S) \to 2^{S \times \Sigma' \to S'}$ reflects a trap door by introducing a new character to the alphabet $\Sigma' \supseteq \Sigma$ and a new transition leading to a possibly new state in $S' \supseteq S$.

This category of fault models can also be transferred to the area of feature transition systems of product lines, where they are also able to modify events [46].

3.2.5 Object-oriented Testing

For object-oriented testing, there are fault models catering to subtyping and polymorphism [121] in object-oriented programming. These are, for example, state definition anomalies (pre or post conditions are possibly violated by subtypes) or anomalous construction behaviors (i.e. the subtype shadows variables used by the constructor of the supertype). The general considerations for both fault models can be described by using transformations similar to α from Section 3.2.2, but at the level of pre- and post conditions rather than at the level of code.

For a state definition anomaly, let a class C contain a method $p^C_{sda}(x) = f(x); v := x\{v \neq NULL\}$; with post condition $v \neq NULL$ for some instance variable v, and a method $q^C_{sda}(x, z) = p^C_{sda}(x); if\ z\ then\ \{v \neq NULL\}\ g(v)$; where the precondition of function g is assumed to require the argument to be different from $NULL$. Class C' is a subclass of C where $p^{C'}_{sda}(x) = p^C_{sda}(x); h(x)\{true\}$; overrides method p^C_{sda} in C'. If the post condition of h in the definition of $p^{C'}_{sda}(x)$ does not imply $v \neq NULL$, then the inherited $q^{C'}_{sda}(x, z) = p^{C'}_{sda}(x); if\ z\ then\ \{v \neq NULL\}\ g(v)$; causes problems if the precondition of g is not met.

There are many different ways of violating pre- or post conditions, and it seems unlikely that these can be comprehensively captured by patterns of textual modifications of code. However, the modification of *explicitly provided or inferred pre- or postconditions* can be specified using α, the domain of which is inherited functions only; in our example, $q^{C'}_{sda}(x)$ is the only one. One possibility then is that $\alpha(q^{C'}_{sda}(x))$ computes to $p^C_{sda}(x); if\ z\ then\ \{v == NULL\}\ g(v)$; by modifying the precondition of function g. Intuitively, this models the possibility that an inherited function leads to a state where the specified precondition of g *cannot* be satisfied. If they exist, $\tilde{\varphi}$ yields $\{(x \mapsto i, z \mapsto true) : i \in \mathbb{N}\ and\ v == NULL\ before\ g\ is\ executed\ from\ within\ q^{C'}_{sda}(x))\}$ and all entries of the set would then provoke a failure when applied to method $p^{C'}_{sda}$ of an object of class C'. Possible technology for computing $\tilde{\varphi}$ includes symbolic execution.

3.2.6 Aspect-oriented Testing

The use of AOP has been shown to induce specific faults [29]. One such fault model concerns the failure to establish expected post-conditions and preserve state invariants.

The post-conditions and state invariants introduced in the basic functionality are contracts that should be preserved in the weaved code. This fault is analogous to object-oriented testing where it can be caused by inheritance (see Section 3.2.5).

A second fault model consists of incorrect changes in the exceptional control flow. Whenever features having their own exception handling are introduced, an exception may trigger the execution of a different catch block than the one intended by the basic functionality. For this fault model, let $p_a = \text{try}\{f_x(x);\} \text{ catch (Exception e)}\{f_e(e);\}$ $\text{try}\{f_y(x);\} \text{ catch (Runtime}- \text{ Exception ex)}\{f_{ex}(ex);\}$ with input parameter x be a program with exception handling f_e for the original functionality f_x and exception handling f_{ex} of an introduced feature f_y. Also let $\tilde{\alpha}(p_a) = \{\text{try}\{f_x(x); f_y(x);\} \text{ catch}$ $(\text{RuntimeException ex}\{f_{ex}(ex);\} \text{ catch (Exception e)}\{f_e(e);\}\}$ be the transformation of p_a, which merges both try/catch blocks and extends the exception handling. Then, the result of $\tilde{\varphi}$ contains inputs triggering a runtime exception (or one of its subtypes) in f_x to let f_x use exception handling f_{ex} instead of the intended f_e. Again, symbolic execution tools can be used on the program to compute $\tilde{\varphi}$.

3.2.7 Performance Testing

One fault model—there are multiple others—for performance testing [118] describes one or multiple hardware component failures or malfunctions causing the BD to have a degraded performance. Such failures or malfunctions could be related to hard drive, network or memory problems. If we model the hardware and software as an FSM, then the transformation α can simulate a malfunction by removing states and transitions to these states. Thus, α requires the system to take more transitions thereby taking more steps for the same computation or blocks the system from ever reaching its desired state causing a failure. Precisely this modification of the transfer function is shown in the FSM fault model in Section 3.2.4.

3.2.8 Concurrency Testing

Fault models concerning in testing concurrent systems regard atomicity and order violations [99], in addition to deadlock and livelock problems. For an atomicity violation the developer did not implement a monitor (or implemented it in the wrong way). Let m be a monitor and $p_{atom} = \text{monitor\_lock}(m); f_x(x); \text{monitor\_unlock(m)}; ||\text{monitor\_lock}(m);$ $f_y(x); \text{monitor\_unlock(m)};$ with input parameter x be a program using this monitor and $f||g$ be defined as the execution of f and g in parallel. Also let $\alpha(p_{atom}) = f_x(x); ||f_y(x);$ be a transformation of p_{atom} removing its usage of monitors. With the usage of concurrency, the semantics of the program are also influenced by the schedule of execution. An atomicity violation typically changes the output of the program when using different schedules while the input remains the same. Thus, the input space must be extended by adding the schedule to the input vector. Then, the resulting $\tilde{\varphi}$ yields a set of those inputs for which the output is different when only using a different schedule.

For order violations the developer made a wrong assumption about the order of execution of statements. No α is required as the developer assumed an execution order s_0, but did not enforce it. Thus, the set produced by φ aims to break the assumption by executing all schedules different from s_0 and checking whether the semantics have changed.

3.2.9 Security Testing

In security testing, one approach to find faults w.r.t. given security properties (e.g. confidentiality and integrity) using a formal system model is presented by Büchler et al. [23]. The transformation α is reflected in semantic mutation operators (see Section 3.2.3) for a model of the system. These operators modify the model such that an assumed vulnerability in the respective implementation is present. α is therefore described by these mutation operators. The idea to induce the failure domain φ is to have a sequence of actions (i.e. a trace) that violate the security property. Practically, this is performed by using a model checker to find this trace τ and executing τ on the implementation of the system. Since the model checker may not return all traces in useful time, φ must be approximated by $\tilde{\varphi}$ and $\tilde{\varphi}$ also contains these unknown traces. Thus, $\tilde{\varphi}$ can be constructed in the same way as in Section 3.2.4.

3.2.10 Limit Testing

The well-known failure model of boundary value analysis (based on the category partition method [122]) is underlying limit testing. As per definition of a failure model, the transformation α is unknown (or, analogously, models all those possibilities to get a BD's treatment of limit values wrong). For the failure model, the set of failure-causing inputs also called failure domain produced by $\tilde{\varphi}$ is described. In this failure model, $\tilde{\varphi}$ requires an input space partition created from any criterion (e.g. code-based test selection criteria). The resulting set of $\tilde{\varphi}$ is then produced by an algorithm. Let γ be an input space partition for the domain $\mathcal{D}$ and partition it into non-empty pairwise disjoint subsets called blocks $D_0...D_k$, where $0...k$ is the identifier of the block and $i \in 0...k$. In addition, let the elements of $\mathcal{D}$ as well as each D_i have a total order and the functions min and max determine the largest and smallest elements of a given partition D_i. Then the algorithm for $\tilde{\varphi}$ returns the set $\bigcup_{1 \leq i \leq k} \{\{i : i \in D_i \wedge i = min(D_i)\} \cup \{i : i \in D_i \wedge i = max(D_i)\}\}$.

As an example, let a partition with three integer blocks contain the numbers -2,147,483,648 to 0, 1 to 100 and 101 to 2,147,483,647 in the respective blocks. $\tilde{\varphi}$ then produces the set containing the inputs -2,147,483,648 and 0 from the first, 1 and 100 from the second and 101 and 2,147,483,647 from the third block.

3.2.11 Combinatorial Testing

The failure model of combinatorial, or n-wise, testing [89] states that only a combination of 2, 3 or n parameters causes a failure, but not all possible combinations of parameters. It thus provides a test selection criteria requiring fewer test cases than exhaustive testing (i.e. all combinations). Since this is a failure model, the $\tilde{\varphi}$ can be computed using an algorithm. The algorithm for multi-way combinatorial testing is called IPOG with its variants IPOG-D and IPOG-C formalized in Lei et al. [96] and Yu et al. [154]. The result of $\tilde{\varphi}$ contains a minimal set of test cases covering all 2-way, 3-way or n-way interactions. Note that, there are multiple possible partitions and an arbitrary minimal partition can be selected (e.g. in the case of 3 parameters with 3 values and all 2-way interactions to be tested, there exist 12 possibilities to select the minimal number of test cases being 9) [96, 154].

As an example, reconsider function f from Section 3.2.1. One exemplary set of inputs testing all pairwise combinations for f is (0,0,0), (0,1,1), (1,0,1), (1,0,0). This set of inputs would find, but is not limited to, the faults described by the stuck-at fault model.

3.2.12 Exploratory Testing

In exploratory testing (see Section 2.1.2), testers manually perform the task of test case selection and execution at the same time on a system testing level. They use their knowledge and experience leading to ad-hoc defect models. Most of the times failure models are used since a fault hypothesis is hard to infer at the system testing level. For other levels of testing, there is the concept of *error guessing* (see Section 2.1.2), where fault and failure models are created ad-hoc.

For performing exploratory testing, Whittaker [151] gives explicit instruction in the form of tours. Tours tell the testers to exercise certain functionality in a certain way and some tours constitute failure models. For the Antisocial tour, the tester aims to enter "either the least likely inputs and/or known bad inputs. If a real user would do *a*, then a tester on the Antisocial tour should never do *a* and instead find a much less meaningful input" [151]. Thus, the estimated failure domain $\tilde{\varphi}$ for the Antisocial tour includes all unlikely/least likely inputs.

3.2.13 Reviews and Inspections

In contrast to the ability of software testing to reveal failures, reviews and inspections reveal faults and have different reading techniques to do so. One of these reading techniques is checklist-based reading/review and has inherent fault models. As is the case with exploratory testing, checklists may contain simple questions such as checking for a "shall" in each requirement as required by e.g. DO-178 [48]. However, they typically also/only contain fault-based questions, which have been created due to defect in the past. Many examples for fault-based checklists can be found in McConnell [112].

These checklists target a plethora of systems as they are independent of specification, domain, implementation language etc. of the system. Example questions are “Are all inputs to the system specified, including their source, accuracy, range of values, and frequency?” [112] or “Does the design have low complexity?” [112]. These questions give rise to an α, which omits details about the inputs or increases the complexity of the design of the system. The respective result of φ then are all inputs, for which the omission of details or the increase of complexity makes a difference in output or any non-functional requirement (e.g. performance). If certain parts of the BD are known to typically contain defects, this can be used as a failure model. φ then yields all inputs leading to these parts to focus the review particularly on these parts. Thus, using α andφ in such a way enables to focus on certain aspects of the BD under review and inspection to particularly look for the defects described in the fault models. In addition, only using φ enables to focus on particular parts of the BD described in a failure model. Thus, operationalizations for reviews and inspections can never completely automatically reveal the defect, but always point the software engineer towards the defect. However, check lists can be constructed from the defect knowledge by assembling all knowledge concerning the context of the system under test. If manual effort of categorization is involved, this assembly can be (semi-)automatic.

3.3 Operationalization

The previous section showed the generality of the generic defect model by showing a representative set of defect models from the literature to be instances of it. This list does not claim completeness w.r.t all existing defect models, but we believe other defect models to be castable into the instantiations above. To explicitly use defect models in practice based on the generic defect model, the operationalization of the described knowledge in automatic test case generators must be enabled. Particularly, the test cases should be good test cases. Thus, they must directly target a fault and be cost-effective. Paving the way for the operationalizations in the following chapters of this thesis, this section characterizes the operationalization of the generic defect model by defining generic operationalization scenarios. These generic operationalization scenarios are then instantiated in the following chapters and their underlying defect models are mapped back to the generic defect model. If such a mapping is possible, it shows them to be operationalizations based on the generic defect models.

In general, operationalizations of defect models perform test input selection and are different for fault and failure models. Both always require the availability of a behavior description.

As fault models capture the underlying faults and faults are typically only known on the lower levels of testing, they are mostly used on the unit testing level. The operationalizations of fault models may require the availability artifacts such as a specification / model and may yield an oracle [128]. In contrast, failure models are typically

used on the integration and system testing level as only the set of the underlying faults (unless clearly visible) is known. In the following, four operationalization scenarios are discussed and linked to the respective chapters.

In the first operationalization scenario, the behavior description is a model developed according to a specification, which is used for the manual derivation of source code. The model is transformed by α to derive defect-based test cases for the source code from the model. The transformed model contains typical faults resulting from the manual derivation of source code. Obviously, it does not satisfy the specification of the original model. Using the transformed model to generate test cases targets exactly the introduced faults and executing them on the manually derived source code is able to reveal them. An exemplary operationalization of this type is the security testing performed by Büchler et al. [23].

In a second operationalization scenario, the behavior description is a model with only an implicit specification of "no run time failures", which is used for the automatic derivation of source code. The operationalization assumes the transformation α_K with K being fault leading to run time failures has been applied to the model and looks for its image (i.e. a smell/bug pattern). Upon finding the image, the operationalization derives the source code and creates a test case targeting the exploitation of the pattern. The test cases represent evidence for the existence of an actual fault by its execution. An exemplary operationalization of this type is 8Cage for Matlab Simulink/Stateflow models in Chapter 5.

In a third operationalization scenario, the behavior description is source code with only an implicit specification of "no run time failures". Again, the operationalization assumes the transformation α_K with K being fault leading to run time failures has been applied and the implementation to contain faults. Using tool-based static analysis with lint-like [81] tools as operationalization, potential faults can be located, but no test cases are created. Using fuzzers [21] or symbolic execution tools (e.g. [25]) as operationalization, the source/binary code can be executed, test cases created and faults detected.

Finally, the operationalization scenario of failure models purely utilizes the input/output relation of the behavior description as failure models capture test strategies. These test strategies are typically given as an algorithm and can be directly implemented by the operationalizations. Example operationalizations include (1) a plethora of tools performing limit testing (e.g. [131, 146]), (2) tools to perform combinatorial testing [63], (3) the integration system testing operationalization to detect superfluous/missing functionality in components OUTFIT in Chapter 6, (4) the control system testing operationalization called Controller Tester in Chapter 7. (1), (2), (3) and (4) then create failure-based test cases.

For reviews/inspections, the fault and failure models can be used and are operationalized in defect-based check lists. One way to gain these check lists is using defect taxonomies for requirements [54, 55]. These can be used for the early detection of

requirements defects during an inspection of requirements using perspective-based reading [138]. A second way is directly re-using the check lists by McConnell [112].

3.4 Conclusion

In this chapter, we started by defining a test case to be good, if it detects a potential, or likely, fault with good cost-effectiveness. We also initially found some test selection techniques to operationalize this definition speculating their encapsulated knowledge and experience to lead to a fault hypothesis.

By understanding the encapsulated knowledge and experience and generalizing the fault hypothesis to generic defect models, we are able to formally capture the fault hypothesis. Defect model is the umbrella term for fault and failure model. Our fault models consist of syntactic transformations (higher-order, or semantic, mutants) and an input space partition. Our failure models consist only of an input space partition. While fault models target specific (classes of) faults in the system, failure models capture test selection strategies. The generality of this definition was evaluated by showing a representative selection of existing fault and failure models (i.e. related work) attained from a literature survey in quality assurance to be an instance of the generic defect model. This selection is not intended to be exhaustive of all possible defect models but aims to guide the casting of other defect models in literature and practice into our generalization.

By operationalizing defect models, we arrived at defect-based quality assurance as one way to derive good test cases / check lists. Particularly, the operationalizations of defect models are automatic test case / check list generators directly targeting the described faults/failures. For the operationalization of defect models, we describe generic operationalization scenarios casting existing operationalizations into the scenarios and guiding the operationalization of defect models in the future. Particularly, these scenarios pave the way for operationalization in the following chapters, where operationalizations are created as a direct instance of the generic defect model. These operationalization are evaluated for their effectiveness and efficiency and their defect models are mapped back to the generic defect model casting them into defect-based quality assurance.

The definition of fault and failure models has close relations to the field of mutation testing (see Section 3.2.3) and its adjacent field of automated program repair [115]. By defining fault models with a transformation that is essentially a cleverly chosen higher order mutant, we connected the notion of using fault injection for test case assessment to using mutants for test case derivation. Whereas test cases in mutation testing are generated to arrive at a test suite to kill all mutants [125], we create test cases based on the assumption of mutation (i.e. fault injection) having indeliberately been performed by the software engineer.

In addition to its ability to derive good test cases, defect-based quality assurance based on defect models yields several other advantages. These advantages concern risk assessment (1), fault tolerance (2) and fault localization (3). For risk assessment (1), the use of defect models can increase the probability of a particular targeted class of faults to not be present in the system after testing. Fault tolerance (2) can be evaluated by using defect models that target faults / failures handled by the fault tolerance systems. It is also noteworthy that classes of faults / failures in fault / failure models can be associated with the impairment of quality attributes in the system (see Section 7.2 for examples). Thereby, testing using these defect models can reduce the risk of impairment in the final product. When using fault models, the fault localization effort (3) can be estimated and reduced since the transformations describe what to look for and where.

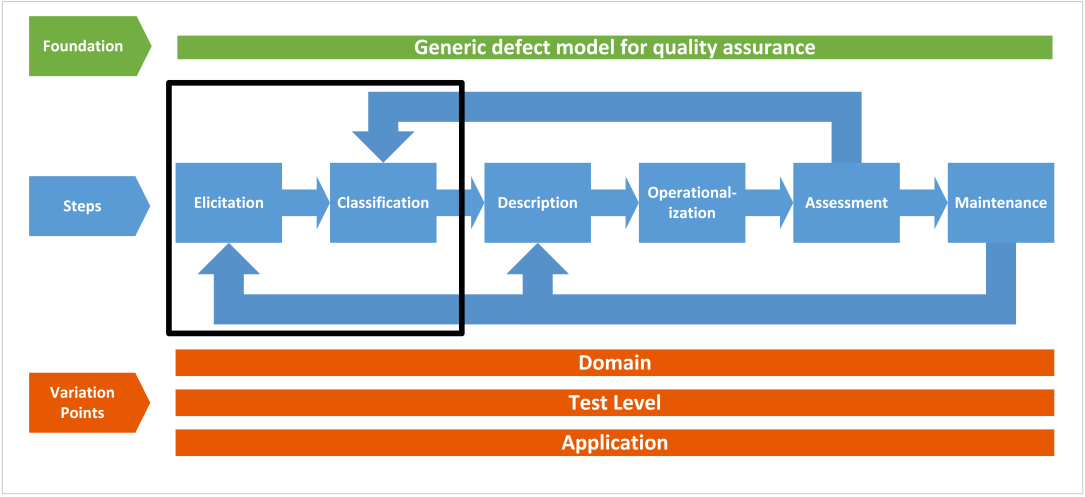
4

Elicitation and Classification

The previous chapter on the generic defect model forms the foundations for a systematic and (semi-)automatic approach to defect-based quality assurance based on defect models. By defining the generic defect model and creating generic operationalization scenarios, it can be used to describe and operationalize defect knowledge. The operationalization of defect models possibly requires the resource-intensive task of implementing (and possibly maintaining) a piece of software for the generation of test cases. Starting such a task requires the careful consideration of the involved cost benefits. In the end, the operationalization of defect models is only cost-effective if the described faults/failures are likely to be present in the system. To arrive at common and recurring faults likely to be in the system, the defect model lifecycle framework presents the deliberate activities of classification and elicitation (see Figure 4.1) in the planning step to aid with this strategic decision.

Eliciting and classifying the "relevant" defects for specific contexts is crucial for taking the decision whether to invest in the description and operationalization of defect

Figure 4.1: Position of elicitation and classification in this thesis



knowledge using defect models. Therefore, the elicitation and classification activities focus on common and recurring defects in the context of a project, organizational unit or organization. To anticipate the effectiveness of defect models and rationalize the strategic investment, defect elicitation and classification methods hence need to be comprehensive and to allow for frequency, and possibly severity, assessments. Common defects are a starting point to yield effective defect models as behavior descriptions are more likely contain them. If these defects are then also recurring defects, the investment into their (semi-)automatic detection yields recurring benefits.

Recalling the definition of defect from Section 2.1.1, a *defect* is an umbrella term including all faults, errors, failures, bugs, and mistakes made when designing or implementing a system. Similar to the notion of quality in general, which constitutes a multifaceted topic with different views and interpretations [57, 88], defects and especially their relevance, too, are something relative to their context. That is, a defect that might be critical to one project might be without relevance to the next. The systematic integration of (domain-specific) defect detection and prevention mechanisms into the quality assurance (QA) of particular socio-economic contexts, e.g. a company or a business unit is therefore crucial.

The approach presented in this chapter is Defect ELIcitation and CLAssification (DELICLA) and is based on [70] by the author of this thesis. It is an in-process elicitation (i.e. eliciting defect models while products are being developed) and classification approach for common and recurring defects. Particularly, DELICLA is a qualitative method with particular focus on interviews for the data collection and Grounded Theory for the analysis. One reason for relying on Grounded Theory coding principles is the categorization as well as the possible elaboration of cause-effect relations for defects. Once the defects are identified, they are integrated in a taxonomy: technical or process-related. The qualitative nature makes the approach agnostic to specific contexts/domains while, at the same time, always yielding context-specific results. By relying on an adaptable defect taxonomy, we follow the baseline of Card [28] and Kalinowski et al. [85], who note that it is beneficial to “tailor it to our [...] specific needs”.

To evaluate of DELICLA, we perform a field study. We want to gain insights into its appropriateness to (1) elicit and classify defects in specific contexts of different application domains; (2) the extent to which it leads to results useful enough for describing and operationalizing defect models; and (3) if there are immediate additional benefits as perceived by practitioners. In the field study, we apply our DELICLA approach to four cases provided by different companies. In each case, we conduct a case study to elicit and classify context-specific defect classes. The goal of our study is to get insights into advantages and limitations of our approach; this knowledge supports us in its further development.

4.1 Related Work

In the classification step of our approach, we provide a basic defect taxonomy / classification. Efforts to create a standardized defect classification for the collection of empirical knowledge have been made in the past [149]. However, there has not yet been a general agreement as defects may be very specific to a context, domain or artifact. This leads to a plethora of taxonomies and classifications techniques in literature and practice. In the area of taxonomies, Beizer [17] provides a well-known example for a taxonomy of defects. Avizienis et al. [8, 9] provide a three-dimensional taxonomy based on the type of defect, the affected attribute and the means by which their aim of dependability and security was attained. IEEE standard 1044 provides a basic taxonomy of defects and attributes that should be recorded along with the defects. Orthogonal Defect Classification (ODC) [32] is a defect classification technique using multiple a-priori fixed classification dimensions. These dimensions span over cause and effect of a defect enabling the analysis of its root cause. Thus, defect trends and their root causes can be measured in-process. Apart from these general classification approaches, there are approaches specifically targeting non-functional software attributes such as security [7, 91, 92] or, based upon ODC, maintenance [100, 101]. Leszac et al. [97] even derive their classification aiming to improve multiple attributes (i.e reliability and maintainability). Our approach, presented next, deliberately chooses to employ a minimalistic/basic defect taxonomy to stay flexible for seamless adaptation to specific contexts and domains. Our lightweight taxonomy enables the approach to be in tune with the expectations/prerequisites of our project partners (see RQ3 in Section 4.3). In contrast to ODC, we are not generalizing our taxonomy to be "independent of the specifics of a product or organization" [32], but rather require adaptability to context. In addition, we do not aim to capture the effects of defects (other than the severity where possible) as it is not required for the elicitation and classification of defects for defect models. However, our taxonomy can be mapped to ODC's cause measures by (1) refining the categories of technical and process-related defects into defect types and (2) using the associated tasks of the role of the interview partner as defect trigger. The severity in our taxonomy can directly be taken in ODC's effect measures, but other required measures such as impact areas, "reliability growth, defect density, etc." [32] must be elicited in addition.

4.2 DELICLA: Eliciting and Classifying Defects

A first decision in the design of DELICLA was to use a qualitative approach for defect elicitation and classification. The central aspect of our approach is further its inductive nature where the focus is on generating theories rather than testing given ones. That is, the approach makes no a-priori assumptions about which defects might be relevant in a specific context, yet our hypothesis is that common and recurring defects exist in

the context. In addition, we rely on circularity yielding further defects, if the approach is repeatedly used in the same or similar contexts.

There exists a multitude of techniques employable in qualitative explorative approaches with the ability to take a defect-based viewpoint. These established techniques have been explored with respect to three goals: (1) cost-effectiveness in their application, (2) comprehensiveness in the obtained results, and (3) ability to establish a trust relationship during the data collection.

Trust is important because humans generally are reluctant to disclose potential problems in individual project environments [61, 85]. The assessed techniques include techniques for document analyses, interview research, participant observation, and creativity techniques such as brainstorming.

Since bug reports in a bug tracker directly represent defects, they provide a starting point for the data collection. However, bug reports typically differ greatly in quality. In addition, bypassing bug trackers, many defects are directly communicated to the respective developers. This raises questions concerning the comprehensiveness of this method. Nonetheless, as only the bug tracker database has to be examined, the cost to perform this method is low, and trust does not need to be established with project participants. However, customizing the in-process measurements of ODC using additional fields in a bug tracker, may yield defect models after a (possibly automatic) analysis.

The quality of results in brainstorming or focus groups is also questionable since it is hard for humans to have an open discussion about their own software defects [61, 85]. This causes strong variations in comprehensiveness and lowers its cost-efficiency. The required trust relationship can be established as these techniques are performed in a closed environment with a moderator present.

The strong variations in comprehensiveness and cost-effectiveness also makes participant observation questionable for the elicitation of defects for defect models. The cost is low since work on projects can continue, but the results are limited to the time given for the observation. Thus, this technique is unable to reflect about past defects or speculate about future defects. However, a trust relationship to the participants can be established as the observer is present.

The comprehensiveness of a questionnaire, and therefore its cost-effectiveness, are also questionable. While the time to answer questions is limited, it does not allow to get any follow-up explanation of the defect. Thus, the specific context of the defect can possibly not be established. In addition, a trust relationship can't be established due to the lack of physical presence and eye contact [61].

Due to their ability to be comprehensive and the possibility to establish a trust relationship [61, 73], personal interviews were chosen as technique in the DELICLA approach. This allows to fully explore the participants' perspectives in their context while adopting their vocabulary. Using this technique in a semi-structured form yields

the ability to guide the interview [61] along predefined questions without interrupting their flow of words.

For the analysis of the collected data, we employed Grounded Theory [58] and code the answers as described by Charmaz [30]. In a manual coding step, we code all mentioned defects including their cause and effect. These codes are then organized in a hierarchy representing a defect taxonomy. Following this form of open coding, we apply axial coding to the results to explicitly capture relationships between defects as well as possible causes and effects. In some cases, we apply selective coding to capture possible causalities between defects. Our DELICLA approach consists of the three steps explained next: (1) *Preparation*, (2) *Execution*, and (3) *Analysis*.

4.2.1 Preparation

The first activity in the preparation step is to create a pool of potential interview candidates (i.e. the participants). Candidates are identified with the project partner by focusing on their projects or domains of expertise. The selection of interview candidates is performed by the interviewer or project partner yielding a variation point. In case the interviewer is able to select the candidates, the context of the study (e.g. the projects and teams focused on) and the expenditure of time for the project partner must be exactly defined. Key aspects to consider before selecting any interview partners are the organizational chart and the assessment of their potential contributions by their managers. The order of interviews was from best to least contributing according to the executives' opinions; and lowest to highest branch in the organizational charts [61]. When interviewing the best performing, the interviewer is able to assess the maximum capabilities of team or project members thereby gaining a perspective of what can be achieved. Subsequent to interviewing executives on higher branches of the organizational chart, defects collected in lower branches can be discussed and used to devise first indications towards future measures. Thus, even if the project partner selects the interview partners, the interviewer should be able to get an overview using an organizational chart and set the order of the interview partners.

After the interview partners have been selected, they are informed about the upcoming interview and their required preparation. An interview preparation sheet is given to them detailing the purpose of the interviews and the questions to be prepared. In our studies, we used the open questions seen in Table 4.1 for preparation similar to those presented by Charmaz [30]. An extension point are additional questions. Depending of the context, questions such as "How meticulously is the SCRUM methodology followed?" may be added. When informing the interview partners, the responsibles on the project partner's side must also be named for potential inquiries of interview partners about internal procedures. Interviews are not part of the everyday working life of the interview partners and may cause feelings of nervousness to anxiousness. To mitigate these feelings, the description of the purpose of the interviews is very detailed

and emphasizes the defect-based view on tools, processes and people in defect models for quality assurance.

Table 4.1: Instrument used for the interview preparation sheet.

ID	Question
Q1	What are the classical faults in the software you review/test?
Q2	What does frequently/always go wrong? With which stakeholder?
Q3	What was the "worst" fault you have ever seen? Which one had the "worst" consequences?
Q4	Which faults are the most difficult ones to spot/remove?
Q5	What faults were you unaware of before working in your context?
Q6	What faults do you find most trivial/annoying?
Q7	What faults do engineers new to your area make?

Each interview requires 30 minutes for the preparation by the interview partner and 30 minutes for the actual interview; usually a negligible amount of time. This lets interview partners prepare so that they "can be prepared to speak directly to the issues of interest" [61]. When planning the concrete times for the interviews, every two interviews include a 30 minute break at the end. In case any interview takes more time than expected, this break is used to prevent the accumulation of delay for the following interviews.

The interviewer must also prepare w.r.t. the processes and tools employed by the interview partners and their roles at the project partner. To establish the trust relationship, an address of reassurance is prepared to be given before the interview. In addition, the room is small and any distractions are removed. All technology used during the interviews is tested beforehand and interviews are recorded as suggested by Warren [61].

4.2.2 Execution

With trust and comprehensiveness of results our main objectives, we follow the basic principles of interview research: At the beginning of the interview, the interview partner and interviewer agree on a first name basis. This basis takes down psychological walls and is a key enabler of an open discussion later in the interview. When sitting down, the interviewer never faces the interview partner as it creates the sense of an interrogation [61]. The interview starts with a short introduction consisting of a description of the survey, its goals and the reasons for personal interviews. This introduction aims to mitigate any fears and allows the interview partners to get used to the interview situation. The interviewer can display knowledge and emotional intelligence at this point by stating that elicited defects will be used rather than judged for example. At the end of the introduction the way of documenting the interview results is agreed upon. There, a trade-off might be necessary between recording the interview results and manually documenting them; we experienced recordings to threaten the validity by potentially influencing the behavior of the participants while manual documentation might be prone to bias. In any case, the anonymity of the analysis is guaranteed before the interview.

Following the introduction is the description of the context by the interview partners. This includes the tasks, activities and processes they are involved in. This part of the interview is individual and helps the interviewer later in the classification of the discussed defects. Questions such as “What are your inputs and outputs?” help the interview partners to express their role, constraints, tasks and results toward the interviewer. The semi-structured approach of the interview helps the interviewer in this part as it allows for inquiries by the interviewers in case of unfamiliar terms and concepts. This part is not described on the questionnaire as interview partners are typically able to elaborate their work context. This also helps them to get into a flow of words as “at a basic level, people like to talk about themselves” [61].

The core part of the interview is the discussion of defects including their description, frequency and severity. Also the resulting failures and possible detection and/or prevention techniques are discussed. Again, this part is individual, but is guided by the questions on the interview preparation sheet. This guidance exploits the order in the heads of the interview partners as they likely prepared the questions in the order they were on the preparation sheet. At the end of the interview, an agreement of future contact has to be reached.

In general, it is the interviewer’s job to keep up an objective atmosphere and tone. It is hard for humans to admit defects and discuss them, but it is in fact the decisive point of the presented approach. Thus, the interviewer must cater to the interview partner using emotional intelligence. Additionally, “whatever the training and intentions of the interviewer, the social interaction of the qualitative interview may unfold in unexpected ways.” [61].

4.2.3 Analysis

The analysis of the interviews is used for the classifications of the collected defects. Defects interesting for the description and operationalization of defect models are common and recurring defects and defects with a high severity. To perform the classification and go from defects to defect classes, the first step of Grounded Theory [58] is employed. In that step, the recordings are coded in chronological order whereas the codes are iteratively abstracted to categories eventually leading to a basic defect taxonomy. Codes may also include contexts, roles and distinctions of the employed quality assurance process. This helps the interviewer to capture “what is happening in the data” [61]. After coding, all excerpts of the recordings are grouped by code and reheard to focus on one particular defect and its context, origination and consequences.

In the classification, the basic taxonomy of defects contains two basic families of defects: technical and process-related defects. Process-related defects concern all methodological, organizational and process aspects (as defined by the defect causal analysis [84]) and contain defects causing technical defects. Technical defects are directly attributable to the product and are detectable by measures of quality assurance. These two families of defects yield extension points. An exemplary extension could

be tool-related defects or defects rooted in the behavior of humans. These can be added dynamically and defects may belong to multiple classes depending on the context. Recall that, we deliberately chose to "tailor [the taxonomy] to our [...] specific needs"[28, 85] to stay flexible for seamless adaptation to specific contexts and domains w.r.t. the creation of defect models.

After the analysis, we created a report summarizing the results to the project partners and used it as basis for discussion in a concluding workshop. In this workshop, we presented the results and the discussion yielded a last validation of the results w.r.t. the expectations of the project partners. Afterwards, eligible defects for the creation and operationalization of defect models were discussed and selected constituting a last contact with the project partners to potentially initiate the development of tools based on the defect models.

4.3 Field Study Design

We conducted our field study by relying in total on four cases. In each case, we follow the same study design. In the following, we report on the design which we organize according to Runeson et al. [133]

4.3.1 Research Questions

The goal is to investigate the advantages and limitations in the elicitation and classification of defects for defect models using our DELICLA approach described in Section 4.2. To this end, we formulate three research questions.

RQ 1 (Suitability):

What (kind of) defects can be elicited with the approach; what is the degree of sensitivity to their context; and how comprehensive is the approach?

The core idea behind the approach is to elicit and classify common and recurring / severe defects independent of the context it is used in while preserving the context-dependent usefulness to adapt QA techniques to those defects. Hence, our first research question targets the adaptability of the approach to different employment contexts and its ability to always elicit and classify defects relevant to quality assurance independent of context. In particular, it should not be affected by changes of domains (information and cyber-physical), test / quality assurance levels (review, inspection, unit, integration and system test) and project partner. Finally, we rate a defect as context-independent if we find a relation to existing evidence in a given baseline. This means, if we find a study that indicates to the same defect in a different context, we may assume that the defect is context independent.

RQ 2 (Operationalizability):

Can the results of the approach be used for the description and operationalization of defect models?

The classification and elicitation of defects for defect models aims at their later description and operationalization. Thus, the results of the approach must yield a basis for decision-making to make the effort to describe and operationalize the respective defect models and yield starting points for their description and operationalization. This research question therefore aims at analyzing whether the basis of decision-making and starting points are retrievable by the approach, thereby manifesting a direct usefulness to project partners. We do not have a clear oracle to answer this question.To answer the research question, we will therefore point to indicators for successful description and operationalization of defect models based on our approach.

RQ 3 (Indirect short-term benefit):

Besides potential defect models and their operationalization, how valuable are our results to the project partners?

When our approach has been applied, project partners are given a final report to inform them about the results. This report contains all elicited and classified defects as well as possible proposals for action. In addition to the value for defect models. This research question targets the usefulness of the report in the eyes of those project partners considering the time invested on the project partner's side, thereby manifesting the indirect benefit of the approach. Again, we do not have an oracle. However, the quality of the results w.r.t. sufficiency and the cost-effectiveness can be rated by the project partners based on subjective expert judgement and feedback gathered during a concluding workshop.

4.3.2 Case and Subject Selection

We apply our process for the elicitation and classification of defect models to four software development projects of different industry project partners (more details in Section 4.4.1). We do not change our process throughout the field study to gain comparable results, although this affects internal validity. The four projects were chosen on an opportunistic basis. As we required real-world development projects and project managers / members to agree, the process was performed when possible. However, the chosen cases are suitable to answer our research questions if the selected projects are distributed across different companies working in different application domains.

4.3.3 Data Collection and Analysis Procedures

To collect and analyze the data, we use our DELICLA approach for the elicitation and classification of defects as described in Section 4.2.

To answer *RQ 1*, we list the top 14 defects (i.e. all defects mentioned in at least two interviews within the same context) we elicited and classified and evaluate their commonality in contrast with their context sensitivity. That is, for each defect, we analyze whether it is context-dependent or context-independent if we find a relation to existing evidence. As a baseline, we use the defects reported by Kalinowski et al. [85] and Leszak et al. [97].

We also quantify the number of defects elicited and give an assessment as to if the interviews allow for a comprehensive defect-based perspective on projects or organizations. There is no clear agreement on a sufficient number of interviews in general, but indicators toward sufficient numbers may be given [10]. For our cases, we agree on the sufficiency of the number of interviews (43) when we observe a saturation in the answers given, i.e. when no new defects arise. Saturation is taken as a sign of comprehensiveness.

To answer *RQ 2*, we list indicators of tools and methods created from defects classified and elicited with our approach. These tools and methods may not have a fully formulated formal defect model description, but are able to demonstrate whether (and how) results may be operationalized.

To answer *RQ 3*, we describe indicators of the quality of the results and the involved costs by gathering expert feedback from project partners after performing our approach. This feedback is a direct external grading of our approach by industry experts and yields an assessment of its cost-effectiveness.

4.4 Case Study Results

We performed the case study in four different industry projects (settings) with different industry partners. For reasons of non-disclosure agreements, we cannot give detailed information on project-specifics and the particularities of context-specific defects. However, we can state their domain, the number of interviews conducted and the classes of defects.

The top 14 defects independent of their setting are shown in Table 4.2. The settings and their respectively elicited and classified defects are shown in Figure 4.2. They are grouped by our basic taxonomy defined in Section 4.2 into technical (Figure 4.2a) and process-related defects (Figure 4.2b) and ordered each according to their context-sensitivity. Interestingly, we have found existing evidence for defects identified as context dependent as the existing evidence provided an extensive, and thereby, fitting defect description. Note that, defects without an ID in Section 4.2 and Figure 4.2a were not common and recurring and are not discussed further.

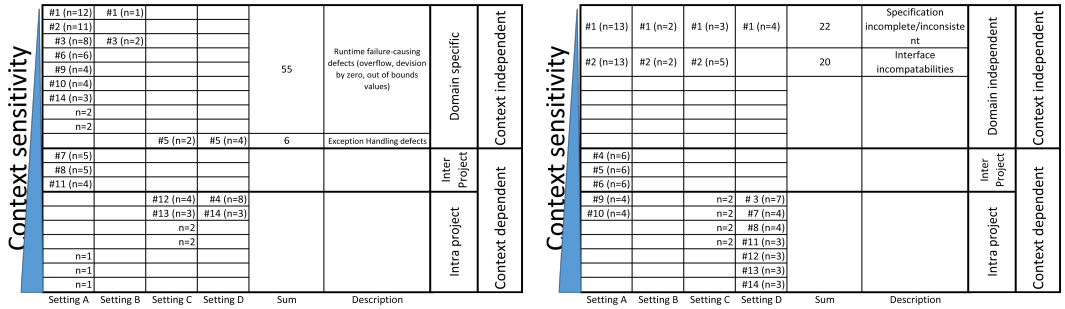
4.4.1 Case Description

Setting A is a medium size cyber-physical software supplier. 24 subjects were interviewed with the aim to draw a organization-wide picture of common and recurring

Table 4.2: Top 14 defects by frequency (at least mentioned twice in the same context (n>2) from 43 interviews)

	#	Name	Ex. Ev.	Description	Mentioned Consequences
Top 15 Technical Defects	1	Signal Range	[97]	Ranges of signals were not as described in the specification	Undefined / unspecified behavior of connected systems
	2	Scaling	[97]	Fixed-point values were scaled incorrectly for their specified range	Possible under-/overflows and/or system outputs differ from specification
	3	Wrong initial value	[97]	The initial values of the system were not set or set incorrectly	Initial system outputs differ from specification
	4	Data dependencies	[85] [97]	Data dependencies were unclear	When changing data formats, not all locations of the data formats were updated
	5	Exception Handling	[97]	Exception handling was either untested or not implemented as specified	Execution of exception handling routines lead to system failure
	6	Dead code due to safeguards	[97]	Dead superfluous safeguards were implemented	Degraded system performance and/or real-time requirements not met
	7	Linkage of Components	[97]	Interfaces of components were not connected as specified	System outputs differ from specification
	8	Variable re-use	[97]	Mandatory re-use of variables and developers assumed incorrect current values	System outputs differ from specification
	9	Different base	[97]	Calculations switched base (10 to 2 and vice versa)	System outputs differ from specification
	10	State chart defects	[97]	Defects related state charts	System outputs differ from specification
	11	Transposed characters	[97]	Characters in user interfaces and framework configurations were transposed	Mapping of code to user interface does not work
	12	Web browser incompatibilities		Web browsers had different interpretations of JavaScript and HTML	Browser-dependent rendering of web pages
	13	Validation of input		Inputs were either not or not validated according to specification	Ability to input arbitrary or malicious data
	14	Concurrency		Concurrency measures were not used as specified	Deadlocks and atomicity violations
Top 14 Process-related Defects	1	Specification incomplete/inconsistent	[85] [97]	The specification was either inconsistent, lacking information or inexistent	Thorough verification of implementation impossible and testing deferred
	2	Interface incompatibilities	[97]	Agreed interfaces of components were not designed as discussed / specified	Rework for interfaces required after deadline for implementation
	3	Missing domain knowledge	[85]	Engineers lacked the concrete domain knowledge to implement a requirement	Delivery delayed and project time exceeded
	4	Cloning		Engineers used cloning as a way to add functionality to systems	Cloned parts provide functionality not required by the system in development
	5	Static Analysis runtime	[97]	Static analysis of the implementation was started too late in the process	Delivery delayed and project time exceeded
	6	Quality assurance deemed unnecessary		Engineers did not see the necessity for quality assurance	Review / Testing not performed according to specified process
	7	Late involvement of users	[97]	Users were involved late or not at all in a SCRUM-based process	Requirements not according to user problem statement
	8	Misestimating of costs	[97]	Inability to estimate cost for requirements in a SCRUM-based process	Delivery delayed and project time exceeded
	9	Call order dependencies	[97]	Call orders were switched without informing engineers	Extra testing effort required with difficult fault localization
	10	Misunderstood instructions		New engineers did not understand given documentation	Delivery delayed and project time exceeded
	11	Distributed development	[97]	Development and test team were at different locations	Communication deficiencies yielded untested components with runtime failures
	12	Insufficient test environment	[97]	The test environment did not contain all components to be tested	Some defects could only be detected in production environment
	13	Development by single person		A single person was developing a large part of the system	Incomprehensible implementation of components
	14	Overloaded employees		Engineers were overwhelmed with the amount of work requested from them	Careless misateks due to stress

Figure 4.2: Defects with applicable ID (#, if number in top 14) and frequency (n) in the classification



(a) Technical defects **(b) Process-related defects**

(mentioned in at least 2 interviews) defects. These systems primarily targeted the automotive domain, but also were in the domain of aerospace, railway and medical. The predominant development process was the V-model.

Setting B is a department of a large German car manufacturer. 3 subjects were interviewed as to try out the approach and enable a first glance at a defect-based perspective in this department using the V-model as development process. Note that, this low number of interviews is discussed in threats to validity.

Setting C is a project of medium size in an information system developing company. 6 subjects were interviewed to give the company an introduction to the approach. The interviews were performed in a large scale website front end project developed using the SCRUM methodology.

Setting D was an information system project of a railway company. 10 subjects were interviewed to show process deficiencies and give a defect-based perspective on currently employed development and quality assurance measures. The project was a graphical rail monitoring application project developed using the SCRUM methodology.

4.4.2 Subject Description

As described in the subject selection, we chose our project partners and projects in an opportunistic manner. All interview participants had an engineering or computer science background and at least one year of experience. The author applied the approach in the case studies.

In setting A, the majority of participants had a background in mechanical or electrical engineering and developed system using Matlab Simulink with either automatic code generation or using Matlab Simulink models as specification for their manually implemented systems.

In setting B, the interview partners were developing and/or testing the functional software for an electronic control unit developed in C++ and integrated into AUTOSAR.

In setting C, the interview partners included a broad selection of roles including architects, developers, testers, test managers, and scrum masters.

In setting D, the interview partners were from several different teams defined in SCRUM to also gain a comprehensive view on synergy effects and defects missed by their managers.

RQ 1: Suitability.

In all studies performed, the results always yielded technical and process-related defects. The top 14 defects of each category are shown in Table 4.2. For each defect, we additionally show whether we could find a relation to existing evidence (see column 4 in. Table 4.2). Figure 4.2 further illustrates each defect (via its identifier provided in Table 4.2) in relation to its degree of sensitivity to the context. Remember from the introduction that "context" here refers to a specific company or business unit of a company.

In setting A, the interviews revealed 15 technical and 7 process-related defects. The technical defects were mainly run-time failures such as overflow due to the abstraction from the underlying computational model in Matlab Simulink. These failures were caused by wrong signal ranges of units, wrong scaling of fixed-point types and wrong initial values. The process-related defects were interface incompatibilities and incomplete / incorrect specifications. We performed 24 interviews in total. However, the top most common and recurring faults were named in 12, 11 and 8 interviews respectively. Since this was a cross-project company wide survey, the diversity of developers and testers interviewed introduced differences in the defects common and recurring in their respective fields. Baker and Edwards [10] hint at 12 interviews to be sufficient. In our setting, saturation was indeed achieved with even fewer interviews; the revealed common and recurring defects can be assumed to be comprehensive.

In setting B, the interviews revealed 3 technical and 1 process-related defect. The technical defects were related to initial values in C++ (2) and overflows (1). The process-related defect were due to interfaces (2) and incomplete specifications (2). Contrary to all other settings, this setting was only to give a first glance as described in the case description. Thus, comprehensiveness was intentionally neglected, but to provide a first glance 3 interviews were sufficient.

In setting C, the interviews revealed 5 technical defects and 6 process-related defects. The technical defects were related to web browser incompatibilities (4), validation of input data (3) and exception handling (2). The most prominent process-related defects were incomplete specification (5) and interface incompatibilities (3). With only 6 interviews, we did not perform a sufficient number of interviews per se. However, the project's size was only 10 persons and effects of saturation were quickly observable. This saturation yields an indication towards comprehensiveness, albeit inconclusive.

In setting D, the interviews revealed 3 technical defects and 7 process-related defects. The top technical defects were related to data dependencies (8), exception handling (4), concurrency (3). The prominent process-related defects were incomplete specifications (7), interface incompatibilities (4) and late involvement of the customers (4). Again, 10 interviews are below the sufficiency baseline, but the project's size was only 18 persons. Once again, saturation yields an indication towards comprehensiveness, albeit inconclusive.

Although the aim of the DELICLA approach was only to elicit and classify technical defects, process-related defects were mentioned by interview partners and were classified as well. Many interview partners stated process-related defects as causes for technical defects, yielding a causal relation between some defects. For instance, in the cyber-physical settings, the inconsistent / incomplete specification was described to lead to incorrect signal ranges and wrong initial values. In the information system domain, the format of user stories as use cases without exceptions was described to lead to untested / incorrect exception handling. We did not believe in advance these causalities or process-related defects to be important at first. However, we later realized their potential for deciding whether to (1) employ defect models for quality assurance to detect or (2) make organizational, methodological or process adjustments to prevent these defects. Many project partners were intrigued about the causalities and estimated the effort to change their processes lower than to employ defect models for some technical defect. In particular, since process improvement by using elicited defects has been described in literature [84, 85].

An interesting observation was the presence of domain independent defects of technical as well as process-related nature (see Figure 4.2). Domain independent technical defects were run time failure causing defects in embedded systems in setting A and B as well as untested exception handling defects in information systems in settings C and D. Moreover, interface incompatibilities and incomplete / inconsistent specifications/requirements were process-related defects present in all settings.

We therefore conclude so far that our approach is suitable to elicit a broad spectrum of defects which cover the particularities of the envisioned context as well as context-independent defects.

RQ 2: Operationalizability.

In all settings, we were able to derive possible solution proposals for handling each elicited and classified defect. These solution proposals do not necessarily include formal descriptions of defect models, but rather are indicators for operationalization possibilities. However, we or our project partners were able to design tools or methods that have an underlying defect model, and in some cases described next, we were able to operationalize the defects via tools.

Setting A resulted in a testing tool called 8CAGE in Chapter 5. 8Cage is a lightweight testing tool for Matlab Simulink systems based on defect models. The employed defect

models target overflow/underflow, division by zero run time failures as well as signal range and scaling problems.

Setting B yielded an internal testing tool for the testing of the interfaces of the software to the AUTOSAR Runtime Environment (RTE) developed by the project partner. Due to frequent changes in the communication between each electronic control unit, the RTE had to be recreated for each change. Sometimes changes were not implemented leading to unusual failure message and large efforts spent in fault localizations. The internal testing tool can now be run to show these unimplemented changes automatically.

Settings C and D did not result in any tools as of now. However, they yielded requirements to specifically test exception handling functionality in Java systems. The task of the tool is to explicitly throw exceptions at applicable points in the code as to deliberately test the developed exception handling. We currently have collected these requirements and tool development is imminent. In addition, quality standards for SCRUM user story standards as a method of early defect detection have been proposed and partially implemented in setting C and B. These methods include perspective-based reading [138] of user stories before accepting them and explicit definitions of acceptance criteria including a specification for exception handling.

Overall, the tools and methods developed enable front-loading of quality assurance activities. This allows developers and testers to focus on common and recurring defects in specification and implementation and either makes them aware of the defects or allows the (semi-)automatic detection. Thus, the defect-based perspective may be able to increase the potential to avoid these defects in the future.

We therefore conclude so far that we could elicit defects suitable for operationalization in the chosen context.

RQ 3: Indirect short-term benefit.

After presenting the results in the workshop meeting of DELICLA, we asked the responsibles to assess the usefulness of our approach in terms of (1) being aware of the elicited and classified defects, (2) future actions based on the report, and (3) cost-effectiveness.

In setting A, the responsibles deemed the results satisfying. The defects were mostly known to them, but they were content to have written results in hand for justification towards their management. Using the results, they could convince their management and customers to invest into consulting regarding specific defects. The efficiency of only one hour per interview while leading to sufficiently comprehensive results was perceived positively. They agreed to perform further interviews in the future. However, they remarked that our approach did not reveal many defects previously unknown to them, but were now able to gain an essential understanding of their frequency. They also commented on the difficulty to select distinct projects for the proposed measures in this inter-project setting.

In setting B, the project responsibles only gave us a limited feedback. They stated all defects to be known and saw the advantage in now having a thorough documentation. We did not create defect models or develop operationalizations for them after applying the approach. However, they developed a tool without our involvement based on one reported defect.

In setting C, the project responsibles were surprised how non-intruding and conciliatory our approach is and how professionally it can be handled. They were aware of most of the defects, but not that 20% of their test cases were already defect-based. The project was already in a late stage when we applied the approach and future actions could not be taken due to the time left. They also perceived the efficiency of one hour per interview partner as positive and described the comprehensiveness of results as given. When discussing further interviews, they questioned the application towards a whole organization as measure to find organization-wide defects with a small number of interviews.

In setting D, the project responsibles were aware of most defects elicited and classified and satisfied with the application of the approach in general. They said the approach "yields good results with little effort" and it provides "a view" on the defects in the project from "a different side". In addition, they stated that "nothing is missing [from the results] and [results are] diagnostically conclusive". Concerning the possible solution approaches presented, they "may not be the way to go", but "give a first idea for discussion in project meetings". Again, further interviews were discussed, but the time required to interview the complete project with more than 50 employees was deemed to much. The project partner rather wanted to use other techniques such as observation or focus groups to minimize time required on their side. However, qualitative interviews were deemed "a good starting point".

4.4.3 Threats to Validity

There is a plethora of threats to the validity, let alone those inherent to case study research. To start with, the qualitative nature of the approach as well as the qualitative nature of the evaluation technique rely to a large extent on subjective expert judgment. First and foremost, the approach was applied by the same person evaluating it. The *internal validity* is particularly threatened by the subjectivity in the interviews and especially in their interpretation. Coding used to classify the defects, for example, is an inherently creative task. However, our aim was to explore the potential of such qualitative methods, to reveal subjective opinions by the study participants, and to elaborate – despite the inherent threats – the suitability of the chosen approach.

The *construct validity* is threatened in two ways. First, the research questions were answered via qualitative methods only and we cannot guarantee that we could fully answer the questions based on our data. We compensated the threat, especially for research question 1, by taking an external baseline as an orientation. Second, we cannot guarantee that we have a sufficient number of interviews to reliably decide on

the completeness of the data to elaborate comprehensive defects. We compensated this threat by applying the principles of Grounded Theory where we explicitly considered a saturation of the answers if no new codes arose. Also, we believe the number of interviews to be less important than the coverage of roles within (different) teams and superordinate roles. This is hinted at by setting B, C and D in particular.

Finally, the *external validity* is threatened by the nature of the approach and its evaluation. We do not claim our findings to generalize outside the field study. Using a different person as interviewer or performing the interviews in different contexts (organization, domain, location, etc.) may yield very different results. Our intention was, however, not to generalize from our findings but to evaluate the extent to which our approach is suitable to cover the particularities of contexts whereby the results hold specifically for those contexts. Yet, by comparing the results with an external baseline, we could determine context-independent defects which potential for generalization.

4.5 Lessons Learned

With each application of our approach, we could gather inputs which we summarize next. We structure the lessons into lessons we made with the qualitative methods in application, and lessons regarding the outcome.

4.5.1 Applying DELICLA

Interestingly, our experience with the few unprepared interview partners has shown that reflections about defects during the stress situation of the interview are mostly impossible. A key factor to the professional attitude was the calming of interview partners and the establishment of a trust relationship at the beginning of the interview. Many interview partners entered the interview room anxious and seemed relieved after the initial address was given. In particular, the phrase that “defects will be utilized instead of judged” helped overcome fears. In addition, a first name basis and the display of integrity of the interviewer as well as a non-facing position of the interviewer also helped calm interview partners.

Concerning the way the interviews are performed, we learned to listen more closely. In setting A, we asked many interposed questions and might have introduced confirmation bias.

Concerning the recording, we applied our approach in settings C and D with recordings and written documentation in parallel. We discovered written documentation of interviews to be biased at the beginning and the end of a setting. At the beginning, the writer did not know what to write down and documented irrelevant information. After some interviews, the writer fell into confirmation bias and always documented defects previously stated as common and recurring, but not necessarily common and recurring in the current interview. Thus, we advise to record interviews to perform the classification of defects separately to their elicitation. In addition, we learned that the

recording device should not be in the line of sight of the interview partner as it might prevent them from talking freely.

A key aspect of the interviewer is the adherence to integrity and the honesty towards the anonymization of data. In addition, the interviewer should handle any defects objectively by staying serious and focused even though defects may induce humor in the conversation.

4.5.2 DELICLA Outcome

Although we were not looking for specific process-related or inter-setting defects in advance (see Figure 4.2), we found two process-related defects to be present in all settings. Not too surprisingly, these defects relate to (1) requirements and (2) architecture. Thus, these defects did not emerge during the development, but rather during planning phases.

4.6 Conclusion

This chapter has introduced and evaluated the DELICLA approach to elicit and classify defects to enable the strategic decision making for their future description and operationalization as defect models. DELICLA is entirely based on existing elicitation and analysis approaches [61]. Our approach uses a qualitative explorative method with personal interviews as elicitation and grounded theory as classification technique.

We have evaluated the approach in a field study with four different companies. The chosen settings varied in their domains. Using our approach, we were able to elicit and classify defects having both a context-dependent and -independent relevance while providing indicators to the extent to which they relate to existing evidence. We deem these defects common and recurring w.r.t. their context and test cases for their (semi-)automatic detection within these respective contexts fulfill the definition of good test cases. The approach was applicable in different contexts/domains due to the employment of qualitative approaches. We could use the elaborated defects to derive requirements for their (semi-) automatic detection by tools and create possible solution proposals in all settings. The feedback given to us by project partner executives was positive yielding “informative results with little effort”. Thus, the results strengthen our confidence that the studies are representative and the approach is suitable to elicit context-specific defects without being too specific for a context. However, we are lacking a study with negative results.

Using DELICLA, we have gained an insight of existing defects in different contexts and domains. For some technical defects, we present their description and operationalization in respective defect models in chapters 5, 6 and 7. These operationalizations are a direct result of the field study. DELICLA elicits and classifies more defects than are sensible to be described and operationalized. However, these defects yield a valuable addition to the defect repository. This includes the process-related defects,

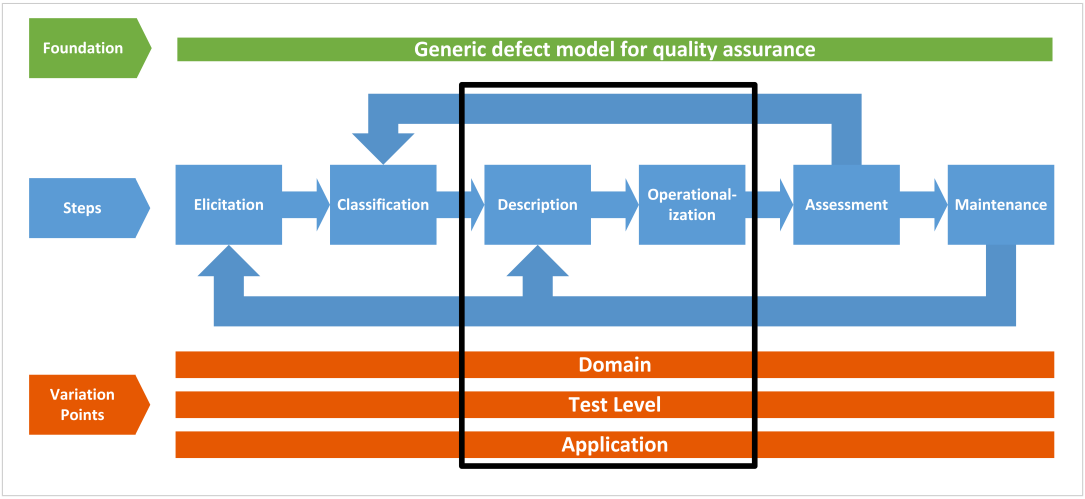
which can be addressed by possibly employing constructive quality assurance (see Section 2.1 and Section 8.2). Moreover, for some defects we were able to find cause effect relationships to other defects. This yields the question about the ability of the method to extensively find cause effect relationships for all defects. To answer this question, further work and maybe a higher level portrayal of defects as provided by the methodology of Kalinowski et al. [85] is required.

5

Description and Operationalization: 8Cage for unit testing

In the last chapters, the foundations for a systematic and (semi-)automatic approach to defect-based quality assurance using defect models were defined and common and recurring defects have been elicited and classified. Based on the results of DELICLA in the embedded system's domain, we developed the operationalization 8Cage to (semi-)automatically detect elicited and classified defects #1, #2, #6 and #10 in the top 14 technical defects of Table 4.2. 8Cage is the first of three explicit descriptions and operationalizations of defect models in the domain of Matlab Simulink systems, which constitutes the method application step in the defect model lifecycle framework (see Figure 5.1). The descriptions and operationalizations of 8Cage reside on the unit testing level and are applicable to all Matlab Simulink systems. Thus, the variation points are a domain of Matlab Simulink, the unit testing level and application (as well as specification) independence.

Figure 5.1: Position of the operationalization 8Cage in this thesis



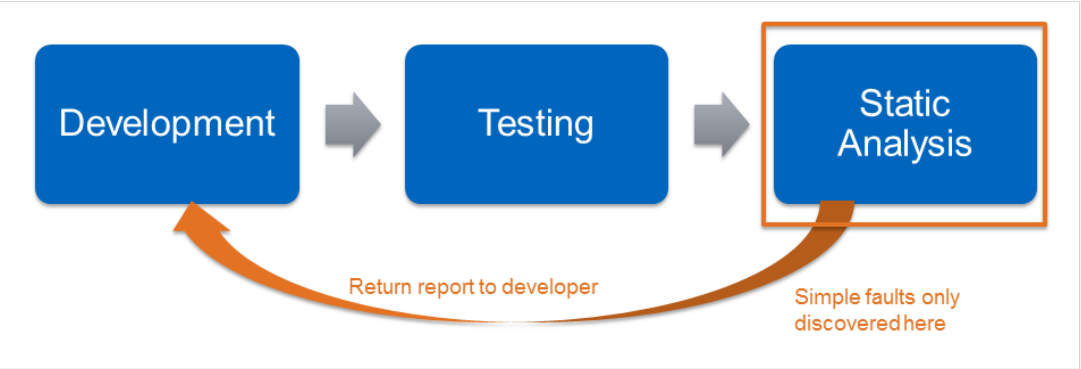
Recalling the considerations of Matlab Simulink systems in Section 2.3, its data flow-driven, block-based notation yields great understandability for engineers. However, it hides—and is supposed to do so—the underlying computational model of processor and memory architecture. This regularly leads to problems with generated code. Certain faults are invisible or hard to detect by review at the model level. Such faults include run time detectable under-/overflows, divisions by zero, comparisons and rounding, state chart faults and possibly infinite loops. In addition, it may be hard for engineers developing a system to assess if their assertions hold.

Further typical faults include exceeded ranges of I/O signals. These ranges are specified by a developer or imposed by physics. For instance, the revolutions per minute (rpm) of a gasoline combustion engine may range from 0 to 9,500. Thus, a 16-bit unsigned integer of range 0 to 65,535 is sufficient to hold the rpm signal's range. However, the rpm signal will never be larger than 9,500 when used/given as I/O. There may be assumptions in certain units that rely on this range. Thus, the rpm signal's range must be rigorously checked to never exceed 9,500. Typically, the ranges are also checked by static analysis late in the development process. Detecting failures caused by invalid/unspecified ranges on a unit level would yield early detection benefits as well.

Unfortunately, many of these "careless" / "simple" faults involving only a single or a simple composition of blocks are detected only by static analysis at the end of a development cycle (see focus in Figure 5.2). Static analysis of a real-world Matlab Simulink system using abstract interpretation (see Section 2.1.3) requires approximately two days of execution time and three days of manual analysis to reveal the faults. In case these faults are careless faults, they taint the analysis. When the careless fault is triggered, the resulting behavior may be undefined leading to a rippling effect of failures. This creates an overhead of at least one development cycle as a further analysis has to be performed after the faults are given to the developers and corrected (see backward arrow in Figure 5.2). Thus, earlier detection would yield cost benefits by reducing the required analysis effort. In addition, developer assumptions about certain blocks or block combinations may be verified only as late as Hardware-In-the-Loop (HIL) tests. HIL tests require the software to be deployed on the target platform and are performed even later than static analysis.

To address this overhead and enable early detection of careless faults, we developed the operationalization 8Cage. 8Cage is an automated test case generator on the unit testing level for Matlab Simulink models based on defect models (see Chapter 3). The fault models currently operationalized in 8Cage relate to over-/underflows, divisions by zero, comparisons/rounding, Matlab Stateflow (state chart implementation in Matlab Simulink) faults and loss of precision. The failure models relate to exceeding of I/O ranges. Although the defect models operationalized by 8Cage involve only a single or a simple composition of blocks, the defects they describe were identified as common and recurring in the field study in Section 4.3. In addition, developer assumptions concerning a single block can be operationalized as developers/testers can specify

Figure 5.2: Common testing process of Matlab Simulink Systems



their own fault hypotheses and have them automatically tested. The description is enabled by a graphical user interface in Matlab Simulink. 8Cage uses a five step process to detect the faults/failures and report them to the developers consisting of configuration (1), smell detection (2), test case generation (3), test case execution (4) and report generation (5). Since the fault models directly relate to single or a simple composition of blocks, the fault localization is also automatic. For the failure models, fault localization requires manual effort. Because 8Cage only uses the implicit specification (possibly including the I/O ranges) of what should not happen, it can detect failures in a model even without a specification. If a possibly failure-causing block or combination of blocks is present, 8Cage aims to produce a test case constituting a counter example. Thereby, it eliminates the need for expert analysis with static verification tools for the described faults/failures and bridges the model/source-code level. We evaluate 8Cage concerning the reproducibility, effectiveness and efficiency of the test case generation step using 29 representative extracted units from three real-world Matlab Simulink systems.

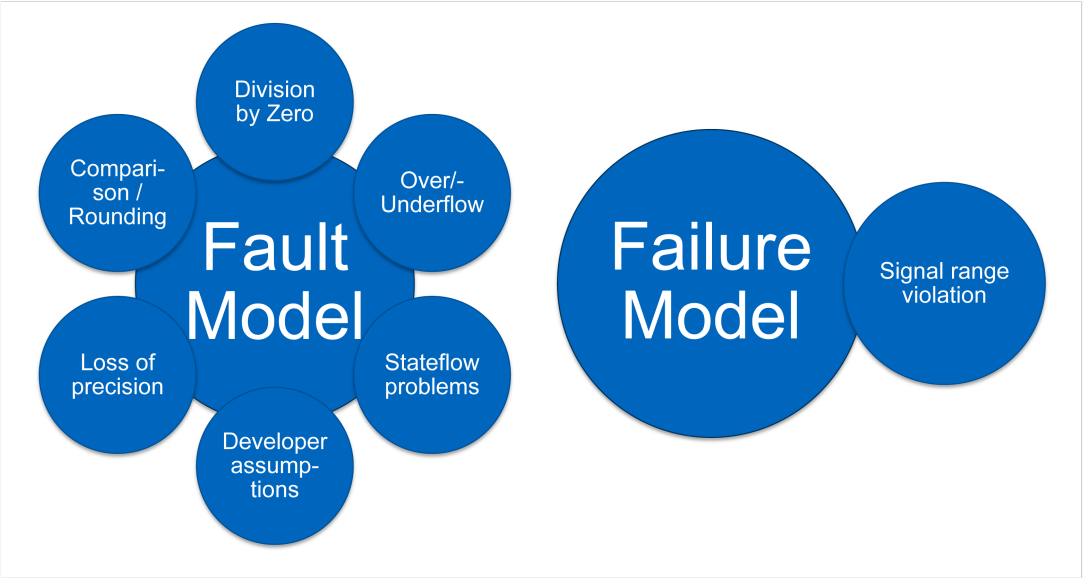
5.1 Fault and Failure Models

The defects for the defects models operationalized in 8Cage were elicited in the field study of common and recurring defects for the creation of defect models (see Section 4.3). The field study elicited and classified relevant common and recurring faults describable in fault models (left in Figure 5.3) and common and recurring failures describable in failure models (right in Figure 5.3). In the following, all defect models are described and mapped to the generic defect model (see Chapter 3). Based on these descriptions, the operationalizations well be created (see Section 5.2).

5.1.1 Fault Models

The fault models operationalized in 8Cage relate to over-/underflows, divisions by zero, comparisons/rounding, Matlab Stateflow (state chart implementation in Matlab Simulink) faults and loss of precision. In addition, custom defect models for developer

Figure 5.3: Defect Models of 8Cage



assumptions are possible. Recall that, a fault model consists of a transformation α and a characterization of the failure domain φ including their approximations $\tilde{\alpha}$ and $\tilde{\varphi}$ as described in Chapter 3. α transforms a correct behavior description (e.g. program) into an incorrect behavior description by injecting faults. Since α injects faults, the idea of a fault model is to use the faults injected by α to create the failure domain φ consisting of all inputs leading to paths through the program impacted by α. For the fault models in 8Cage, all captured faults present a pattern in the BD. This pattern is independent of the co-domain of α and requires only the image of the transformation α to be described. More precisely, measures to prevent certain faults are removed by α and the exact nature of these measures is negligible. Upon finding such a pattern, 8Cage assumes α has been applied and aims to provide evidence for an actual fault to be present by creating a test case leading to a failure. As many of the described faults lead to run time failures, the default oracle of all fault models in 8Cage is a robustness oracle w.r.t. any error messages produced by Matlab Simulink when executing the Model in the Loop (MIL) (see Section 2.3) simulation. In case Matlab produces an error or crashes, the test case verdict is fail. Otherwise it is pass. In the following descriptions of fault models in 8Cage, the image of α always expresses a fault pattern (syntactic or static prerequisites). The characterization of the failure domain φ for the respective fault pattern requires certain inputs/outputs (semantic or dynamic prerequisites) to be the inputs/outputs of the fault pattern. Thus, the pattern yields a potential fault and evidence for an actual fault to be present is provided in form of a test case. The operationalization is according to the second operationalization scenario in Section 3.3.

Division by zero

A division by zero is a classic run time failure (see Section 3.2.2) leading to undefined behavior or program crashes depending on the employed computational architecture. The run time failure occurs whenever an integer division operation is performed using the value 0 as divisor. In Matlab Simulink, the divide block can cause such a run time failure, iff given 0 as divisor. In all Matlab versions up to 2012a, the guard for division by zero has to be created by the developer. Since version 2012a, the guard for division by zero is activated by default, but can be deactivated by the developer using optimization commands. Nevertheless, 8Cage provides help searching for a way to cause a division by zero, showing the requirement of the guard. In case the guard is activated, 8Cage can deactivate it and then search for a division by zero. Thus, 8Cage is be able to show the necessity to enable the guard.

The α for the division by zero fault model is any transformation deactivating the guard or removing one or more protection mechanisms for the 0 value to reach the divisor input port of the divide block. Although this is a plethora of transformations, we focus on the produced image / pattern as it is always a divide block with a disabled guard and an integer data type as input as the divisor. This pattern requires a test case causing the input of the divisor port of the divide block to be 0 to go from a potential fault to an actual fault with evidence. The test case is selected from the failure domain φ, which contains all values leading to 0 value as divisor input to the block. Formally, let

$$div_{no-guard} : integer \times integer \rightarrow integer \text{ with } div_{no-guard}(p_1, p_2) = p_1/p_2$$

be the formalization of the integer division block in Matlab Simulink. Further, let $f(i, b, p)$ be the function applied to the input signal i of the Matlab Simulink model up to the point of a port p of block b. Then the failure domain φ is characterized by the set $\{i : f(i, div_{no-guard}, p_2) == 0\}$ describing all inputs leading to 0 as input to the divisor port p_2 of the integer division block $div_{no-guard}$. Upon finding a $div_{no-guard}$ in a Matlab Simulink model, 8Cage assumes the transformation $\tilde{\varphi}$ is created by using symbolic execution trying to find f.

Like the division by zero, some blocks in Matlab Simulink have a limited space of inputs (#1 in the top 14 technical defects of Table 4.2). As an example, one such block is the square root with integer inputs. Given a negative value, the approximation of the square root enters an infinite loop. Thus, a further fault model in this category is the negative square root fault model, which can be defined analogously to the division by zero above.

Over-/underflow

Over-/underflows are typical run time failures of Matlab Simulink models (#1, #2 and #6 in the top 14 technical defects of Table 4.2). All Simulink blocks that can

cause over-/underflows have a binary property called “saturate on integer overflow” (SIO), which can be set by the developer. Enabling this property generates an over-/underflow-preventing safety check for the respective block. Enabling this property for every possible block would avert the problem of over-/underflows completely. However, each check requires processing time and may lead to dead code. Standards such as MISRA-C [114] require avoidance of dead code and processing time on embedded hardware may be costly. Thus, developers must carefully decide whether to enable the SIO property. 8Cage provides helps by searching for a way to cause an over-/underflow, showing the need to activate the SIO property. Since any of the blocks causing an over-/underflow can be described in a fault model and the schema is repetitive, we only describe two fault models concerning over-/underflows in Matlab Simulink models in this section. Further fault models can be defined analogously.

As an example, one over-/underflow fault model concerns the built-in Abs block of Simulink calculates the mathematical absolute value function of the input x (i.e. $|x|$). The code generated from this block is: `if (x < 0) return -x; else return x;`. This implementation of the absolute value function is efficient and works for all integer inputs except the most minimal one. Let x be a signed 8-bit integer. x then has a range from -128 to 127 making -128 the only integer without a positive counter-part. Due to -128 being stored in two’s complement (`1000 0000b`) and negation of it (flipping all bits and adding 1) will yield -128 again. A developer assumption for using the absolute value could be the non-negativity of the output, which cannot be guaranteed for all integers.

The fault pattern for the absolute value overflow fault model is again independent of the transformation α. α deactivates the SIO property or removes any other protection mechanism for the lowest integer to reach the single input port of the absolute value block. The fault pattern (in the image of α) is an absolute value block with a disabled SIO property abs_{no-SIO} and an integer data type as input the single port p_1. Thus, abs_{no-SIO} is defined as

$$abs_{no-SIO} : integer \rightarrow integer \text{ with } abs_{no-SIO}(p_1) = \begin{cases} p_1 & \text{if } 0 \leq p_1 \\ -(p_1) & \text{otherwise} \end{cases}$$

. φ then is the set $\{i : f(i, abs_{no-SIO}, p_1) == MIN\}$, where MIN is the lowest possible integer value of the input to p_1 of abs_{no-SIO}. $\tilde{\varphi}$ can be created by using symbolic execution as described in division by zero.

A second over-/underflows fault model considers a division overflow by a small fixed-point number. Many electronic control units (ECUs) do not contain a floating point unit due to cost reasons. Thus, fixed-point numbers are chosen to represent decimals. In Matlab, fixed-point numbers use integer data types and fix the precision (i.e. the number of digits after the decimal point). An 8 bit unsigned integer x with 1 bit of precision yields a range from 0 to 127.5 where the bit of precision can express .0 and .5 after the decimal point [109]. Computing $64/x$, where x can assume any value

in its range, can lead to an overflow if $x = 0.5$. This constitutes a multiplication by 2 with a desired result of 128 and an actual result of 0.

The fault pattern is the same as in the division by zero fault model in the last section with the slight modification of a fixed-point integer input to p_2. φ then is the set $\{i : 0 < f(i, div_{no-SIO}, p_2) << 1\}$ leading to a multiplication causing an overflow. $\tilde{\varphi}$ can be created by using symbolic execution and aims to pass the smallest possible number of the fixed-point data type MIN to p_2 of div_{no-SIO}. Further over-/underflows fault models can be described analogously.

Comparisons and rounding

A fault initially treated in foundation level programming courses is the direct comparison of floating point numbers (#1 and #2 in the top 14 technical defects of Table 4.2). Due to the scaled precision of floating point values, the comparison `1.1 - 0.1 == 1.0` will not always be true and an ϵ must be used. Matlab Simulink does not show the signal data types by default, which typically causes engineers to create the fault of a direct comparison.

The fault pattern for the comparison fault model is a direct comparison of floating point values in a comparison block and again independent of the transformation α. Let

$$comp : float \times float \rightarrow boolean \text{ with } comp(p_1, p_2) = p_1 == p_2$$

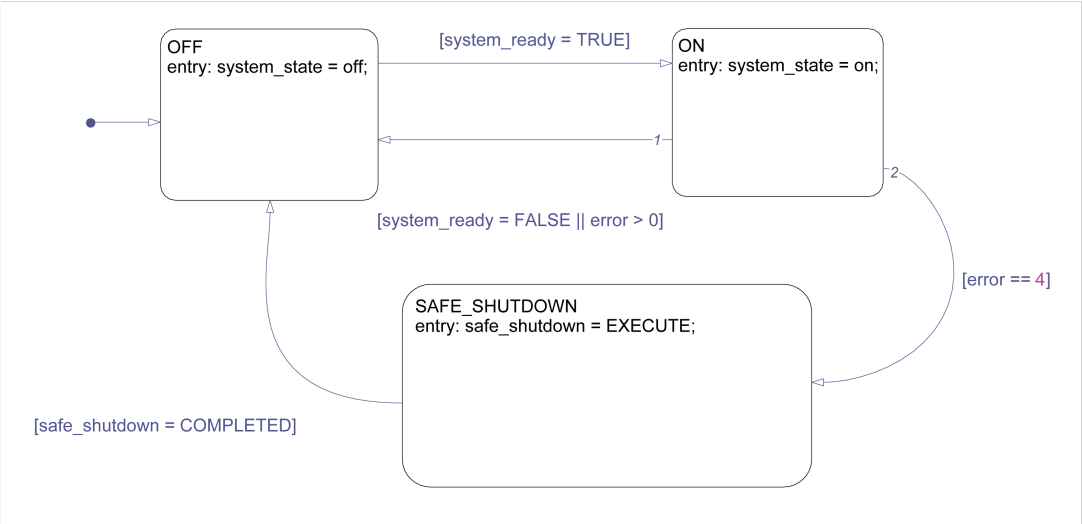
be a comparison block with equality according to the semantic of the C programming language and two floating point data types connected to the input ports p_1 and p_2. φ then produces the set $\{i : \neg(f(i, comp, p_1) == f(i, comp, p_2)) \wedge (abs(f(i, comp, p_1) - f(i, comp, p_2)) < \epsilon)\}$ containing all inputs falsely leading to negative comparison without taking ϵ into account. $\tilde{\varphi}$ then can be created by using symbolic execution.

The rounding fault model considers the integral block. This block sums up its inputs in an internal state and has a property called rounding mode. Rounding mode determines how the single input to the integral block is rounded. If the rounding mode is floor and the input given to the integral block is very small, the rounding can cause the addition of a 0 value leading to no state change of the integral. The fault pattern (in the image of α) is a integral block with a fixed point integer input and φ contains all inputs rounded to 0 when reaching the integral block. $\tilde{\varphi}$ then can be created by using symbolic execution. Rounding is performed in many Matlab Simulink blocks and we will use the example above as representative fault models. All further fault models concerning rounding in Matlab Simulink models can be defined analogously.

Stateflow faults

A Matlab Stateflow diagram is a special Matlab Simulink block, which allows the implementation of control logic by using state charts / machines in a graphical modeling notation. Engineers typically make a number of mistakes when using Stateflow

Figure 5.4: Matlab Stateflow diagram with unreachable state



diagrams (#10 in the top 14 technical defects of Table 4.2). One fault is the usage of $<$ or $>$ for enumerable variables in transition conditions. In Figure 5.4 `error` is an enumerable variable. The specification was to go back to off in case `error` is not 0, but either 1, 2 or 3. The specification was then changed to include a value of `error` of 4. If this value is received, the system is supposed to perform a safe shutdown. However, the engineers implemented a dead state `SAFE_SHUTDOWN` in Figure 5.4 with a lower transition priority than the already implemented transition to `OFF`. In the deterministic semantics of Matlab Stateflow, the transition condition with the lower transition priority will always be evaluated first leading to a positive transition condition evaluation for the transition to `OFF` before the transition to `SAFE_SHUTDOWN` is considered. Note that, the transition condition on the transition to `OFF` subsumes the transition condition on the transition to `SAFE_SHUTDOWN`.

In general, this fault can be translated to an unreachable state fault. The fault pattern (in the image of α) is a Stateflow diagram, which contains one or multiple potentially unreachable states. These state were either injected or made unreachable by α. φ then contains all traces supposed to lead to the unreachable state(s) in the Stateflow diagram. $\tilde{\varphi}$ can be created by using symbolic execution with the aim to cover all state in the Stateflow diagram. A manual inspection of the coverage then reveals the unreachable state(s).

A fault engineers commonly make in Stateflow diagrams is re-using the notation of the specification. It may be specified, that to transition to a certain state the condition $0 < x < 10$ of input x must be satisfied. Directly using this expression as transition condition yields the operator precedence of $(0 < x) < 10$, where $(0 < x)$ will produce either 1 or 0 value. This value will then be compared to < 10. Thus, $0 < x < 10$ is true for any value larger than 0. The correct implementation is $0 < x \&\& x < 10$.

The α for the range in Stateflow fault model is any transformation, which has a Stateflow diagram containing two direct comparisons in a transition condition as image. φ then contains all traces using the transition. This set is approximated by $\tilde{\varphi}$ by using symbolic execution.

Note that, over-/underflows of counters are also a problem in Stateflow diagrams and can be described as elaborated above.

Loss of precision

A loss of precision generally occurs when using fixed-point data types with too little precision for the operations or the wrong order of operations (#1 and #2 in the top 14 technical defects of Table 4.2). One fault model concerning the loss of precision is the division prior to multiplication fault model. A division prior to a multiplication causes a loss of precision, if the same result could also be achieved by performing the division after the multiplication. Of course, over-underflows may hinder such a switch of operation. As a simplified example, let's assume that both the division and multiplication are by multiples of 2. Then the division constitutes a right shift, while the multiplication is a left shift. When dividing before multiplication, the least significant bits are shifted out and by the subsequent multiplication zeros bits are shifted in. Thus, the least significant bits are zeroed and some precision of the fixed-point value is lost.

The fault pattern for the division prior to multiplication fault models is a division block before a multiplication block and independent of α. α switches the multiplication and division blocks such that they constitute the fault pattern. φ then contains all inputs passing through the division block before passing through the multiplication block. $\tilde{\varphi}$ then can be created by using symbolic execution and aims to find very precise inputs (i.e. least significant bit(s) is / are 1) to demonstrate the loss of precision.

Custom fault models

8Cage allows to specify single block fault models by using the Simulink syntax and a small run time extension. In the syntax of Matlab Simulink, each Simulink block has properties such as its name, type and inputs/output with a specified data type. The properties and data types on the input/output ports are static (i.e. they are a property directly visible from the model) and defined as the static properties (i.e. the fault pattern in the image of α) required by 8Cage. In addition, 8Cage requires knowledge about failure-causing input and/or output values at run time for each described block (i.e. elements of the resulting set of φ). These values do not necessarily need to be specified as absolute values (i.e. -128), but can also be specified as relative values (e.g. $MinValue$). The absolute value fault model above can be specified by telling 8Cage to locate blocks of type "Abs." Its properties should be "Saturate on integer overflow" set to value "off." The input value should have an integer type and the value should be $MinValue$.

Developers/Testers can specify these fault models using a graphical user interface in Matlab as users are familiarized. Thus, the specification of developer assumptions on single blocks is possible.

5.1.2 Failure Models

The failure models operationalized in 8Cage relate to signal range violations. All of the failure models require only a φ as only the failure(s), but not the underlying fault(s) are known.

Signal range violation

The signal ranges for each input and output of a Matlab Simulink model are typically described in the specification (#1 and #2 in the top 14 technical defects of Table 4.2). The input signal ranges constrain the set of inputs produced by $\tilde{\varphi}$. The output signal of a Matlab Simulink model must be checked to be within the specified output ranges for any given element of the constrained input set as input. Thus, the signal range violation failure model φ produces a set containing all inputs violating the output signal ranges. Formally, this set is $\{i : ([\![m]\!](i) < O_L \vee [\![m]\!](i) > O_H) \wedge I_L \leq i \leq I_H\}$ with m being the Matlab Simulink model and I_L and I_H as well as O_L and O_H being the lower and upper constraints signals for inputs and outputs respectively. $\tilde{\varphi}$ is created by using symbolic execution and contains one input signal to break the output signal range for each output signal (if possible to create).

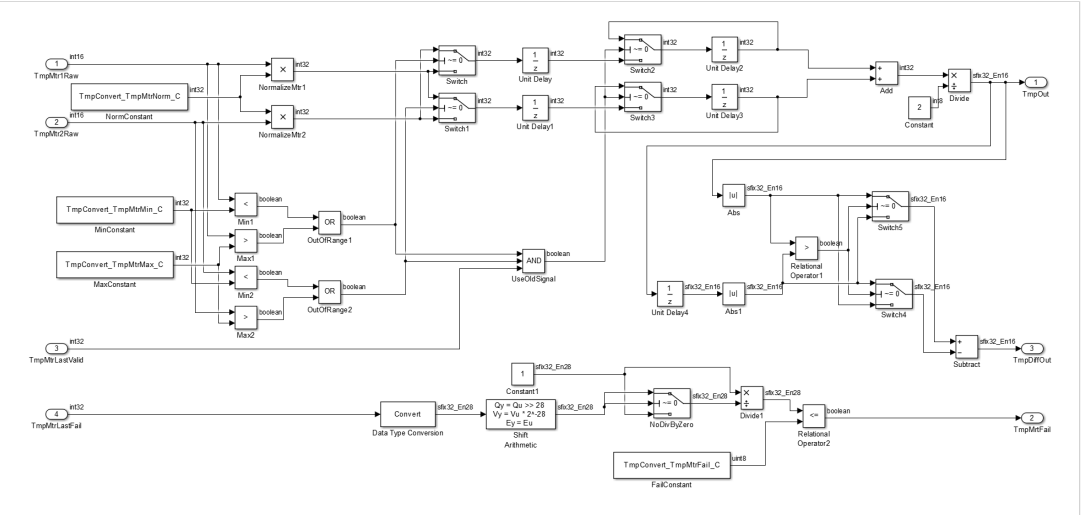
5.1.3 Summary

All of the faults and failures described in defect models above, are typically detected late using static analysis. Automatically detecting them as early as nightly builds yields time and cost benefits. The same holds true for other elementary faults such as dividing by zero or other single block related faults. This constitutes expert relief by operationalizing the defect models above. Note that, 8Cage is not limited to only these fault models, but has been and can further be extended to any failure/assumption caused/violated by a single Simulink block. We also only present defect models based on defects elicited and classified in a field study (see Section 4.3) and cannot ascertain comprehensiveness.

5.2 Operationalization

8Cage operationalizes the defect models above at the unit level of Matlab Simulink. 8Cage uses five steps for the operationalization of each defect model: configuration (Section 5.2.1), smell detection (Section 5.2.2), test case generation (Section 5.2.3), test case execution (Section 5.2.4) and report generation (Section 5.2.5). The purpose

Figure 5.5: An exemplary typical Matlab Simulink unit



of these steps is to find model smells and to generate evidence that these smells are actual faults.

The exemplary Matlab Simulink unit shown in Figure 5.5 is used as running example in each of the steps. The exemplary unit was created / synthesized by the author of this thesis based on the inspection of 60 Matlab Simulink units in an industrial electric engine control system. It has never been used in practical scenarios and serves the pure purpose of demonstration. However, it was deemed realistic by audiences of several presentations and conferences including ASE 2014 [72].

The unit represents the input processing of a temperature sensor. The sensor contains two physical sensors due to fault tolerance considerations. The input signal `TmpMtr1Raw` is the raw output of the first sensor, while `TmpMtr2Raw` is the raw output of the second. `TmpMtrLastValid` is a diagnostic input signal from the sensor acting as a dead man's switch. In case the sensor encounters any problem, the `TmpMtrLastValid` will yield a zero value. As soon as `TmpMtrLastValid` is 0, the `TmpMtrLastFail` input signal contains the fault information of the encountered problem.

The values of `TmpMtr1Raw` and `TmpMtr2Raw` have to be normalized before processing occurs. The normalization constant is `TmpConvert_TmpMtrNorm_C` and multiplied with the respective values. At the same time, a plausibility check for the temperature values is performed to see whether they are within the specified range of `TmpConvert_TmpMtrMin_C` and `TmpConvert_TmpMtrMax_C`. In case one of the temperature values is out of range, the in range temperature value overrides the out of range value. In case all values are in range and `TmpMtrLastValid` is not 0, the value is passed on to the calculation of the average of both temperature values, which is the output `TmpOut`. If the temperature values are no in range or `TmpMtrLastValid` is 0, the stored last valid values (Unit Delay is a storage block) of the temperature are used. The output signal `TmpDiffOut` is the temperature difference from the last calculation of

temperatures to the current temperature. The input signal `TmpMtrLastFail` is a bit mask, which is converted to the boolean output signal `TmpMtrFail`, which signals a faulty temperature sensor. The conversion is performed by performing a bit shift, a division and a comparison to the predefined constant `TmpConvert_TmpMtrFail_C`.

5.2.1 Configuration

The first step in the operationalization of any defect model using 8Cage is the configuration. The configuration parameters of 8Cage include the model file, its respective configuration files and the unit to test. In addition, the number of steps to simulate the model and the defect models to be used need to be provided. The configuration can be performed using an XML file or a graphical wizard. For each unit, the configuration has to be performed once, but can be stored and continuously executed in a continuous integration environment. This step is performed on the Microsoft Windows machines of the engineers.

For the exemplary unit, we give the model file containing only the exemplary unit along with a file setting the model parameters (`TmpConvert_*_C`) to 8Cage. In addition, we tell 8Cage to use 10 simulation steps. In Section 5.3, we see that 10 steps are typically sufficient. As fault models to use, we select the absolute value overflow fault model, division overflow by a small fixed-point number fault model and the signal ranges failure model. For the ranges, we give 8Cage the range -50 to 199 for `TmpOut`.

Upon submitting the configuration, the automatic part of the detection process in 8Cage begins and the smell detection and test data generation step are executed for each selected defect model.

5.2.2 Smell detection

During the second step of smell detection, the model is loaded into Matlab and static block properties are checked according to the image of transformation α. These include block types/names and I/O data types.

For the absolute value overflow fault model, smell detection searches for blocks of type “Abs.” All found “Abs” blocks are checked for their SIO properties to be off and integer input data types. If a block matches, it is marked in the model and code is generated to perform the next step. For the division overflow by a small fixed-point number fault model, smell detection searches for blocks of type “Divide.” with an fixed-point data type input as divisor. All found blocks constitute model smells. This step is performed in Matlab Simulink running on Microsoft Windows.

This step includes launching Matlab Simulink, loading the model and its parameters and searching for the blocks. Even for large models, searching for blocks takes stays within one minute. However, launching Matlab Simulink and loading the model with its parameters may take several minutes.

For the exemplary system, the blocks “Abs” and “Abs1” (mid right) are detected as smells for the absolute value overflow fault model and the block “Divide 1” (bottom right) is detected for the division overflow by a small fixed-point number fault model. The failure models do not contain any smells and, therefore, are not considered in the smell detection.

5.2.3 Test data generation

The third step checks dynamic properties of fault models to describe φ and the φ described by the failure models. It first transforms the Matlab Simulink model into C source code by using the code generator Embedded Coder of Matlab. Smell markers in the models are used to create assertions in the derived code. The C source code is compiled to LLVM bitcode to be symbolically executed using KLEE [25]. As the steps of transformation, marker injection and compilation are always performed subsequently, we refer to them as transpilation. KLEE is directed toward the assertions representing the dynamic properties and generates test input data. In this step, only test input data is generated and the oracle can either be the robustness oracle or must be generated separately according to the respective defect model.

We assume the Software in the Loop (SIL) level to have the same semantics as the MIL level as model to source code compilers have been used for over a decade and their produced code is deployed in the real world. This assumption enables us to re-use existing software for symbolic execution of C source code. In addition to MIL faults, checking the source code at the SIL level also enables further run time failure detection by KLEE. Such failures include memory out of bounds and particularly the transparent division by zero protection invisible to the MIL level. In case the model contains floating point operations, KLEE cannot be used. However, there is a version of KLEE called KLEE-FP [34], which has the capabilities to perform symbolic execution on floating point operations.

The transpilation part of this step takes less than 30 seconds, if Matlab Simulink is still open from the previous step and the model with its parameters is still loaded.

The actual test data generation is the core step in the automatic detection performed by 8Cage. It has the potentially longest execution time of all the steps performed in the analysis. However, it can be parallelized for every found block of a fault model and every failure model. As KLEE is single threaded, the test data generation can be parallelized by running several instances of KLEE at the same time. The transpilation part of this step is performed in Matlab Simulink running on Microsoft Windows, where clang compiled operating system and processor architecture independent LLVM bitcode. KLEE cannot be ported to Windows causing the rest of the step to be performed in Linux.

For the exemplary system, a test case meeting the constraints for each defect model is created. “Abs” and “Abs1” overflow, iff given -32768 as input value of `TmpMtr1Raw` and `TmpMtr2Raw` in the first step. The “Divide 1” block overflows, iff given -255 as

input value of `TmpMtrLastFail` in the first step. The signal ranges are violated, iff 249 is given as input value of `TmpMtr1Raw` and `TmpMtr2Raw` in the first step.

5.2.4 Test case execution

In the fourth step, a test case based on the results of KLEE is created to gather evidence for the presence of an actual fault. The test case is created in a test execution engine for Simulink models created by a project partner. The created test case is run as a Model-in-the-Loop (MIL) test as the single block faults will already fail within Matlab itself. The results are determined and stored for reporting. This step is performed in Matlab Simulink running on Microsoft Windows.

A harness for the Matlab Simulink unit is created in this step, which takes several minutes. After its creation, the test cases can be created and executed. The execution time per test case depends on the complexity of the unit, but should be within 30 seconds.

For the exemplary system, the test cases described in the last step fail. For the absolute value overflow fault model and division overflow by a small fixed-point number fault model, Matlab gives a warning that an overflow occurred including the respective originating block. For the signal ranges failure model, the created oracle gives failure once the value increases above 300.

5.2.5 Report generation

In the fifth and final step, 8Cage generates a report concerning found smells and possible evidence for the presence of an actual fault. This report details results of the test cases and contains them for developer/tester reproducibility. This step is particularly useful in continuous integration [49] environments, where tests are automatically started and reports detail failures. As sometimes faults are intended for performance reasons, this report can be given to the static analysis engineers. Using the report, they can directly spot intended faults and disregard them. This step is performed on Microsoft Windows after the test case execution is complete.

5.2.6 Summary

The steps above constitute an operationalization as found in Section 3.3. It requires a bug pattern (syntactic or static prerequisites) and inputs/outputs of the bug pattern (semantic or dynamic prerequisites) to collect evidence of an actual fault to be present. The bug pattern is detected by using the search functions of Matlab Simulink and the input values are determined by using symbolic execution. The choice to run all Matlab Simulink related steps in Microsoft Windows was made as this simplified the number of interfaces required to access Matlab Simulink and also reduced the number of licenses required to deploy 8Cage. A demo video portraying the steps above can be found at `https://www22.in.tum.de/8cage/`.

5.3 Evaluation

In the evaluation of 8Cage, we particularly concentrated on the automated steps. These are smell detection, test data generation and test case execution. The test data generation step particularly interesting as it is the only step involving non-determinism. The steps of smell detection and test case execution are deterministic in execution and run time.

We evaluate the test data generation step using a hybrid electrical engine control system (ca. 31 000 blocks) and two drive train control systems (ca. 76 000 blocks each). The smell detection on the hybrid electrical engine control system using all defect models above resulted in 29 units with potential over-/underflows. Out of these 29 units, 22 units (between 8 and 655 blocks each) were extractable according to the instructions given by our project partner. For two drive train control systems, 3 units (between 50 and 155 blocks each) were extracted from first and 7 units (between 7 and 26 blocks each) were extracted from the second. The extraction of the units from the first system revealed a weakness of 8Cage. It came to light that the prototype actually used floating point arithmetic in most of its units. Since we developed 8Cage using KLEE, only 3 units with potential defects and no usage of floating point were extracted. The second system also revealed a weakness as it used specific global variables linked within a plethora of units. The extraction of any unit using these global variables was impossible since (1) the extraction method given by the project partner did not work and/or (2) the global variables were accessed in multiple units making a unit separation impossible.

Our evaluation hardware systems are a quad-core Intel Core i7 2640M with 8 GB of RAM running Microsoft Windows and an octa-core Intel Xeon E5540 at 2.5 GHz with 40 GB of RAM running Ubuntu Linux. Matlab Simulink, Clang and the unit/integration testing framework of our project partner are running on the Windows machine while only KLEE [25] is running on the Linux machine. However, KLEE is single threaded making only use of one core and has a 2GB memory limit. The linux machine was merely used to run multiple instances of KLEE at the same time without performance implications. All units were configured to execute for 10 time steps[1] with a timeout of 1 minute and use the embedded coder of Matlab for C source code generation. All steps were performed 10 times to counter effects of single runs and obtain an average run time of each step. We focus particularly on the test data generation step executing KLEE using the command line in Listing 2.2 as it includes non-determinism.

The evaluation's goal is to show (1) reproducibility, (2) effectiveness, and (3) efficiency of 8Cage and, therefore, of the defect models introduced in Section 5.1.

Firstly, symbolic execution as implemented in KLEE [25] is non-deterministic as choosing a path may occur at random. Thus, reproducibility must be assessed to

[1]We hypothesize the time to find failure-causing inputs to increase when executing more time steps and impact scalability of the approach

evaluate if multiple executions lead to the same result. In practice, every execution of 8Cage should either lead to a test case giving evidence to the presence of a fault or nor give a test case at all. Low reproducibility in the generation of test data would render 8Cage unbeneficial in practice.

Secondly, the effectiveness of 8Cage must be evaluated w.r.t. the faults detected by 8Cage to assess the benefit of using 8Cage. We execute 8Cage on all extracted units and look into the detected defects and their severity.

Thirdly, the efficiency of 8Cage needs to be evaluated w.r.t. the time consumed in the test data generation step. The time consumed must be reasonable to run 8Cage overnight for deployment in a continuous integration context. Thus, the execution time for all extracted units of one system on our standard hardware evaluation systems is not to exceed 8 hours as defined reasonable by our industry partner.

5.3.1 Reproducibility

Concerning the reproducibility of the results, smell detection and test case execution are deterministic and always produce the same result. To assure reproducibility of the test data generation, we examined the extracted failure-causing inputs at the end of each run of the test data generation. For all evaluate systems, it turned out that the inputs were always failure causing and only had minor variations in signals with no impact in the provoking of the failure.

Thus, we conclude the failure-causing inputs completely reproducible for the systems and units used in the case study.

5.3.2 Effectiveness

For the electrical engine control system, we have found inputs for the FaultFilerting unit such that an underflow can cause the system to go into fault mode and potentially stop working within the first step. In a re-used component called Tlbx_TransientFilterValidBitSet we found an underflow during a subtract in a unit calculating a derivative. This underflow leads to out-of-range outputs of several units. The results of the evaluation can be found in the remark column of Table 5.2, where the first column (#) represents the index of the unit under test. A unit to number mapping can be found in Table 5.1.

In the two power train control systems, we have found inputs for the Calculate output, ErrCanMon and Subsystem unit such that an underflow in an addition can occur. Calculate output and ErrCanMon are only used once in each respective system. However, Subsystem is re-used a number of times, but has a special TestPoint at the output of the absolute value. Thus, we think the engineers took care of testing this overflow and other units likely take care by setting inputs such that an overflow cannot occur. The same is true for Calculate output and ErrCanMon, where inputs seem to represent modes, that typically do not reach the maximum of the data type, these overflows could be unit specific. The results of the evaluation of the two power train

control systems can be found in Table 5.4. A unit to number mapping can be found in 5.3.

One interesting aspect of the evaluation was the deliberate test of the overflow protection by 8Cage. The units APTorque1 and ManipulationKorrekturWert both contained overflow protections. 8Cage did not come up with a test case to cause an overflow giving an increased confidence, that the overflow protection measures are working. However, when the measures were deactivated, 8Cage could produce an overflow test case and give rise to the sensibility of the overflow protection. Thus, 8Cage gives a first indication of the sensibility of overflow protections, but other tools such as Polyspace must be used in addition to get a proof via abstract interpretation (see Section 2.1.3).

We forwarded the detected faults to the developers of the systems and they confirmed them to be actual faults. When inquiring about other software faults detected during the test of the examined systems, developers responded with functional faults arising from either a wrong specification or implementation. Since these faults are not the target of 8Cage, 8Cage did not detect them.

Thus, we conclude 8Cage to be effective at detecting over-/underflows for the systems and units used in the case study.

5.3.3 Efficiency

Concerning the efficiency (i.e. time for the steps smell detection, test data generation and test case execution), the reference system large_example took an average of 45.6 seconds for the smell detection, 25.8 seconds for the transpilation part of the test data generation, 18 seconds for symbolic execution part of the test data generation and 214.4 seconds for the test case execution.

For the electrical engine control system, the smell detection took 147.3 s, the transpilation part of the test data generation took 68.6 s and the test case execution took 236.7 s on average. The symbolic execution part of the test data generation for all other extracted units took even less time to find a fault than the baseline. The longest time was 9 seconds for the PI Controller unit. All other units took less than 3 seconds. The result of each execution of symbolic execution and its average can be in columns 2 - 12 of Table 5.2.

For the two power train control systems, the smell detection took 78.8 s, the transpilation part of the test data generation took 28.9 s and the test case execution took 222.3 s on average. Similar to electrical engine, all units took less than 3 seconds to symbolically execute. The result of each execution of symbolic execution and its average can be in columns 2 - 12 of Table 5.4.

For all executions of the smell detection, transpilation part of the test data generation and test case execution in all evaluated systems had less than 2 seconds of standard deviation deeming these execution times stable. When at the individual execution times of these steps, the smell detection step seems to be correlated with

the number of blocks. However, the correlation is likely non-linear as the electrical engine control system had a two orders of magnitude higher block count. For the test case execution, only one unit-size piece of the system is involved, which yields similar execution times.

In case no fault was detected in a unit, we injected a fault near the outputs to see whether (1) KLEE was unable to symbolically execute the unit or (2) there was actually no detectable fault. In all cases, the injection yielded a test case in less than 3 seconds. Thus, the detection time for the symbolic execution with KLEE stayed below a 20 seconds threshold.

All evaluated units did not contain any Stateflow diagrams. To evaluate 8Cage also w.r.t. Stateflow diagrams, we measured the time to find failure-causing inputs in units containing Stateflow diagrams. As none of the units had Stateflow faults, we injected a fault into a transition in a nested Stateflow diagram of the VirtualClutch unit. The time for the symbolic execution to find failure-causing inputs was less than 3 seconds and 1.5 seconds on average.

One hypothesis in the evaluation of efficiency was an increase in execution time with a growing number of states. States can be introduced by delay blocks (i.e. latches) in Matlab Simulink and the PI Controller as well as the large example use unit delay blocks. The maximum number of steps required in these to detect the fault units was 3 steps. To test this hypothesis, we built a system with an absolute value overflow only possible after 15 time steps. Again, failure-causing inputs were found within 3 seconds on average. If not dependent on states, we hypothesized there to be a correlation between the test input generation run time and the number of blocks and/or the number of inputs. To test this hypothesis, we generated inputs on a completely integrated System Control component consisting of multiple dozen units. These inputs were found within 0.2 seconds on average and falsified the hypothesis. Finally, we arrived at the hypothesis of more simulation steps to yield an increase in execution time of test case generation. To evaluate this hypothesis, we changed the PI_Controller (longest time to find failure-causing inputs in a real-world system) to use 20 simulation steps instead of 10. The average time to find failure-causing inputs was the same as with 10 steps as the failure-causing input only required 5 steps.

Thus, we speculate all failure-causing inputs in units as found in our three evaluation systems to be performable within 20 seconds. In addition, injected faults are detected in Matlab Simulink and Stateflow units within the same time frame. Generally, the number of blocks, inputs and states does not affect the test case generation or our experiments only included units below its efficiency threshold. In sum, all automatic steps can always be executed in an eight hour overnight build. More precisely, our results indicate a typical execution times of minutes rather than hours.

Table 5.1: Test data generation run time evaluation of 8Cage: electrical engine control system: unit name mapping

#	Name	# blocks	# interfaces
1	SysSup_CurrOffsNotCalib	7	1
2	large example (reference system)	45	4
3	PI_Controller	47	8
4	ConversionToThreshold	67	7
5	SysSup_CheckTrqReq	68	4
6	SysSup_BISG_SlipDetection	153	5
7	CalDiag_JobCheckResolver	27	3
8	ExternalTorqueLimitation	31	5
9	VirtualClutch	32	4
10	SysSup_ClCloseInStandby	14	2
11	FaultFiltering	655	23
12	SysSup_DCCFailure	108	2
13	SysSup_CheckHvDcUnplugged	83	5
14	Tlbx_StatisticFilter1	26	2
15	SwitchingFrequencyLimitation	568	3
16	SysSup_CheckBatteryContactor	130	2
17	SysSup_CheckCrash	55	1
18	CalcPos	142	13
19	FaultProcessing	568	16
20	SysSup_DCCFailurel	62	2
21	Tlbx_TransientFilterValidBitSet	38	1
22	IapFltActive	8	0

Table 5.2: Test data generation run time evaluation of 8Cage: electrical engine control system: timings (time in s)

#	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7	Time 8	Time 9	Time 10	Average	Remark
1	40.007	47.007	42.006	31.011	32.012	32.011	34.006	32.001	31.009	32.006	35.3076	No defect found
2	18.811	8.856	31.672	8.832	20.012	8.791	5.498	26.503	40.817	17.827	18.7619	Deliberate overflow
3	16.132	7.478	7.674	7.089	1.913	6.815	30.185	1.854	7.095	7.483	9.3718	No defect found
4	0	0	0	0	0	0	0	0	0	0	0	No defect found + timeout reached
5	0	0	0	0	0	0	0	0	0	0	0	No defect found + timeout reached
6	2.908	2.132	2.94	2.968	3.008	3.028	2.93	2.909	2.931	2.94	2.8694	Abs overflow due to subtraction overflow
7	0	0	0	0	0	0	0	0	0	0	2.5	No defect found + timeout reached
8	2.009	2.009	2.007	2.009	2.007	1.01	2.009	2.01	2.009	2.005	1.9084	No defect found
9	1.463	1.502	1.611	1.629	1.485	1.402	1.419	1.659	1.493	1.64	1.5303	No defect found
10	0	0		0	0	0	0	0	0	0	0	No defect found + timeout reached
11	0.913	0.974	0.919	0.922	0.92	0.916	0.912	0.927	0.949	0.91	0.9262	Possible fault mode due to underflow of uint16
12	0.289	0.288	0.299	0.273	0.311	0.325	0.322	0.294	0.294	0.289	0.2984	Integer undeflow in Tlbx_TransientFilterValidBitSet
13	0.309	0.286	0.296	0.275	0.266	0.298	0.288	0.293	0.309	0.314	0.2934	Integer undeflow in Tlbx_TransientFilterValidBitSet
14	0.146	0.146	0.163	0.19	0.169	0.183	0.17	0.148	0.178	0.154	0.1647	No defect found
15	0.123	0.175	0.131	0.183	0.145	0.153	0.173	0.188	0.21	0.13	0.1611	Integer undeflow in Tlbx_TransientFilterValidBitSet
16	0.137	0.131	0.164	0.16	0.166	0.142	0.159	0.14	0.172	0.167	0.1538	Integer undeflow in Tlbx_TransientFilterValidBitSet
17	0.139	0.128	0.112	0.145	0.12	0.134	0.133	0.13	0.116	0.129	0.1286	Integer undeflow in Tlbx_TransientFilterValidBitSet
18	0.13	0.094	0.097	0.13	0.136	0.139	0.132	0.11	0.153	0.142	0.1263	Found abs overflow
19	0.234	0.109	0.107	0.11	0.121	0.119	0.117	0.114	0.105	0.11	0.1246	Integer undeflow in Tlbx_TransientFilterValidBitSet
20	0.082	0.08	0.111	0.108	0.119	0.124	0.113	0.122	0.112	0.129	0.11	Integer undeflow in Tlbx_TransientFilterValidBitSet
21	0.091	0.109	0.102	0.088	0.092	0.095	0.094	0.066	0.102	0.084	0.0923	Deliberate overflow with saturate
22	0.076	0.077	0.085	0.085	0.084	0.08	0.084	0.086	0.081	0.087	0.0825	No defect found

Table 5.3: Test data generation run time evaluation of 8Cage: electrical engine control system: power train control systems: unit name mapping

#	Name	# blocks	# interfaces
1	ETC2_TxCurrRng_Cval	50	1
2	ETC2_TxSelRng_Cval2	81	4
3	Calculate output	155	2
4	AP_Torque/APTorque1	26	3
5	ManipulationKorrekturWert	9	1
6	ErrCanMon	7	2
7	FIT_AP/AP_RQV/TorqueRQV	18	2
8	DiffError	21	2
9	Subsystem	14	3

Table 5.4: Test data generation run time evaluation of 8Cage: electrical engine control system: power train control systems: timings (time in s)

#	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7	Time 8	Time 9	Time 10	Average	Remark
1	20.011	20.007	20.007	20.006	19.006	21.013	21.007	21.008	21.007	21.008	20.408	No defect found
2	62.008	62.008	62.01	62.011	62.01	62.01	62.01	62.008	62.009	62.008	62.0092	No defect found + timeout reached
3	1.055	1.052	1.05	1.044	1.051	1.064	1.053	1.057	1.061	1.054	1.0541	Potential overflow in addition
4	5.838	5.887	5.82	5.831	5.836	5.848	5.869	5.859	5.829	5.858	5.848	Tested overflow protection is required
5	1.015	1.013	1.014	1.01	1.011	1.013	1.009	1.013	1.012	1.012	1.012	Tested overflow protection is required
6	1.032	1.047	1.044	1.038	1.048	1.037	1.041	1.045	1.037	1.038	1.041	Potential addition overflow
7	1.013	1.011	1.008	1.012	1.012	1.012	1.012	1.011	1.012	1.011	1.011	No defect found
8	1.012	1.013	1.012	1.012	1.012	1.007	1.012	1.014	1.013	1.011	1.012	No defect found
9	1.056	1.045	1.047	1.047	1.045	1.047	1.048	1.05	1.05	1.048	1.048	Tested overflow protection is required

5.3.4 Summary

In summary, 8Cage demonstrated reproducibility, effectiveness and efficient procurement of the results for the systems and units used in the case study. Albeit, there were several threats to validity concerning our case study. Our aim in the selection of the systems in the evaluation was to be representative. However, our subject selection (1) had to be performed on an opportunistic basis as the system needed to be in development / test by our project partner and had to allow the derivation of C source code from the model of the system. In addition, we did not detect any fault other than overflows (2) in the Matlab Simulink systems, which may be due to them not being contained in the system or 8Cage not being able to detect them. Based on these results, we injected faults into the Matlab Simulink and Stateflow units and detected them efficiently. However, the injection may be unrealistic (3). Thus, our results may not generalize to all Matlab Simulink / Stateflow systems.

One of the units in the evaluation contained a PI controller. Closed-loop controllers are typically hard to symbolically execute as they contain a feedback loop back to the controller (see Figure 2.1). In our evaluation, the average time for the symbolic execution was less than 10 seconds for the 10 steps. We speculate the execution time to be low as the PI controller only controls a small part of the system and may have been simplified for this part. We also only performed 10 and 20 steps with no difference in execution time. However, this may not be enough to provoke a failure. For any complex controller, we speculate the execution time to increase and the number of steps required to provoke a failure to be higher.

During the extraction of the units, one drawback of 8Cage became clealry visible: the units may have dependencies to other units / components and the environment. One instance was the usage of global variables. These global variables could be set in other parts of the system when executing it completely. Thus, the unit under test would be dependent on the global variables during execution and change behavior according to the values contained in them. This made the testing of these units problematic since the complete behavior could not be assessed on the unit level.

There were several aspects noteworthy when performing the evaluation of 8Cage. The inputs of the units used for the evaluation was the usage of seemingly much larger precision than required (1). All fixed-point calculations were performed with 32-bit integers on 28-bit of precisions. However, neither parameters nor signal ranges required such high precision. Furthermore, some parts of the code were not reachable due to configuration parameters within the units (2). For instance, the VirtualClutch unit had a parameter whether to use the virtual clutch, which was configured to false. Thus, a whole contained Stateflow diagram turned out to be dead code in this configuration.

5.4 Related Work

8Cage is related to static analysis tools. Tools using abstract interpretation are Astrée [37] and Polyspace Code Prover [130]. Both can detect run time failures such as over-/underflows and out of bounds pointers as well as potentially infinite loops and potentially dead code. However, it requires an expert to prepare a software system for analysis in these tools as they work on a C/C++ code level. Thus, analyzing each unit often is deemed excessive work and only the completely developed system is analyzed. Results are returned by coloring source code—code verified to cause a run-time failure; code that may cause a failure; code that will not cause a problem; and code deemed dead. Unfortunately, because of approximations, often large parts of the source code is marked as potentially failure causing. Then, an expert has to walk through the C code. Finally, faults must be localized to go from the reported failures to the actual faults. This result and suggestions to fix are then given back to the developers (or testers in some cases). 8Cage allows the developer/tester to start the automatic detection and yields results directly traceable to blocks in the model. Thus, no expert is involved in the analysis. However, 8Cage only detects faults related to single blocks on a unit/integration level. This makes the use of static analysis tools still advisable to detect other faults at a later stage.

Static analysis is also performed by smell finding tools such as Lint [81], Simulink Model Advisor [107] and Polyspace Bug Finder [130]. They return a plethora of possible problems for C code without dynamic execution. 8Cage also performs a lint-like smell detection of the fault pattern, which could be performed by any other smell detection tool. However, after the detection of the smell, 8Cage always aims to gather evidence for an actual fault to be present by creating a test case leading to a failure. This second step is not performed by other smell finding tools leading to many false positives. In fact, the number of false positives tends to discourage developers and testers from checking for an actual fault to be present.

Tools related to 8Cage in the field of model-based testing of Simulink models are, among others, Simulink Design Verifier [108], Reactis [131] and TPT [146]. Design Verifier allows the detection of dead logic, integer and fixed-point overflows, array access violations, division by zero in an automatic fashion. The other two tools allow the manual specification and automated execution of test cases. Design Verifier and Reactis can generate test cases for coverage. In addition, Reactis can generate random test inputs. TPT uses a graphical test case notation abstracting from actual I/O in a keyword-driven way. In contrast, 8Cage generates test cases that target specific faults within the Simulink model by operationalizing defect models. It is thereby able to detect all faults described in Section 5.1 and further faults specified by developers and testers in custom fault models.

5.5 Discussion and Conclusion

We have described an operationalization of defect models using 8Cage. 8Cage detects faults related to single blocks / block combinations in Matlab Simulink models. These faults are described as defect models (see Chapter 3) using parts of the Simulink / Stateflow model syntax with a minor extension. The fault models currently operationalized in 8Cage relate to over-/underflows, divisions by zero, comparisons/rounding, Matlab Stateflow (state chart implementation in Matlab Simulink) faults and loss of precision. The failure models relate to exceeded I/O ranges. The defect models are operationalized by 8Cage in automatic detection consisting of smell detection, test data generation and test case execution. During smell detection, the static aspects of a block (i.e. the image of the transformation) such as its type, name and properties are utilized to find potentially failure-causing blocks (i.e. model smells) as a syntactic or static prerequisite. Test data generation performs a symbolic execution using the semantic or dynamic prerequisite to search for failure-causing inputs to the model (i.e. an input space partition). Test case execution creates a test case using the failure-causing inputs and provides evidence for a fault. Note that, smell detection is only required for fault models, but not for failure models.

Scalability of symbolic execution is a concern (see Section 2.2). Test data generation may take significant proportion of the analysis time. However, experience suggests that almost all single block faults can be found within the first 10 time steps, a pattern also exploited by unfolding heuristics in static analyzers. Test data generation has to be performed for every potentially failure-causing block. Because test data generation is independent for each model smell, parallelization may yield performance increases.

We have evaluated 8Cage on 29 units extracted from three Matlab Simulink systems. In the evaluation, 8Cage demonstrated reproducibility, effectiveness and efficiency when creating the results in the case study. Even the results of efficiency w.r.t. the execution time of the symbolic execution part of the test data generation turned out to be scalable for the evaluated units. Major concerns were the representativeness of the selected systems and no detected faults other than overflows limiting the generalizability of the case study. Further major concerns were control systems, global variables and floating point numbers. Control systems have an inherent loop back to the controller and their analysis was demonstrated for the selected systems. However, the results may not be generalizable either. Global variables were used by multiple units and these variables can be written outside of the unit. This access could lead to faults not detectable on a unit level with 8Cage. As electronic control units in the automotive area often lack floating point units because of cost, this is not a drawback for 8Cage yet. However, it is projected that floating point units will be built in within a few years. Thus, we are currently looking into a floating point version of KLEE [35].

8Cage can solely detect prespecified faults. New or different faults need to be specified to be detectable. Thus, the library of fault models needs to be maintained

(for maintenance of defect models see Section 8.2). Concerning the specification, one major issue seen when performing the evaluation was the dependency on the unit specification. If this specification is wrong or missing, the results of 8Cage may contain false positives since 8Cage uses the full range of values for all signals by default. This value range may not be achievable because of range limits in a part providing the inputs to the unit examined by 8Cage. In addition, true positives may not be found because of the parametrization (configuration) of the system disallowing taking paths to the smell.

We are unaware of tools to provide evidence of actual faults related to single blocks / block combinations to be present in Matlab Simulink models at an early stage. Of course, Simulink Model Advisor [107]is able to produce potential faults, but also creates many false positives. When presented with 8Cage, model developers/testers had a very positive response. 8Cage enables developers to detect these faults before even checking their model into version control. Moreover, these faults could be detected in continuous integration [49] leading to their continuous nightly detection. In some cases, developers use under-/overflows deliberately to make computations faster. In these cases, the results of 8Cage can be given to the static analysis responsibles to make them aware of these deliberate (desired) under-/overflows.

In the future, we want to compare 8Cage to other techniques, such as reading techniques using checklists. Reading techniques could be more efficient than the detection performed by 8Cage. Particularly for more complex faults including multiple blocks or multiple units, we speculate checklists to be the better choice. In addition, 8Cage is a first operationalization on the unit testing level according to second generic operationalization scenario in Section 3.3. Using the experiences gained in this operationalization, other operationalizations in different implementation languages on the unit are analogously creatable. For instance, these could target #13 in the top 14 technical defects of Table 4.2 in web-based information systems by explicitly creating test cases to test the input sanitation. Such test cases could then not only check user input, but also try to provoke security faults leading to an SQL injection failure.

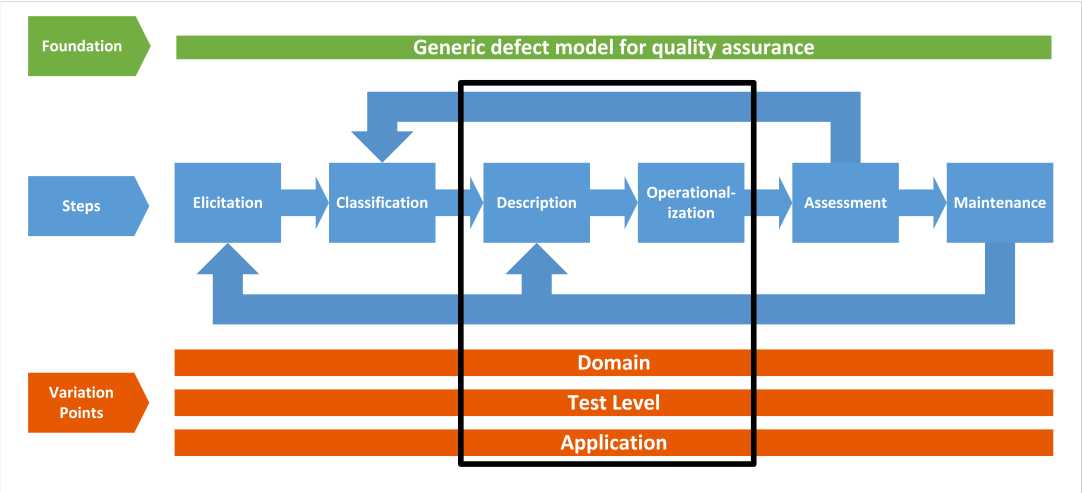
6

Description and Operationalization: OUTFIT for integration testing

In the previous chapters the foundations for a systematic and (semi-)automatic approach to defect-based quality assurance based on defect models were defined, relevant defects have been elicited and classified and a first description and operationalization on the unit level called 8Cage has been created. To demonstrate the effectiveness and efficiency of the operationalization of defect models in integration testing, this chapter describes and operationalizes defect models in a tool called OUTFIT (short for prOfiting from Unit Test For Integration Testing), which re-uses unit tests to perform integration testing. Concerning the variation points OUTFIT resides on the integration testing level in the Matlab Simulink domain. The failure models described and operationalized in OUTFIT are application and specification independent. However, for one of the defect models, the integration must involve fault handling. This chapter is the second of three descriptions and operationalizations of defect models residing in the method application step of the lifecycle framework as seen in Figure 6.1.

Recalling the definition of integration testing in Section 2.1.2, integration testing tests the interaction between two or more software modules / components. It aims at detecting faults concerning interfaces, design decisions and non-functional requirements. In theory, an explicit integration testing strategy prescribes how combinations of components are integrated and what is to be tested. Each integration test requires test cases, a number of components to be tested and a test environment consisting of stubs and drivers emulating components not part of the integrated system. Because this is too expensive, in practice, integration testing is done more opportunistically. Typically, integration is done for a sensible, usually ad-hoc, selection of components, at a permissible cost for the creation of test cases and test environments. However, using such an integration testing strategy (1) may increase the effort for fault localization as many components are integrated at once and (2) may not detect particular defects detectable only when integrating a small number of components.

Figure 6.1: Position of the operationalization OUTFIT in this thesis



In Chapter 4, we have elicited and classified common and recurring defects. Particularly, we identified several domain-independent defects particularly detectable and common in integration testing. These include superfluous or missing functionality and untested exception/fault handling. By targeting these defects in the integration testing of two components or subsystems, we aim for their (semi-)automatic detection. As in the previous chapter, we describe the defects above in defect models to automatically derive test cases and test environment in their operationalization. Since the defects above can be caused by a previously unknown set of transformations, we use failure models for their description and operationalization.

By describing and operationalizing, we front-load their detection by performing integration testing re-using unit tests as components become available. Particularly, the unit tests can be created automatically or manually, with high coverage being the only condition for their re-use. Although the failure detection ability of test cases for coverage is disputed (see Chapter 3 and [77, 129, 150]), they are able to detect common and recurring defects found in integration testing of components connected in the pipe and filters pattern. By almost full automation, we are able to add additional defect-based integration tests for execution on any (or specific) aggregates. Since this approach directly profits from unit tests for integration testing, the name OUTFIT was chosen for the operationalization. Recall that Matlab Simulink commonly uses the pipe and filters pattern in the design and implementation of embedded systems (see Section 2.3) and is particularly used at our industry partner. However, using knowledge of typical failures in integration testing is also possible in other contexts. Note that, the signal range failure model is able to target the common and recurring defect of signal range violations leading to failures on the integration test level. We applied this failure model at the level of unit testing, but it could intuitively be re-used on the level of integrated components. Only the possibility to perform symbolic execution on the respective components is required.

To the best of our knowledge, this is the first operationalization of failure models for integration testing (as described by [71]). OUTFIT automatically creates integration tests and environments. Only the test verdict has to be inferred by manual inspection. Our approach also reduces the effort of fault localization for the presented defects as the automatic bottom-up integration of the system only requires small parts of the system to be inspected. We evaluate our tool on three representative components of a real-world electrical engine control system of a hybrid car. We can demonstrate reproducibility, effectiveness and efficient procurement for practical applications.

6.1 Failure Models

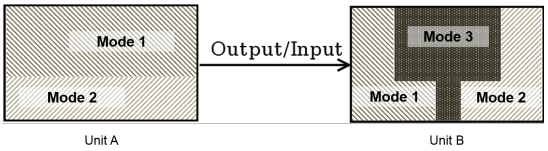
The failure models for integration testing were elicited and classified in the field study of common and recurring defects for the creation of defect models (see Section 4.3). Two defect models for integration testing were elicited and classified in this study and are described in detail on a program/domain/paradigm/-independent level of abstraction in the following.

6.1.1 Superfluous or missing functionality

A common and recurring defect concerning integration testing is missing or superfluous functionality. Superfluous functionality typically occurs due to cloning or over-engineering. When re-using large software parts of existing projects, developers may not have the full knowledge of all functionality a component implements. Some functionality may have been project specific making it dead code. Missing functionality typically occurs due to changes made to one component, but neglected in another component. Finding missing or superfluous functionality is not only important for the correct functioning of the system, but also for fulfilling the non-functional requirements. As an example, when an additional mode of the system was introduced, this addition was forgotten in one component, resulting in the scenario shown in Figure 6.2. Whenever the system switched to mode 3, the missing functionality for mode 3 in component A caused a failure. Since mode 3 was rarely used, the fault localization effort of the system testers was large. In a further example, a component was copied from an old system. This component compensated for the signal jitter of a sensor. However, the sensor was replaced and the jitter compensation was no longer required, which created a computational overhead and made some of its functionality superfluous. Thus, our first failure model aims to front-load the detection of missing and superfluous functionality with reduced localization effort.

According to the definition of the generic defect model in Chapter 3, a failure model characterizes the set of failure-causing inputs φ of the input domain of a program. Test cases selected from the approximation $\tilde{\varphi}$ are then potentially failure-causing, defect-based and "good" (see Chapter 3). To formally describe φ for this failure model let the input space be all possible calls in the integrated system containing both components

Figure 6.2: Example of missing functionality

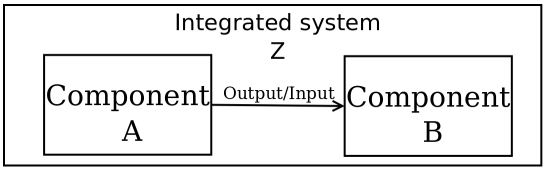


above. We deliberately leave the notion of call abstract as these could be function calls in imperative programming languages, method calls in object-oriented programming languages or simply calls passing data for data-flow oriented languages. Formally, this space is partitioned into four blocks. The first block (1) contains all calls causing the interaction of the components as one component needs to call the other for the completion of the call. The second (2) and third (3) block contain calls handled by a single component (i.e. that do not yield interactions with other components). The fourth (4) block contains all other calls (i.e. that lead to arbitrary other interactions). Recall that, the operationalization of φ is impossible as it is in general undecidable, if a call will lead to the interaction of the components or not . However, empirical methods of path exploration (e.g. symbolic execution) were proposed in Chapter 3 for the approximation of this component interaction leading to the approximation $\tilde{\varphi}$. The set produced by $\tilde{\varphi}$ then contains calls belonging to the first, second and third block. This set of calls is rather large, but we want to capture all possible interactions as to excercise as much functionality within the components as possible. Calls in the first block are able to reveal missing functionality of a component when it is interacted with. Calls in the second and third block reveal potentially superfluous in the called component as they are handled by a single component. The operationalization of this failure model is (semi-)automatic as only a robustness oracle can be automatically created, which is sufficient in this case. In Section 6.2, this failure model is operationalized using coverage test cases in Matlab Simulink systems, which reveal missing/superfluous functionality after manual inspection (see Figure 6.5).

6.1.2 Untested integrated exception/fault handling

A further common and recurring defect concerning integration testing is exception/fault handling in untested compositions. Every system has exception/fault handling on a unit basis. If an exception/fault cannot be handled by the unit, it will be propagated to a higher instance. This higher instance may propagate the exception/fault to even another instance, thereby creating a propagation chain until either handling is performed or the propagation reaches the application boundary and terminates it ungracefully. In real-world systems, the ungraceful termination of the application must strictly be avoided. Thus, there is typically a central component dealing with exceptions/faults within the system. This component handles system degradations after partial system malfunctions and graceful shutdowns. Not testing the integrated exception/fault handling of all propagators to this central unit can lead to undesired

Figure 6.3: Integrated system Z of A and B



system failures. This may cause anything from data loss in IT systems to death in safety-critical systems. As a real-world example, a fault in the exception handling of an IT system led to the termination of the entire application because a single component's exception handling forced the application to exit after an exception was not handled. A further example was a fault in the fault handling of an embedded system leading to no reaction of the system although an emergency shutdown was required. Both faults were only detected after deployment and system testing respectively and yielded a large effort in localization. Thus, our second failure model aims to front-load the explicit test of integrated exception/fault handling with reduced localization effort.

The set produced by φ according to the generic defect model in Chapter 3 for this failure model again a subset of all possible function calls. More precisely, φ contains all calls leading to unhandled or wrongly handled exceptions/faults. In practice, the formal description again is not operationalizable as it is in general undecidable whether a call leads to unhandled or wrongly handled exceptions/faults. Thus, empirical methods can be used to approximate φ as $\tilde{\varphi}$ using path exploration to find all calls leading to an exception/fault in one component propagated to the next higher instance / central component. Again, the operationalization of this failure model is (semi-)automatic as only a robustness oracle can be automatically created. In Section 6.2, this failure model is operationalized using coverage test cases in Matlab Simulink systems for the unit exception/fault handling. Potentially exercising all combinations of exceptions/faults then reveals untested integrated exception/fault handling by sorting test cases by there outcome for the system.

6.2 Operationalization

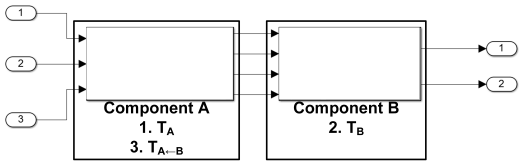
To perform integration testing of Matlab Simulink systems, let Z be the composition of two components A and B as shown simplified in Figure 6.3. To be as generic as possible, a component in this context is defined as either a unit or a composition of other compositions or units. Recall that, Matlab Simulink uses a block-based data-flow driven notation where blocks represent operations and lines represent data flow (see Section 2.3). Thus, each component has one or multiple data inputs and one or multiple data outputs. Matlab Simulink components typically implement a sequence of operations on data where the output signals (i.e. data values over time) of A act as the input signals of B (see Figure 6.3). An example for A is signal processing where B is a control system. The integrated system Z then has the inputs of A and the outputs of B.

6.2.1 Detecting superfluous or missing functionality

To operationalize the first failure models in Section 6.1.1, we re-use or generate high-coverage unit/component test suites in our OUTFIT approach. These test suites either exist from previous unit/component testing stages, or are automatically created using symbolic execution with KLEE [25]. After the execution of the test suites, manual inspection of covered parts of the system reveals superfluous or missing functionalities. OUTFIT performs three fully automatic steps to procure the coverage measurements: Model Transpilation, Test Case Creation, and Test Case Execution.

In the following, these steps are explained for the situation where there are no tests for either component. If instead test suites for components A and B are available, functional end-to-end testing of the composition Z is directly possible by matching actual outputs of T_A with the required inputs of T_B under the assumption of no other component being involved. However, issues when combining the test suites include deviation in time and value dimension of the signal and have extensively been discussed in signal processing [18].

Figure 6.4: Components involved in test case generation

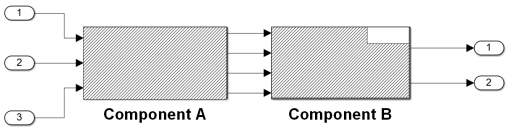


1. Model Transpilation

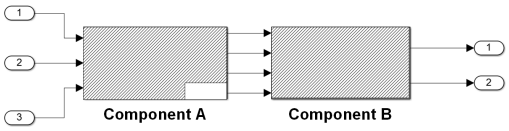
To generate high coverage test cases at the component level with KLEE, components A and B are extracted from the Matlab Simulink system model and transformed into C code (equivalent to the transpilation part of the test data generation of 8Cage in Section 5.2.3). This transformation to C code and compilation (i.e. transpilation) is done by the embedded coder or Matlab Simulink and commonly performed for the deployment of the system. For each component, the input signals are made symbolic in the C code. This means that symbolic execution, which we use for test case generation,

Figure 6.5: Exemplary coverage results of A and B in Z after executing T_A (a) and $T_{A\leftarrow B}$ (b)

(a) Superfluous functionality in B or missing functionality in A



(b) Superfluous functionality in A or missing functionality in B



will perform path exploration using these input signals. The C code for each component is separately compiled to LLVM bitcode using clang.

2. Test Case Generation

The two LLVM bitcode files produces by the Model Transpilation are then given to KLEE. KLEE automatically creates a best effort high coverage test suite, aiming for full path coverage. Because the input channels of A and Z coincide, the generated component test suite of A (T_A) can directly be executed in Z. For the test suite of B (T_B), inputs of A have to be derived that let A produce the output required by each test case in T_B. By collecting the inputs for each test case in T_B and using them as an output constraint for A, test cases for A leading to the inputs required by B ($T_{A \leftarrow B}$) are derived by KLEE. All created test suites and parts of the integrated systems are shown in Figure 6.4.

3. Test Case Execution

The generated test suites T_A and $T_{A \leftarrow B}$ are given to a unit/integration testing framework and executed separately as model in the loop tests (MIL) in Z. Z is an excerpt of the system model and also automatically generated as test environment. The test case execution step presents an instance of robustness testing as the test suites do not contain oracles and only system crashes/errors can be detected.

4. Manual Inspection

As a last manual step, an inspection of test suite coverage is performed on Z using the integrated model coverage features of Matlab Simulink[1].

The manual inspection of the coverage produced by T_A reveals missing functionality in A or superfluous functionality in B as shown in Figure 6.5a. After executing the high coverage test suite T_A for A, the functionality of A is (completely) covered (shaded in Figure 6.5a). In case there is an uncovered part of the functionality of B (white in Figure 6.5a), this either means (1) it is superfluous as it is not used by A or (2) there is missing functionality in A, which is supposed to use this part of the functionality in B . Vice versa, the manual inspection of the coverage produced by $T_{A \leftarrow B}$ reveals missing functionality in B or superfluous functionality in A as shown in Figure 6.5b. If B is completely covered (shaded in Figure 6.5b), any uncovered part of the functionality of A (white in Figure 6.5a) is either superfluous or there is functionality missing in B. Interestingly, the inspection does not only form the test verdict, but also reduces the effort for the fault localization to the two considered components.

Applying this operationalization to the system in Figure 6.2, the results of T_A leaves mode 3 in B uncovered and a manual inspection of the uncovered functionality reveals the inability of B to switch to mode 3. A following inspection of the causes reveals and localizes the missing mode in A. Executing $T_{A \leftarrow B}$ reveals no defect as there is no

[1] `http://www.mathworks.com/help/slvnv/ug/types-of-model-coverage.html`

superfluous functionality in A or missing functionality in B for the integrated system shown in Figure 6.2.

Dependencies

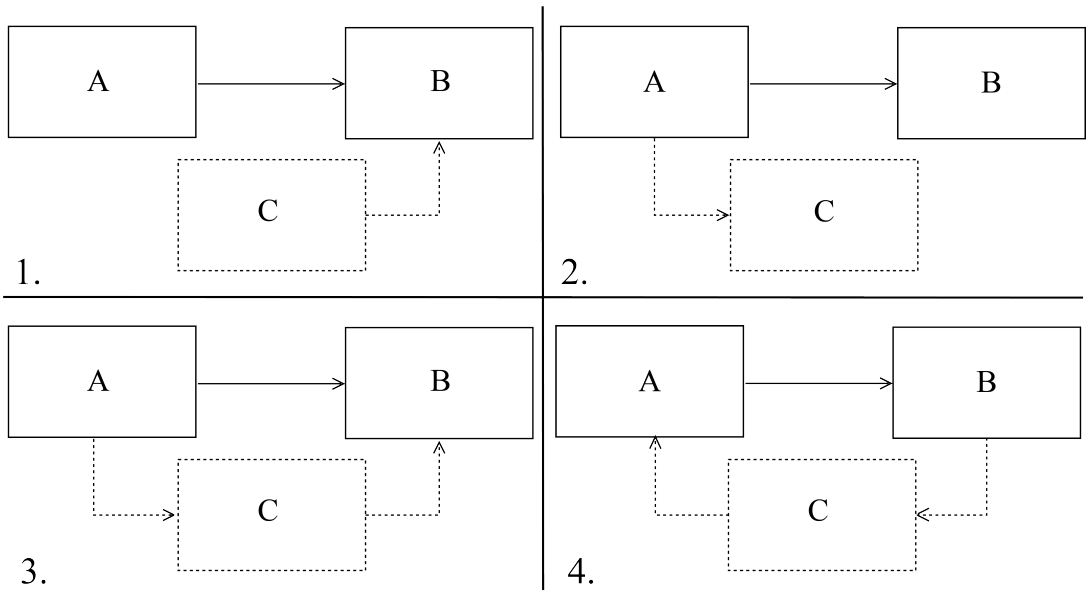
In a typical system, components A and B are not only connected in the back-to-back style sketched in the scenarios above, but send and receive data from other source or loop back to themselves. Other sources can be other components of the system or external systems. For the integration test scenarios, there are four scenarios of sending/receiving outside data from a component C as seen in Figure 6.6.

The first scenario is the reception of additional data by component B. In this case, high coverage of test cases of A will likely not be able to cover all functionality of B as it is also used by component C in some way. Thus, the usage of B by C leads to areas uncovered in B, which are not superfluous functionality, but false positives. To mitigate the false positives, the test cases can be extended with inputs to B thereby simulating C. In practice, this is done by adding all inputs of B not outputs of A as (symbolic) inputs to Z.

The second scenario is sending of additional data by component A to another component C. In this case, the approach does not need to be amended as high coverage test suites of A will also cover C in addition to B.

The third scenario is the sending of data by component A to another component C and the reception of data by component B from C. In this case, there are two ways of mitigation: (1) either the test cases are extended with the inputs to B as presented in the first scenario or the outputs of A to C are recorded for each test case in T_A, given

Figure 6.6: Scenarios of dependencies



to C as unit tests producing outputs for B. In practice, the latter can be implemented automatically if C only consists of components without outside system dependencies.

The fourth scenario is a loop back from B sending data back to A through another component C being the inverse scenario to the third scenario. If simplified by C applying the identity function for all inputs, this scenario constitutes a simple backward loop as is the case in control systems (see Figure 2.1). OUTFIT is unable to handle even such a simplified scenario as the functionality of each component cannot be considered separately. The only possible mitigation is to skip this particular integration and proceed to the next higher level of integration testing, thus turning the composition of A and B into a new component D and hiding the loop.

6.2.2 Detecting untested integrated exception/fault handling

To operationalize the second defect model of Section 6.1.2, we can re-use the above operationalization with three changes. Firstly, component A must propagate exceptions/faults to component B. A particular choice for A is the exception/fault handling units within each component, while B is the global exception/fault handler/watchdog. Typically these components can be extracted as the only output of a local fault handler (A) is an input to the global fault handler (B). Secondly, a high coverage test suite must only exist or be created for component A as only B is to be integration-tested. Thirdly, the actual output of the test cases must be groupable by the integer output signal of B representing the system mode (e.g. system operation/degradation / failure mode).

When executing the high coverage test suite for A in Z, (ideally) all respective exception/faults are triggered and propagated to B. B then switches the system's mode according to the severity of the exception/fault in A and test cases are grouped by this mode output. By manually inspecting test inputs and the resulting system mode, it can be assessed whether the system switched into the correct mode. Particularly interesting are critical test cases leading to modes such as system shutdown, but also no reaction to an exception/fault may be incorrect and endanger data or life.

6.3 Evaluation

Based on the failure models of superfluous or missing functionality and untested integrated exception/fault handling above, we implemented the OUTFIT tool following the above ideas and the methodology. We evaluate our tool using a real-world electrical engine control system of a hybrid car. Our evaluation systems are a quad-core Intel Core i7 920 with 4GB of RAM running Microsoft Windows and an octa-core Intel Xeon E5540 at 2.5 GHz with 40 GB of RAM running Ubuntu Linux. Matlab Simulink, Clang and the unit/integration testing framework of our project partner are running on the Windows machine while only KLEE [25] is running on the Linux machine. We use the computationally stronger machine for Linux as KLEE cannot be ported to Windows and is known to require a lot of computational power. Thus, the Windows machine

performs the tasks of transforming (parts of) the Matlab Simulink model to C and then to LLVM bitcode in the Model Transpilation step. Then the Linux machine runs KLEE on the created LLVM bitcode to derive test cases in the Test Case Creation Step. These test cases are sent back to the Windows machine an executed in the unit/integration testing framework of our project partner in the Test Case Execution step.

The evaluation's goal is to show (1) reproducibility, (2) effectiveness, and (3) efficiency of OUTFIT and, therefore, of the superfluous or missing functionality and untested integrated exception/fault handling failure models introduced in Sections 6.1.1 and 6.1.2.

Firstly, symbolic execution as implemented in KLEE [25] is non-deterministic as choosing a path may occur at random. Thus, reproducibility must be assessed to evaluate if multiple executions lead to the same result. In practice, every execution of OUTFIT should lead to comparable coverage values as varying coverages lead to a significant manual inspection overhead and decrease the effectiveness. This manual inspection overhead arises from parts of the system not covered by the test suite that are neither superfluous nor missing and only arise due to low overall coverage. Thus, reproducibility puts the results of effectiveness into perspective as unreliable reproducibility destroys any effectiveness.

Secondly, the effectiveness of OUTFIT must be evaluated w.r.t. potentially detected defects to assess the benefit of using OUTFIT. We perform a manual inspection and assess the uncovered parts of the model to detect any missing or superfluous parts and untested integrated exception/fault handling. We then examine whether the parts could be covered manually/trigger the correct fault handling or yield true defects.

Thirdly, the efficiency of OUTFIT needs to be evaluated w.r.t. the time consumed for one execution. The time consumed must be reasonable to run OUTFIT overnight for deployment in a continuous integration context. Thus, the execution time per integrated system on our standard hardware evaluation systems is not to exceed 8 hours for two components as defined reasonable by our industry partner.

The Matlab Simulink model used is an electrical engine control system deployed in a real hybrid car. The system model has over 31 000 blocks in 11 components using Matlab Simulink as implementation language only. The high level components deal with signal and bus inputs, engine control and signal and bus outputs as well as system monitoring and diagnostics. All high level components contain at least 3 levels of nested components and we selected 3 components to test. The three components selected for evaluation are in the input processing and engine control high level component. Their characteristics are shown in Table 6.1. ProcessCurrents and CalculateCurrents are responsible for calculating the current from analog input values in input processing and AdjustVoltage is an open loop controller for the voltage to the electrical engine in the engine control. When selecting the components, we wanted them to be representative and independent of other components. Since the components are from different high level components with different scopes, the

implemented functionality and complexity differs. AdjustVoltage is part of a controller while the two current calculation components are transformative with only a fault state. Each of the components is independent of other components and, therefore, is able to potentially reach high coverage by only using its inputs. In addition, ProcessCurrents is able to evaluate the superfluous or missing functionality and untested integrated exception/fault handling failure model as it contains fault handling.

Table 6.1: Components used in the evaluation

Name	blocks	Inputs	LOC	Subsystems
ProcessCurrents	486	28	472	A: Calculate present currents B: Process faults during calculation
CalculateCurrents	244	7	321	A: Normalize input values B: Replace implausible values
AdjustVoltage	199	39	259	A: Open loop voltage controller B: Saturate output voltage

For the evaluation, we were given test cases for component A of AdjustVoltage to reuse. These functional test cases yield less coverage (60% condition and 45% decision coverage in Matlab Simulink) than the generated test cases by KLEE (see Table 6.4). Thus, multiple functionalities are actually coverable and re-using the test cases only produced additional manual inspection effort. Based on these results, we will always generate test suites with KLEE in the following also due to the evaluation of the worst case execution time of OUTFIT. However, we recommend to always generate test suites with KLEE in order to avoid manual inspection overhead due to actually coverable functionality. Thus, coverage-based testing, which lacks a defect model (see Chapter 3), is injected with a defect model leading to defect-based testing when used in OUTFIT.

For all goals of the evaluation, we execute OUTFIT 10 times. OUTFIT is configured with the respective A and B for each of the three components. The only configuration of OUTFIT concerns the Test Case Creation step with KLEE (see Section 2.2 for the used command line options). We execute KLEE with this command line yielding a maximum of 20 minutes execution time and a signal (or trace) length of 6. These parameters were chosen as the results of the evaluation of 8Cage (see Section 5.3) using this maximum execution time and signal length were positive when running KLEE on Matlab Simulink models. In addition, the test cases created by KLEE for T_B may require a certain signal value in the first step. However, this signal value may not be producible by A in the first time step as calculations in A must be performed before it can be given as an output. Thus, for the creation of $T_{A \leftarrow B}$, we added a lag to the configuration, which shifts the signal required by the test cases T_B by a number of time steps allowing A time to produce this output. For the evaluation, we set the (possible) lag to 3 time steps. If KLEE is unable to create $T_{A \leftarrow B}$ directly using the signal values required by the test cases of T_B, this configures KLEE to neglect the signal values

produced by A within 1, 2 or 3 time steps. Thus, it only applies the output constraint for the signal values required by the test cases of T_B to A after 1, 2 or 3 time steps.

Table 6.2: Coverage of the integrated system ProcessCurrents

Execution	Component	No. test cases	Coverage (Model in %) CC	DC
1.	T_A	6	70 (55/78)	75 (49/65)
	$T_{A\leftarrow B}$ (T_B)	0 (4)	x	x
2.	T_A	5	71 (56/78)	75 (49/65)
	$T_{A\leftarrow B}$ (T_B)	1 (4)	50 (39/78)	56 (37/65)
3.	T_A	5	73 (57/78)	76 (50/65)
	$T_{A\leftarrow B}$ (T_B)	0 (4)	x	x
4.	T_A	5	69 (54/78)	73 (48/65)
	$T_{A\leftarrow B}$ (T_B)	0 (4)	x	x
5.	T_A	5	70 (55/78)	75 (49/65)
	$T_{A\leftarrow B}$ (T_B)	1 (4)	50 (39/78)	56 (37/65)
6.	T_A	5	73 (57/78)	73 (48/65)
	$T_{A\leftarrow B}$ (T_B)	0 (4)	x	x
7.	T_A	5	73 (57/78)	76 (50/65)
	$T_{A\leftarrow B}$ (T_B)	1 (4)	50 (39/78)	56 (37/65)
8.	T_A	5	71 (56/78)	75 (49/65)
	$T_{A\leftarrow B}$ (T_B)	1 (4)	50 (39/78)	56 (37/65)
9.	T_A	4	70 (55/78)	76 (50/65)
	$T_{A\leftarrow B}$ (T_B)	1 (4)	50 (39/78)	56 (37/65)
10.	T_A	5	73 (57/78)	73 (48/65)
	$T_{A\leftarrow B}$ (T_B)	0 (4)	x	x

6.3.1 Reproducibility

To evaluate the reproducibility of OUTFIT, we question the reproducibility of the results of KLEE w.r.t. the achieved unit coverage. KLEE uses non-deterministic features such as random path exploration among others. After 10 execution of OUTFIT, we assess the resulting coverage. Our baseline for reproducibility is less than 10% standard deviation in condition and decision coverage. These thresholds were chosen to (1) minimize the manual inspection effort according to the expectations of our project partner and (2) to be on the same level as previous results by Cadar et al. [25] using KLEE on the GNU COREUTILS without libraries. In the following, coverage is measured within Matlab Simulink after executing the integration test suites separately in Z (i.e. the integrated system of A and B). We do not execute T_A and $T_{A\leftarrow B}$ in sequence in Z as maximum coverage of Z is not required by our failure models. If a high coverage test suite is required, KLEE can be executed on Z directly.

Table 6.2 shows the number of test cases, condition and decision coverage for T_A and $T_{A\leftarrow B}$ in each execution of OUTFIT for the ProcessCurrents component. It also includes the number of conditions/decisions covered and the total number of

conditions/decisions in Z. The average condition coverage achieved by T_A is 72% while the average decision coverage is 74.7% with an average of 5 test cases. The respective standard deviations are 3% and 1%. The average condition coverage of $T_{A \leftarrow B}$ is 50% and decision coverage is 56% with an average of 1 test case. The standard deviations for decision coverage are 0%. Thus, we deem the results of ProcessCurrents within our thresholds and reproducible. Admittedly, coverage of 50% is rather low after performing the symbolic execution and will likely lead to a large manual inspection effort in some systems. However, this low coverage may also be the effect of code generation and compiler optimization. In cases with low coverage, a review / inspection of the component concerning the failure models could yield less effort. Also recall that, we are creating the test suite $T_{A \leftarrow B}$ based on T_B, which is not directly executable on the integrated system. Thus, it may occur that KLEE is unable to satisfy the output constraints created by the inputs of the test cases of T_B and generates no test cases. This is denoted as x in Table 6.2 including the number of generated test cases of T_B in parentheses.

Examining the results of CalculateCurrents in Table 6.3, the average condition coverage achieved by T_A is 93% while the average decision coverage is 95% with 5 test cases in all executions. The respective standard deviations are 6% and 0%. The average condition coverage of $T_{A \leftarrow B}$ is 50% and decision coverage is 68% with 1 test case in all executions. The standard deviations for decision coverage in CalculateCurrents are 0% and 3%. Thus, we deem the results of CalculateCurrents within our thresholds and reproducible. Admittedly, again coverage of 50% / 68% is rather low and will likely lead to a large manual inspection effort in other systems as was the case with ProcessCurrents. In addition, the low coverage when executing $T_{A \leftarrow B}$ was a potential defect as seen in Section 6.3.2.

Creating and executing T_A and $T_{A \leftarrow B}$ for AdjustVoltage as shown in Table 6.4, the average condition coverage achieved by T_A is 76% while the average decision coverage is 66% with 3 test cases in all executions. The respective standard deviations are 5%. The average condition coverage of $T_{A \leftarrow B}$ is 91% and decision coverage is 83% with 3.3 test cases on average. The standard deviations are 7% and 6% respectively. Thus, we deem the results of CalculateCurrents within our thresholds and reproducible.

We hence conclude the reproducibility of the results of OUTFIT to be high w.r.t. the standard deviation of the achieved coverage. However, this does not necessarily mean that the same parts are always covered and different parts may be covered in each run leading to more or less manual inspection effort. We speculate the large number of operations (i.e. blocks) and branchings in ProcessCurrents to influence the ability to derive the test suite $T_{A \leftarrow B}$ based on T_B and/or that the inputs of the test cases of T_B are infeasible to produce for component A. This is supported by the integrated KLEE statistics stating that it is spending 99% of the execution time in constraint solving. However, further investigation is required.

Table 6.3: Coverage of the integrated system CalculateCurrents

Execution	Component	No. test cases	Coverage (Model in %) CC	DC
1.	T_A	5	96 (27/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	65 (13/20)
2.	T_A	5	89 (25/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	65 (13/20)
3.	T_A	5	100 (28/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	70 (14/20)
4.	T_A	5	89 (25/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	65 (13/20)
5.	T_A	5	100 (28/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	70 (14/20)
6.	T_A	5	92 (26/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	65 (13/20)
7.	T_A	5	100 (28/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	70 (14/20)
8.	T_A	5	92 (26/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	70 (14/20)
9.	T_A	5	89 (25/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	70 (14/20)
10.	T_A	5	92 (26/28)	95 (19/20)
	$T_{A\leftarrow B}$ (T_B)	1 (1)	50 (14/28)	70 (14/20)

6.3.2 Effectiveness

For the evaluation of effectiveness of OUTFIT, we perform a manual inspection of uncovered parts to assess whether these parts are missing/superfluous functionality or untested integrated exception/fault handling. In case we find an uncovered part, we examine whether this part could be covered by manually creating test cases or is indeed a potential defect.

While performing the manual inspection, we found a potential defect in the CalculateCurrents component. CalculateCurrents performs a normalization of input values and a plausibility check on the normalized values. In case they are not plausible, they are replaced by default or other input values. After executing OUTFIT 10 times, it was never possible to cover the plausibility check completely when executing T_A. This means that although the generated T_A covers the normalization (A) of CalculateCurrents in 3 executions, these test cases are unable to achieve coverage of the plausibility check (B). Thus, there are three possible causes: (1) A is missing functionality, (2) B has superfluous functionality or (3) it is possible to completely cover A while not necessarily exercising all its functionality. All three causes were investigated as they may lead to failure or extended execution times in the safety-critical electric engine control system. In the end, it was possible to cover A while not necessarily exercising

Table 6.4: Coverage of the integrated system AdjustVoltage

Execution	Component	No. test cases	Coverage (Model in %) CC	DC
1.	T_A	3	80 (8/10)	70 (14/20)
	$T_{A \leftarrow B}$ (T_B)	3 (3)	90 (9/10)	90 (18/20)
2.	T_A	3	80 (8/10)	70 (14/20)
	$T_{A \leftarrow B}$ (T_B)	4 (4)	90 (9/10)	85 (17,20)
3.	T_A	3	70 (7/10)	60 (12/20)
	$T_{A \leftarrow B}$ (T_B)	4 (4)	100(10/10)	95 (19/20)
4.	T_A	3	70(7/10)	60 (12/20)
	$T_{A \leftarrow B}$ (T_B)	3 (3)	90 (9/10)	80 (16/20)
5.	T_A	3	80 (8/10)	70 (14/20)
	$T_{A \leftarrow B}$ (T_B)	3 (3)	80 (8/10)	75 (15/20)
6.	T_A	3	70 (7/10)	60 (12/20)
	$T_{A \leftarrow B}$ (T_B)	3 (3)	80 (8/10)	80 (16/20)
7.	T_A	3	80 (8/10)	70 (14/20)
	$T_{A \leftarrow B}$ (T_B)	4 (4)	100(10/10)	80 (16/20)
8.	T_A	3	80 (8/10)	70 (14/20)
	$T_{A \leftarrow B}$ (T_B)	3 (3)	90 (9/10)	80 (16/20)
9.	T_A	3	70 (7/10)	60 (12/20)
	$T_{A \leftarrow B}$ (T_B)	3 (3)	90 (9/10)	75 (15/20)
10.	T_A	3	80 (8/10)	70 (14/20)
	$T_{A \leftarrow B}$ (T_B)	5 (5)	100(10/10)	95 (19/20)

all functionality as part of the plausibility check in B needed not to be triggered when covering A.

For the untested integrated exception/fault handling failure model, the test cases were grouped by the output propagated to the central fault handler. We performed a manual inspection of the inputs leading to each propagated fault and could not find any discrepancies w.r.t. the specification. A discrepancy would have been (1) the propagation of a major system fault although only a minor fault occurred or (2) the omission of propagation of a major system fault. Examples for these discrepancies are the short-time loss of an outside temperature sensor value or a significant loss of voltage from the high voltage battery.

While we investigated the three integrated systems and their respective coverage, we expected many false positives as coverages vary between 50% and 96%. However, we found the manual inspection to be almost effortless in our case study as the graphical nature of Matlab Simulink lead to a quick reasoning about uncovered functionality. Most of the time, the uncovered functionality was related to configuration parameters leading to a usage of one part of the component to perform the computation or the other. The only system difficult to investigate was ProcessCurrents as the coverage achieved by $T_{A \leftarrow B}$ for B was incomplete, because KLEE was unable to provide test cases.

Although we have only found one potential defect, we conclude OUTFIT to be effective in our case study. The system we used for evaluation was already completely unit, integration and system tested as well as deployed hinting at a small number of defects left. Still, we were able to find and localize a potential defect in every execution leading to possible untested functionality.

6.3.3 Efficiency

We evaluate the efficiency of OUTFIT by calculating the average execution time and its standard deviation of 10 executions on all systems of the evaluation as shown in Table 6.5. Our baseline is the possibility to execute OUTFIT within 8 hours overnight.

ProcessCurrents is the most complex component due to the number of operations performed and also takes the longest on average with more than 90 minutes. The standard deviation is less than 10% for the execution time with a worst case execution time of 107 minutes. This execution time is due to reaching the maximum execution time of 20 minutes for KLEE in the creation of T_A, T_B and $T_{A \leftarrow B}$ including all possible lags. For CalculateCurrents and AdjustVoltage, the average execution time is just below 20 minutes with a standard deviation of 1 - 2 minutes. These systems never reached the maximum execution time of KLEE as KLEE was able to create T_A, T_B and $T_{A \leftarrow B}$ within 10 minutes. The execution time of the Test Case Creation step is also the only varying factor as the execution time for Model Transpilation and Test Case Execution stay stable within 45 second bound. The execution times of Model Transpilation and Test Case Execution are due to Matlab Simulink and Clang in Model Transpilation and the unit/integration testing framework of our industry partner in Test Case Execution.

We hence conclude the execution times of OUTFIT to be stable for all evaluated systems and execution within 8 hours to be possible by far. There is an increase in execution time when using components with more operations attributed to KLEE in the Test Case Creation step. This increase allows only 5 components of the complexity of ProcessCurrents to be analyzed in an 8 hour overnight period on one system equivalent to the evaluation system. Note that, ProcessCurrents is part of a high level component and represents (almost) maximum complexity. Also, this is a worst case execution time of the most complex system in our evaluation. We have chosen components consisting of multiple integrated units as A and B in our three evaluated components

Table 6.5: OUTFIT execution time for the evaluated components (step numbers as in Section 6.2

Name	1.* (s)	2.* (s)	3.* (s)	Total (s)	Total (min)	σ (min)
ProcessCurrents	264.8	5011.5	380.8	5657.1	94.3	9.0
CalculateCurrents	219.6	520.3	378.3	1118.2	18.6	1.6
AdjustVoltage	245.0	527.0	356.3	1128.3	18.8	0.9

*Number corresponds to consecutive step as per section in Section 6.2

and hypothesize (1) the times to reduce when using units as A and B and (2) the time to increase when using high level components.

6.3.4 Summary

In summary, we find the results of OUTFIT to be reproducible, effective and efficiently provided for the missing/superfluous functionality failure model in our case study. Although the creation of $T_{A \leftarrow B}$ from T_B is hard and sometimes infeasible, the overall performance of the test case creation is reproducible and effective also leading us to a potential defect of the evaluated real-world system components. The untested integrated exception/fault handling failure model needs to be further evaluated w.r.t. effectiveness as we were unable to find any defects in our evaluation. One issue we found was the conversion from LLVM bitcode to model coverage. In some cases KLEE reported a high coverage of the bitcode, which was not reflected in the coverage of the model. We hypothesize the reason for this discrepancy to be the transformation from model to C code and the compilation to bitcode, which applies optimization possibly decreasing the number of conditions and decisions to cover. It is well-known that symbolic execution suffers from scalability issues [27]. Since our approach uses symbolic execution, it also suffers from scalability issues. Thus, our results may not scale to larger, more-complex components. The results of CalculateCurrents already hint towards scalability issues due to complexity. Therefore, a larger evaluation with more than three extracted systems is required to examine the issues above further and add to generalizability of the tentative results of our case study.

Although test case generation and execution is automatic, there is no oracle for the test cases. Thus, there is an effort involved in manually inspecting the respective coverage. For the components in the evaluation, this effort was minimal. Firstly, only a small part of the plausibility check (B) of CalculateCurrents was not covered. Secondly, coloring of coverage is automatically added in Matlab Simulink making the uncovered parts clearly visible. Thirdly, scaling to the complete component on our 1080p screen still yielded readable and reviewable results. However, to gain generalizable results concerning the manual inspection effort, further (possibly large-scale case study) research is required. Such research is also able to yield results w.r.t. comparability to a completely manual approach.

6.4 Related Work

Using unit test cases with MC/DC coverage for integration testing has already been studied in previous work [127]. However, this present work proposes an explicitly defect-based way to perform integration testing based on coverage tests cases with particular defects and described defect models.

Apart from the well-known bottom-up and top-down integration testing approaches [17, 123], previous work discusses integration test selection criteria and integration testing for object-oriented as well as component-based approaches.

In the area of integration test selection criteria, Harrold and Soffa [65] extend structural test selection approaches to interprocedural testing by analyzing call dependencies. Jin and Offut [80] use coupling as a basis for coupling-based testing and define coupling-based criteria based on data-flow test selection criteria. Le Traon et al. [95] present a model-based integration test approach for planning integration and regression tests. For the assessment of test selection criteria, Delamaro et al. [43] present an approach based on mutation testing of the interfaces.

For the integration testing of object-oriented software, Bashir and Paul [11] discuss the processes involved in integration testing of object oriented software. Jorgenson and Erickson [82] categorize the testing approaches into unit, integration and system testing and add a graph-oriented notation.

In the area of integration testing of component-based software, Ipate and Holcombe [78] propose using X-machines for the formal verification of correct integration of components. Similarly, Cristiá et al. [39] propose integration testing based on a specification in Z whereas Kandl and Elshuber [86] describe a formal approach based on SystemC. Elsafi [51] proposes to infer component behavior for integration testing in case component information and/or source code is not available to test against the inferred model.

The previous work above focuses on creating integration test cases targeting functional defects in the interaction of components partially in a program/domain/-paradigm specific manner. Particularly, the test selection criteria based on coverage and coupling allow automatic creation of test cases while object-oriented and component-based approaches require a model/specification for test case creation/derivation. OUTFIT fits into the first criteria of test selection that allow automatic derivation. However, it focuses on the (semi-)automatic creation of defect-based integration test cases targeting a particular defect in the system. Obviously, the targeted defects are part of the functionality, but are independent of any explicit specification and rely on manual inspection of the results. Other approaches may also inadvertently detect the described defects, but do not create test cases as operationalization of their defect model to directly target them. In addition, OUTFIT is also able to create test environments automatically leading to the possibility of testing any integrated systems. Thus, it is program and domain independent, but limited to the implementation paradigm used in Matlab Simulink. However, the described failure models are paradigm independent with the abstraction of function calls and as long as there is a form of exception handling. Particularly, they can be operationalized in the object-oriented paradigm, which is left as future work.

In practice, there are several tools focusing on the functional software integration testing of embedded system including TPT [146], Reactis [131] and slUnit [19]. These

tools allow the creation of manual (model-based) test cases or are able to create test cases using random or coverage-based criteria and their execution. Using the framework of OUTFIT, these tools can also be used for the derivation of high coverage test suites instead of KLEE.

6.5 Conclusion

In this chapter, we describe an approach for integration testing based on defect models for defect-based quality assurance. This deviates from earlier approaches, where test cases are created for the verification of requirements whereas our approach specifically targets common and recurring defects detectable during integration testing. We describe two failure models based on the common and recurring defects targeting superfluous or missing functionality and untested exception/fault handling. These failure models are the basis for our operationalization OUTFIT for Matlab Simulink. OUTFIT re-uses high coverage unit tests or creates them automatically using symbolic execution (e.g., KLEE [25]) in a three step process. The only manual step is the inspection of the coverage results yielding verdict and fault localization. Thus, OUTFIT is (semi-)automatic.

We evaluate OUTFIT using three components of a real-world electrical engine control system of a hybrid car. These components are representative of engine control systems and contain between 199 and 486 operations. OUTFIT was always able to produce test cases with at least 50% coverage and at most 10% standard deviation, but found a correlation between the number of operations and the decrease in coverage/increase in standard deviation. We even found a potential defect in the real-world model, which could have been superfluous or missing functionality. It turned out to be an indication towards the ability of a test suite to have coverage, but not exercise all functionality. Overall, the results of OUTFIT were reproducible, effective and efficiently procured and fulfilled the prerequisites given by our project partner for practical application.

Although the evaluated systems are representative for engine control in the automotive domain, further evaluation of OUTFIT on Matlab Simulink systems from other domains such as medical and avionic systems is required. These systems also have real-time constraints and are safety-critical similarly to the evaluated engine control system. Superfluous functionality in components leads to longer execution times possibly not meeting real-time constraints while untested integrated fault handling can lead to unsafe behavior. Thus, we project our presented failure models to also be able to detect defects concerning the real-time and safety-critical behavior.

In the future, further operationalizations based on the described defect models could be created. Particularly focusing on explicitly testing the exception handling in object-oriented IT systems as a common and recurring defect (#5 in the top 14 technical defects of Table 4.2) could be beneficial. However, preliminary results have

shown that particularly symbolic execution for object-oriented systems is harder than for procedural languages such as C. In addition, more studies to elicit and classify further defect models operationalizable in the area of integration testing are required. Particularly interesting are defects attributed to the environment of the system as these are hard to detect by path exploration.

To extend OUTFIT further, other techniques to create high coverage test suites such as the ones presented by Perandam et al. [126] and Matinnejad et al. [104] could be used. The test suites may also be generated by using fuzz [21, 56] or random testing [4, 6]. These techniques are also particularly interesting for the operationalization of other defect models.

7

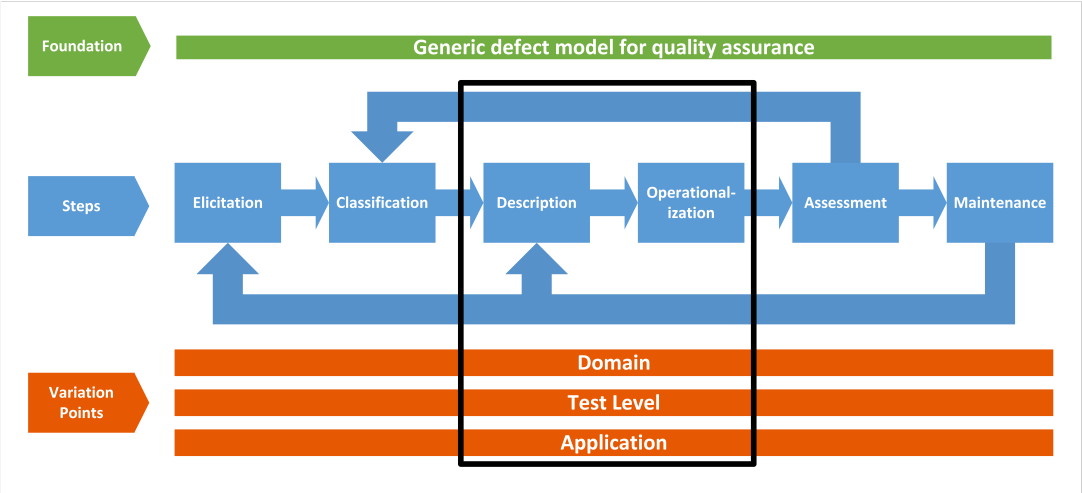
Description and Operationalization: Controller Tester for system testing

The previous chapter introduced the second explicit description and operationalization OUTFIT on integration testing level based on the generic defect model in a systematic and (semi-)automatic approach to defect-based quality assurance. To completely demonstrate the effectiveness and efficiency of the operationalization of defect models on all levels of testing (unit, integration and system testing), this chapter describes and operationalizes defect models on the system testing level in a tool called Controller Tester. Controller Tester performs (semi-)automatic testing of control systems developed in the domain of Matlab Simulink systems. Thus, it is application specific to control system. This chapter is the third of three instances of the method application step within the mentioned variation points in the lifecycle framework as seen in Figure 7.1.

When testing continuous control systems, the selection of good test cases is essential as exhaustive testing of the operating range is infeasible. Control system engineers tend to use well-established methods to provoke *failures* typically based on manual formal analysis of the continuous control system under test. These methods aim to provoke failures as a violation of specific quality criteria such as stability, liveness, smoothness, responsiveness. The quality criteria have been formalized to evaluate the results of automatic test selection methods based on one typical value-response scenario [105, 106].

We create a comprehensive library of failure models and quality criteria for the automated testing of continuous control systems. Our failure models and quality criteria are based on (1) existing work of Matinnejad et al. [105, 106] and (2) a practitioner survey and literature review. Based on the feedback of 7 control system experts having at least 2 years practical industry experience, we gained two quality

Figure 7.1: Position of the operationalization Controller Tester in this thesis



criteria and 4 failure models. We could identify these quality criteria and failure models also in existing literature. The additional quality criteria measure the extent of steady-state oscillation for steadily oscillating control systems (steadiness) and the retention of the control system within its operational range (reliability). The failure models relate to sinusoidal and disturbance response as well as the comparison of control systems and dead zone violation (allowed oscillation). These libraries increase the variety of behaviors automatically testable and the detail of assessment.

We operationalize the failure models in an automated testing tool and are interested in how useful it is. From a practitioner's perspective, the test generator should be *effective* (it finds the relevant tests that potentially violate the quality requirements). It should also be *efficient* in that the time to compute test cases for common scenarios happens in the order of minutes rather than hours. Moreover, we would like to understand what the gain in effectiveness is if we add resources to the test case generator. Given that test case generation relies on random testing, results should be *reproducible*: Running the system twice should yield test cases of similar quality.

In sum, continuous control systems have a broad operating range spawning a large multidimensional input space of their signals. This range makes exhaustive tests infeasible and calls for cost-effective test case derivation: testing controllers is expensive due to their test case duration. We want to derive "good" test cases "efficiently" and be sure that respective test case generators are "reproducible," and that the deployment of resources can rationally be justified.

By negating domain-independent quality criteria, and by imposing additional practically accepted constraints on the input domain (called scenarios), we present failure models which characterize those parts of a controller's input domain that are likely to violate the quality criteria. We modify the ideas behind existing test case generators to get a generic tool for failure-based testing of continuous controllers.

We show (1) that existing work [105, 106] is an instance of our schema, and therefore generalize these results. On the grounds of interviews with domain experts, we (2) present failure models and quality criteria that have not been described as such in the literature yet. To the best of our knowledge, the combination of (1) and (2) is the first creation of an extensive and comprehensive library of failure models and quality criteria for control systems. Finally, on the grounds of several experiments, we (3) provide evidence that existing work, and specifically our generalization, scales; is effective; efficient; reproducible; and predictable: thus yielding a solution that is a candidate for deployment in industrial practice.

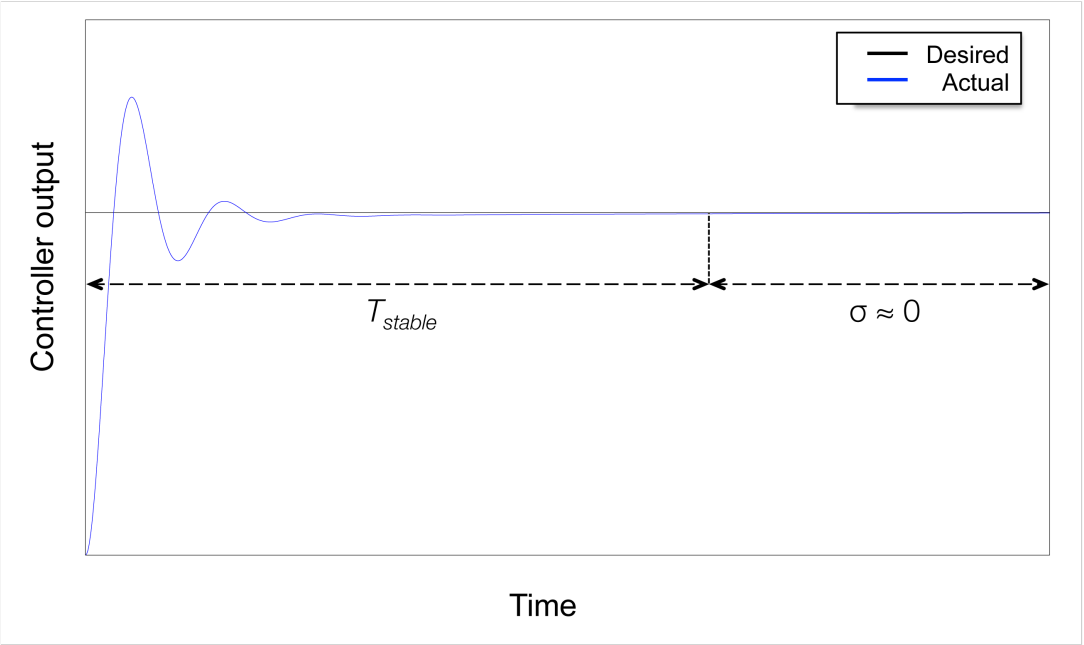
7.1 Search-based control system testing

Recalling the formal manual approach in Section 2.3, an automated search-based approach for testing continuous controllers, that our work fundamentally builds upon, was introduced by Matinnejad et al. [105]. It uses a search-based strategy in MiL testing in Matlab Simulink by performing a search space exploration, followed by a refining single-state search. The aim is to find the worst case control system response by driving the system towards a certain goal with the required quality. Their approach uses a scenario of finding two distinct desired values such that the controlled process's response violates the quality requirements. To this end, they introduce an extended step function using two desired values to test the reference-value response. In a first phase, the search space is explored by partitioning this two-dimensional input space into regions and randomly selecting a number of points within each region as test inputs. This random selection can either be purely random, or adaptively random which tries to avoid clusters of points by maximizing the distance between the points in each region [106]. By using four quality criteria for the control system, parts of the trajectory of actual value in each test case are evaluated. The results are visually presented as a heat map to a control system engineer, who will have to choose regions to further investigate for the worst-case. In the second phase, a subsequent single-state search, an optimization algorithm is used to find the global maximum of an objective function in each selected region yielding (an approximation of) the worst case control system response. This again is implemented using various techniques, including forms of hill climbing and simulated annealing.

7.2 Quality Criteria

The quality criteria described by Matinnejad et al. [105, 106] are stability, liveness, smoothness and responsiveness. The quality criteria represent abstractions of expected outputs usable by control system engineers as oracles to form a verdict whether the tested control system conforms to its quality requirements or when failure is present. Since the work of Matinnejad et al. presents an instance of our generalization negating these quality criteria, we directly re-use them. In addition, two further criteria required

Figure 7.2: The stability quality criterion



for negation by our newly described failure models exist. The two additional criteria are steadiness [50] and reliability [59]. When presented to seven control system experts of our project partner, they concluded the six quality criteria to be representative, adequate and minimal. While this is not representative, it gives us some confidence that we did not miss out on any relevant criteria.

Stability

Matinnejad et al. [106] introduce the standard deviation σ of the process's controlled variable as a measurement for stability (see Figure 7.2). σ is measured after T_{stable} and should be $\sigma \approx 0$ for a stable real-world controller. However, there are steadily oscillating control systems (see steadiness below) due to an unpreventable steady oscillation of the process. Such systems will always lead to a standard deviation larger than 0.

Liveness

To compute the steady-state error of a control system, liveness [106] measures the maximum difference between desired and actual values after T_{stable} (see Figure 7.3). Maximum liveness is reached when this measure is approximately 0. For a steadily oscillating control system (see steadiness below), the best liveness will be the amplitude of the steady-state oscillation.

Figure 7.3: The liveness quality criterion

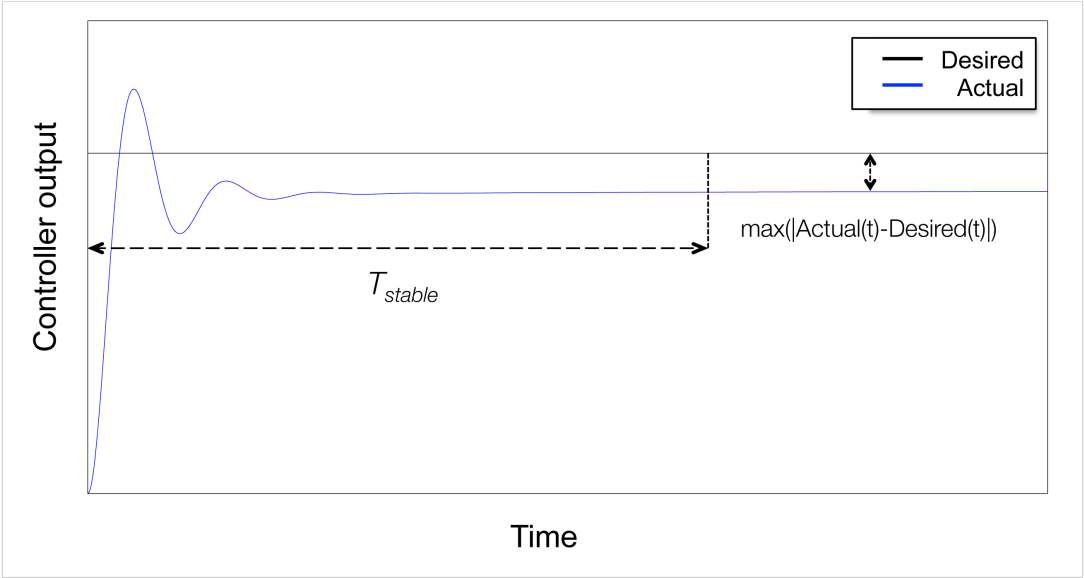
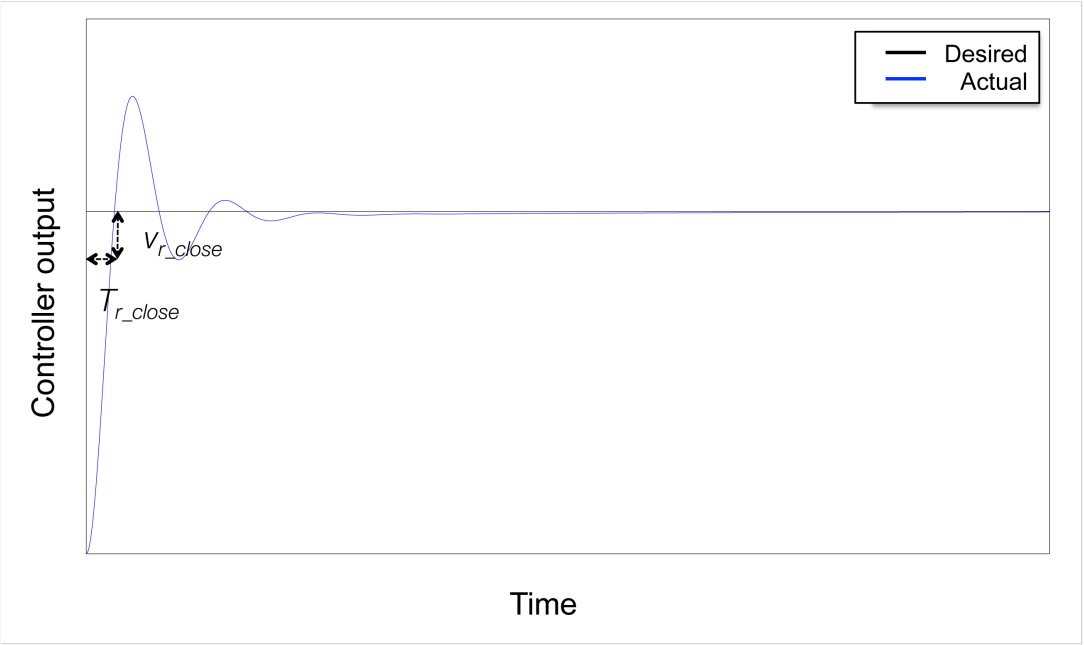


Figure 7.4: The smoothness quality criterion

Smoothness

To measure over- or undershoot, we utilize smoothness [105]. This is because it is particularly useful in our situation, where we use two (differing) intended values of the control variable (the step failure model, see Section 7.3.1). Smoothness is measured as the maximum absolute difference between the desired and actual values once the actual value gets as close as v_{s_close} to the desired value (see Figure 7.4). Smoothness should be as low as possible and within a range specified by the requirements.

Figure 7.5: The responsiveness quality criterion



Responsiveness

To measure response time, we use responsiveness [105], which once again is particularly useful in the step model (Section 7.3.1). It measures the time it takes the actual value to get as close as v_{r_close} to the desired value (see Figure 7.5). In a control system, responsiveness should be as low as possible and within a range specified by the requirements.

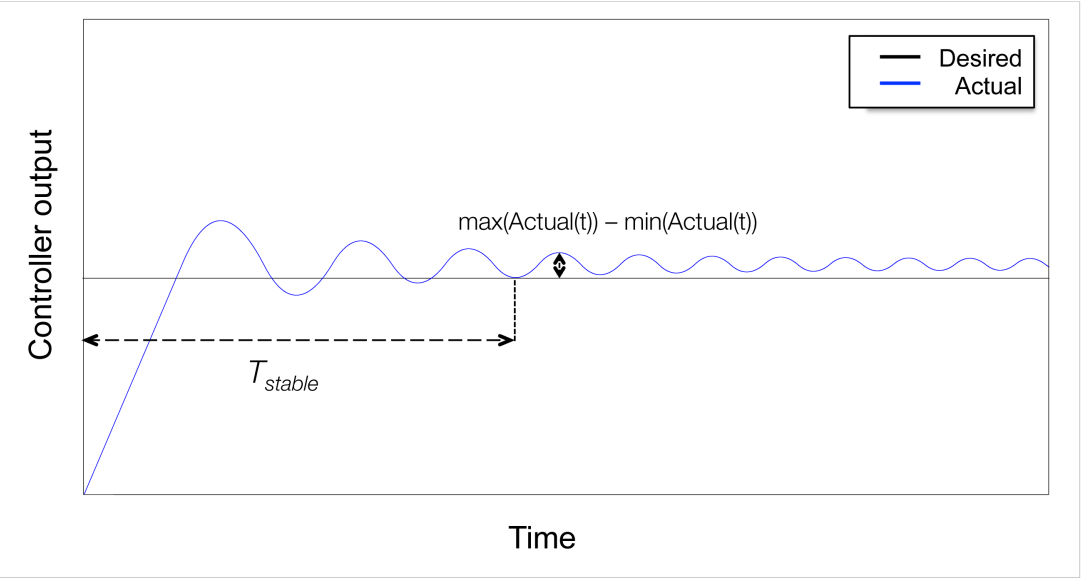
Steadiness

Steadily oscillating control systems are common when using multiple controllers in a cascade, relying on imprecise measurements or controlling processes with large internal disturbances. For a system steadily oscillating around the desired value as center of oscillation, the steady-state error provided by the liveness quality criterion provides the amplitude of the steady oscillation. In case the center of oscillation is different from the desired value, Ellis [50] proposes to measure the amplitude of the oscillation (see Figure 7.6). Calculating the maximum deviation of the actual value after some defined T_{stable} is to result in a value twice the amplitude for a stable system. In case this measurement exceeds the expected amplitude, system bounds may have been violated.

Reliability

Each control system has a defined operating range. In case the actual value leaves this operating range, there may be physical damages to the system or its surroundings. This

Figure 7.6: The steadiness quality criterion



has far reaching implications concerning safety or other requirements of the control system. The quality criterion is presented by Goodwin [59]; its Boolean value checks the actual value signal to only contain values within the operating range. Reliability should always be given for any control system. If signals such as the control variable or internal signals of the controller are available for measurement, further Boolean checks can be defined accordingly to perform additional bounds checks.

In short, the quality criteria represent abstraction of expected outputs usable by control system engineers as oracles during a manual inspection of the results to form a verdict whether the tested control system conforms to its quality requirements or a failure is present. By gathering control system expert feedback and performing a literature survey, we are able to contribute a rationale as to how these quality criteria relate to quality requirements of control systems and ascertain their common use in practice. In addition, all quality criteria are defined such that a greater value is more likely to represent a failure. Thus, the next step is to find input signals/trajectories to the system that are likely to yield large values w.r.t. the quality criteria.

7.3 Failure Models

When testing control system, engineers provide desired and/or disturbance values over time to the control system, in order to contrast the system's actual response with some intended response (that is usually left implicit and exists in the minds of the test engineer). Specific choices of the used desired and disturbance signals can provoke failures. Failures in our context are violations of the quality requirements judged by a control system engineer using the quality criteria introduced in Section 7.2. Since for all criteria higher values mean a possible violation of quality requirements, we aim at

finding input values that maximize these quality criteria. A failure is the transgression of a specified upper bound. For technical reasons, the search space for input values needs to be restricted. This is why we add characterizations of those input values (i.e. scenarios) that are likely to yield high quality criteria values, and thereby may break the system requirements. Failure models provide these input characterizations together with an assertion when quality requirements are violated. In the following, we present several such failure models and show how to operationalize them for test case generation.

An input signal is a discretized sequence of values given to the control system over time. Formally, let $\tau = \{0, \ldots, T\}$ be a time series and $min_{desired}$ and $max_{desired}$ be the minimum and maximum desired values of the control systems, respectively. An input signal then maps each time step to a value in the operating range of the control system: $Desired : \tau \rightarrow \left[min_{desired}, \ldots, max_{desired}\right]$ ($r(t)$ in Figure 2.1). Correspondingly, the discretized output signal or actual value of the control system can be defined as $Actual : \tau \rightarrow \left[min_{actual}, \ldots, max_{actual}\right]$ ($y(t)$ in Figure 2.1). Note that the range of actual values may be larger than the operating range of the control system. This manifests an out-of-bounds failure as described in Section 7.2. Also note that we restrict ourselves to one controlled variable, but generalization is possible.

In the following, we present failure models for testing control systems. We concentrate on the characterization of the potentially failure causing inputs; the failures themselves always consist of violations of specified quality requirements.

The step failure model was directly derived from Matinnejad et al. [105, 106]. The others are the result of a literature [50, 59, 105, 106, 120] survey and discussions with seven of our project partner's control system experts. The additional failure models extend the library of failure models with typical test scenarios for continuous control systems used in literature and practice including sinusoidal and disturbance behavior. For each presented failure model, we not only provide the input signals for testing, but also a rationale as to why and when it is sensible to use. Remember that a failure model characterizes the failure domain φ (see Section 3.1.5). In the case of control systems, the characterization using input values (intended values for the controlled variable and disturbance values in this case) is infeasible since control system behavior is complex and possibly non-linear. However, there are not only input values, but desired and disturbance trajectories and environment conditions to take into account. Thus, the approximation $\tilde{\varphi}$ explicitly puts constraints on the input domain of either trajectory (e.g., "two intended values for the controlled variables are sufficient") and fixes the environment conditions. Every failure model constrains the space of trajectories of the desired value and disturbance signals creating a scenario. The search space exploration is then performed on this constrained typically two-dimensional search space. The surveyed control system experts concurred to specifically use boundary values as they are commonly examined. Thus, the limit testing failure model

(see Section 2.1.2 and Section 3.2.10) is combined with each presented failure model, always leading to the creation of additional, non-random, limit test cases.

7.3.1 Step

Goodwin et al. [59] and Ellis [50] describe the step response of continuous control systems as the fundamental desired value signal to judge reference-value response. It uses an underlying step function defined as a partial function over discrete time. Matinnejad et al. [105] introduce a two-dimensional step function using *two* desired values. The initial desired value $InitialDesired \in [min_{desired}, \ldots, max_{desired}]$ is given as the desired value for the first half of the total simulation time T. The final desired value $FinalDesired \in [min_{desired}, \ldots, max_{desired}]$ is established for the second half. Thus, the desired value signal of this failure model is defined as $Desired(t) = \begin{cases} InitialDesired & \text{if } 0 \leq x \leq \frac{T}{2} \\ FinalDesired & \text{if } \frac{T}{2} < x \leq T \end{cases}$. In a closed-loop control system, it is impossible to start from any other value than the initial value (e.g. 0) due to the feedback loop. Any test case using a single desired value tests only a step originating from the initial value. However, in real-world systems a step from any desired value to any other may occur and should therefore be tested. For instance, an air conditioning system can be set from 19 degrees Celsius to 17 degrees Celsius. Note that, if the initial value is 0, only using one desired value leads to solely performing tests using a positive step. However, failures are likely occur when performing a negative step according to the experts' feedback. Therefore, the rationale of this failure model is to start at an arbitrary initial value by setting it as $InitialDesired$. All quality criteria are then measured for the actual value signal when $FinalDesired$ is set as desired value. To steer the system into violating its quality criteria requirements, this failure model constrains the trajectories of the desired value signal to all possible two-dimensional steps. The disturbance signal is constrained to a constant 0 as only reference-value response is to be tested.

7.3.2 Sine

In a closed-loop continuous control system, a sine wave of $Desired(t)$ may result in a sine wave of $Actual(t)$. If the sine wave of $Actual(t)$ happens to amplify the sine wave of $Desired(t)$, the control system may become unstable. For control system engineers, it is utterly important to verify frequencies of instability to judge whether they are in the operating range. Thus, "frequency responses are a very useful tool for all aspects of analysis, synthesis and design of controllers and filters" [59]. The sine failure model is explicitly designed to show the frequency response for the reference-value response. The signal of the desired value is $Desired(t) = desired + a * \sin(2\pi ft)$, where $desired \in [min_{desired}, \ldots, max_{desired}]$ is the absolute term, f is the frequency and a the amplitude of the sine wave. The h-dimensionality of this failure model can be reduced to two dimensions as the amplitude only influences the time until the escalation of

instability, but not the instability itself. $desired$ is required because signal clipping in boundary regions may have different effects on stability. Thus, $desired$ and f form a two-dimensional search space. To specifically violate the quality criterion of stability and steadiness, this failure model constrains the trajectories of the desired value signal to all possible two-dimensional sine waves. The disturbance signal is again constraint to 0 because only reference-variable response is to be tested.

7.3.3 Disturbance

To test the disturbance response introduced in Section 2.3, we define a disturbance failure model. The rationale behind using closed-loop control systems is to react to disturbances on the process by having a feedback loop. The goal is to test whether the controller can recover from the disturbance as specified in its requirements. In particular, the over- or undershoot produced by engaging/removing the disturbance as well as the overall stability and liveness are of interest. Using the disturbance failure model, the disturbance is introduced in the feedback loop as $Disturbance : \tau \rightarrow disturbance$ ($d(t)$ in Figure 2.1), enabling testing of disturbances in the process and measurement errors at the same time. As we want to find the worst case reaction of a control system to a disturbance, we always use the maximum value $disturbance$ and duration T_{dist} of the disturbance according to the requirements. Variable factors are only the signal's start time t_{dist} and desired value $desired$. Once again, we can reduce the three dimensionality to two dimensions by fixing the signal to the same trapezoidal ramp, pulse or sine wave for all tests. The desired value signal is $Desired(t) = desired$.

The disturbance signal is $Disturbance(t) = \begin{cases} 0 & \text{if } t < t_{dist} \\ disturbance & \text{if } t_{dist} \leq t \leq t_{dist} + T_{dist} \\ 0 & \text{if } t_{dist} + T_{dist} < t \leq T \end{cases}$

for a pulse signal. The other signal types of $Disturbance(t)$ can be defined analogously. In this failure model, all quality criteria are measured after t_{dist} and $t_{dist} + T_{dist}$ to assess the control system behavior during and after the occurrence of the disturbance. As $Desired(t)$ is a constant trajectory and $Disturbance(t)$ is different from zero only after a start time t_{dist} as well as for a fixed duration of T_{dist}, the search space is again two-dimensional. Thus, this failure model aims at steering the system into violating all its quality criteria requirements by constraining the trajectories of the disturbance signal to having the specified form, value and duration. The trajectories of the desired value signal are constrained to all constant trajectories. Note that, the constraining of the desired value tries to separate reference-value and disturbance behavior. However, since the superposition principle does not hold for non-linear systems, a clear distinction may not be possible.

7.3.4 Comparison

A failure model derived in interviews with control system experts is the comparison failure model. When developing control systems, they are commonly designed using

a continuous time and value space and are later discretized in both dimensions. By performing a discretization, controller behavior previously assumed conforming to the requirements may now be violating them. Thus, it is not sufficient to only re-use continuous control system test cases as "one cannot simply treat digital control as if it were exactly the same as continuous control" [59]. For the purpose of comparing the response of two control system, we introduce the comparison failure model. The failure model is an extension of the step failure model where $Desired(t)$ is given to both control systems. Both $Actual(t)$ are assessed using the presented quality criteria. We then compute the difference in quality between the two controllers under comparison, in order to point out differences in response. Thus, this failure model aims at steering the two control systems into producing different responses for the same desired value signal. The constraints are the same as in the step failure model.

7.3.5 Allowed Oscillation

A further failure model derived given by the control system experts is the allowed oscillation failure model. Controllers used with steadily oscillating processes often have a deadband requirement. Within this deadband, the controller is supposed to show no reaction to changes in difference of desired or actual value. Typically, the deadband is specified by a minimal percentage $db_{[\%]}$ of change in difference to which the controller is to react. Any changes below this threshold should yield no reaction by the controller. The allowed oscillation failure model tries to provoke a reaction by using a step to just below $db_{[\%]}$ and just above $db_{[\%]}$. Thus, the desired value signal space is constrained into two blocks. One contains all step input signals just inside the deadband for a desired value; and another one contains all step input signals just outside of the deadband. This failure model does not require any quality criteria to be measured, but includes the binary oracle of whether the controller reacted or not. In case the controller reacts to a difference just below $db_{[\%]}$ or does not react to a difference just above $db_{[\%]}$, the verdict of the test case is fail; otherwise pass. $Disturbance(t) = 0$ is the constraint for the disturbance signal.

7.3.6 Summary

In sum, our failure models first present reasonable constraints on the input domain (input models for desired values and disturbances as a step, sine, ramp etc. function). Then, they describe what constitutes a failure by either directly relating to quality criteria and respective allowed maximum values, or, in the case of the comparison failure model, to the difference in quality between two controllers.

7.4 Evaluation

Based on the above failure models, we developed an automated testing operationalization called ControllerTester for Matlab Simulink, following the ideas and method-

ology in [105]. Because we add the explicit perspective of failure-based testing, we contribute by generalizing that cited approach (a demo video, open-source version and installer of the tool are available at `https://www22.in.tum.de/en/tools/controller-tester/`).

We evaluate the tool using a two Intel Xeon E5-2687W v2 processor machine having 16 logical cores clocked at 3.4 Ghz and 128 GB of memory. Matlab is running on Windows 8.1 in version 2014a. Because one controller requires 32-bit, we use a 32-bit version of Matlab in all experiments. The evaluation's goal is to show (1) reproducibility, (2) effectiveness/efficiency and (3) sensitivity of the methodology. Firstly, the search space exploration is non-deterministic as it is based on random selections. Thus, a reproducibility evaluation helps assess whether multiple executions lead to similar results—control system engineers need to be sure that independently of a chosen random seed, the tool is likely to yield similar results. Secondly, we want to better understand the relative and absolute effectiveness of the methodology. In terms of relative effectiveness, we compare the worst cases found by different search strategies. In absolute terms, we want to find out if our tool comes close to, or surpasses, the results independently obtained by three control system experts familiar with the systems. In a third step, we want to check for the sensitivity of the methodology by using different configurations of regions and points per regions. In particular, we want to know whether increasing points or regions in the search space exploration (Section 7.1) yields better reproducibility and higher relative effectiveness. Hence, we configure controller tester to use 8x8 regions, 10x10 regions or 12x12 regions with 8, 10, 12 points in each region configuration and at least 10 repetitions for each control system in our experiments to assess sensitivity. Our baseline configuration is 10x10 regions with 10 points each to assess reproducibility and effectiveness. We do not evaluate the second (optional) step of the search ("single-state search") because its effectiveness has already been thoroughly examined and distributions for its reproducibility have been shown [106]. Thus, we contribute a thorough evaluation of the first step of the search ("search space exploration").

We focus the evaluation on the step, sine and disturbance failure models. As the comparison and allowed oscillation failure models re-use the step failure model, the results of the step failure model apply to them as well.

Our control systems for evaluation are six control systems used for training (numbered 1-6), one real-world complex control system of a brushless motor (numbered 7), and one real-world control system of a pump (with feed-forward enabled and disabled, numbered 8 and 9) given to us by an industry partner. Each control system has an input range either specified in rpm or rad/sec as the desired rotational speed of a motor, in cm of a pin to drive out, or in m^3/h as the flow rate. Each control system also has different requirements w.r.t. maximum response time (liveness), maximum over-/undershoot (smoothness) and steady oscillation (steadiness). These are described in Table 7.1. The training systems are excerpts of real-world systems in a marginally

Table 7.1: Models of control systems used in our evaluation

No.	Input Range	T*	Max. resp. time	Over-/Under-shoot (max)	Steady oscilla-tion	Critical fre-quency	Max. Dis-tur-bance
1	0 - 6150 rpm	0.85s	0.1s	5%	no	1 khz	0.05
2	0 - 6150 rpm	0.16s	0.02s	10%	no	-	0.05
3	0.02 - 0.1m	2.3s	2s	5%	no	-	0.05
4	0 - 6150 rpm	0.62s	0.1s	5%	no	1.1 khz	0.05
5	0.02 - 0.1m	2.3s	2s	5%	yes	-	0.05
6	0 - 1 m	1.16s	11s	5%	yes	-	0.05
7	.5 - 4 k rad/s	10.26s	1s	5%	yes	-	0.2
8	2 - 7 m 3 /h	20s	10s	35%	no	-	-
9	2 - 7 m 3 /h	20s	10s	35%	no	-	-

*Total simulation time (T) as described in Section 7.3

simplified form to train new engineers. The controller of the brushless motor and the pump are real-world applications and deployed in a commercial context. According to the control systems experts of the industry partner, the training control systems are of medium complexity (avg. 18 Simulink operation blocks per controller) while the brushless motor control system is of maximum complexity (302 Simulink operation blocks for the controller itself) and the pump control system is of medium complexity (143 Simulink operation blocks for the controller itself). Note that an interesting aspect of the pump control system is the usage of characteristic curves making this control system highly non-linear.

7.4.1 Reproducibility

To address reproducibility, we question how reproducible the pure random search and adaptive random search are. We solely use our baseline configuration for reproducibility using 10x10 regions with 10 points per region performing 30 repetitions of search space exploration using pure random and adaptive random search (remember that Section 7.1 introduces the idea of searching in a two-dimensional search space). Note that, although statistically 30 repetitions may appear to cover the search space of the controller, the signal values use double precision and the search space is not linear. Using all worst cases obtained in the 30 repetitions, we compute the standard deviation, the coefficient of variation (COV) and the quartile coefficient of dispersion (QCD) for each quality criterion. These measurements are used to assess the dispersion of the worst cases throughout the experiments. Good reproducibility is witnessed by low values for all three measures. In addition, we measure the percentage of how many times (out of the thirty replications of each simulation) the search space exploration came at least 95% close to the found worst case. This will be referred to as 95% closeness in the sequel. However, 95% closeness is a particularly strong measure

and we believe 80% closeness to be sufficient in practice. In the following, we will further examine all cases where the measurements were not 0 and 95%, 90% and 80% closeness was not achieved since otherwise reproducibility is obviously given.

RQ1: Is pure random search reproducible?

For the step failure model, the standard deviation for the training control systems is 0 for all quality criteria except for smoothness in control systems 1 and 4. However, the coefficients of variation are 0.01 and 0.02 and the quartile coefficients of dispersion are 0.01 and 0.01 respectively yielding a marginal decrease in reproducibility (first row in Table 7.2). Only 95% closeness of the smoothness of control system 4 is slightly impacted. As it stays within 90% closeness, the decrease is negligible. In the complex control system, the coefficient of variation and quartile coefficient of dispersion are 0.1 and .07 for liveness as well as 0.08 and 0.07 for steadiness leading to impacted results in 95% closeness. As 80% closeness is almost completely given, we believe this to be sufficient in practice. For all other quality criteria, the standard deviation is 0 in the complex control system and the pump control system.

For the sine failure model, the standard deviation for the training control systems is 0 for all quality criteria except for smoothness in control systems 1 and 4. The respective coefficients of variation are 0.04 and 0.04 and the quartile coefficients of dispersion are 0.03 and 0.02 (second row in Table 7.2). Again, this yields a marginal decrease in reproducibility. The impact on the smoothness of control system 1 and 4 is 30% of 95% closeness, 10% at 90% closeness and 0% at 80% closeness. This means that 9 of the worst cases are further than 5% apart form the worst worst case, but all worst cases are within a 20% radius, which we believe to be sufficient in practice. In the complex control system, the coefficient of variation and quartile coefficient of dispersion are 0.08 and 0.06 for stability, 0.07 and 0.08 for liveness, 0.03 and 0.01 for smoothness and 0.04 and 0.02 for steadiness. This impacts the 95% and 90% closeness, but yields 80% closeness. For the pump control system, the standard deviation is 0 when feed forward is disabled. When enabled, the coefficient of variation and quartile coefficient of the smoothness are 0.13 and 0.04. Again, 80% closeness is given.

Table 7.2: COV / QCD / 95% / 90% closeness for smoothness*

Failure Model	Control System No.							
	1				4			
	COV	QCD	95% cl	90% cl	COV	QCD	95% cl	90% cl
Step	0.01	0.01	100%	100%	0.02	0.01	97%	100%
Sine	0.04	0.03	70%	90%	0.04	0.02	70%	90%
Disturbance	0.04	0.01	70%	90%	0.04	0.02	70%	90%

*All values 0 for other control systems

For the disturbance failure model, the result for the standard deviation for the training control systems is identical to the sine failure model (third row in Table 7.2). In the complex control system, the coefficient of variation and quartile coefficient of dispersion are 0.07 and 0.06 for stability and 0.16 and 0.12 for liveness. This impacts the 95% and 90% closeness, but yields 80% closeness.

From the results above, we tentatively conclude test case generation with pure random search to be highly reproducible for the presented control systems. However, we see the smoothness impacted when using feed forward control as "Feed-forward calculates a best guess; it predicts the signal that should be sent" [50]. This impact is not representative and needs to be further investigated.

RQ2: Is adaptive random search reproducible?

For the step failure model, the standard deviation is negligibly small except for the smoothness quality criteria in control systems 1 and 4. However, the coefficients of variation and the quartile coefficients of dispersion are again negligible small. For the complex control system, the coefficient of variation and quartile coefficient again decreases the reproducibility marginally, but 80% closeness is almost completely given.

For the sine failure model, the result for the training control systems is identical to the step failure model. In the complex control system, the coefficient of variation and quartile coefficient of dispersion again decreases the reproducibility marginally, but 80% closeness is completely given. When feed forward is enabled in the pump control system, our measures of reproducibility yield the same values as with pure random search space exploration for the smoothness. Otherwise, the standard deviation is 0.

For the disturbance failure model, the result for the training control systems is identical to the step failure model. In the complex control system, 95% closeness is impacted by 77% on average for liveness, but 80% closeness is given for all quality criteria.

We hence conclude the reproducibility of the worst case in adaptive random search space exploration to be high. We speculate the reason for the decrease of closeness to be the use of discrete time and feed forward for some controller components in control system 4, 7 and 9. However, further investigation is required.

7.4.2 Effectiveness

To address the aspect of effectiveness, we question how "good" the worst cases found by pure random and adaptive random search space exploration are. Our baseline for effectiveness is as stated above and we perform 30 repetitions of pure random and adaptive random search space exploration. Our measure for relative effectiveness is the median worst case found for each quality criterion by both methods as it constitutes the average expected worst case. For absolute effectiveness, we use test cases derived by three control system experts and compare their worst case with the absolute worst case

found for each quality criterion. Good absolute effectiveness is given if we find a better worst case than the manual tests. In addition, finding a better worst case violating the requirements would even demonstrate failure-finding ability of the methodology.

RQ3: What is the relative effectiveness of pure random and adaptive random search space exploration?

Table 7.3: Control system 1: % improv./deteri. of adaptive vs. pure random search space exploration

Failure model	Stability	Liveness	Smoothness	Responsiveness	Steadiness
Step	0	0	0.7	0	0
Sine	0	0	3.4	0	0
Disturbance	0	0	3.6	0	0

Table 7.4: Control system 3: % improv./deteri. of adaptive vs. pure random search space exploration

Failure model	Stability	Liveness	Smoothness	Responsiveness	Steadiness
Step	0	0	1.1	0	0
Sine	0	0	1.3	0	0
Disturbance	0	0	1.6	0	0

Table 7.5: Control system 7: % improv./deteri. of adaptive vs. pure random search space exploration

Failure model	Stability	Liveness	Smoothness	Responsiveness	Steadiness
Step	3.4	-3.8	-0.4	0	1.2
Sine	-1.8	3.7	0.7	0	2.7
Disturbance	7.1	3.7	2.1	0	7.3

As values were 0 except for the values depicted in Tables 7.3, 7.4 and 7.5, there is no winning strategy in terms of relative effectiveness in our experiments for the step and sine failure model. The best demonstration is the results of the complex control system as a maximum improvement and deterioration of 3-4 % occur at the same time (first row of Table 7.5). For the disturbance failure model, the adaptive random strategy is better by a maximum of 7% (third row of Table 7.5). We consider this a marginal increase not relevant in practice. Thus, we tentatively conclude both strategies to be equally relatively effective.

RQ4: What is the absolute effectiveness of the methodology compared to human test case derivation?

We analyze the worst case found for each requirement (see Table 7.1) and compare them to manual test in cases provided by the control system experts. The manual test cases for control systems 1, 2 and 4 were to try a step of 500/600, 1000/2000

Table 7.6: Violated controller requirements by expert (ex) / controller tester (ct) test cases

No.	Stability	Liveness	Smoothness	Responsiveness	Steadiness
1	-	-	-	ex + ct	-
2	ex + ct	ex + ct	ex + ct	ex + ct	-
3	-	-	ct	ex + ct	-
4	-	-	ct	ex + ct	-
5	-	ex + ct	ct	ex + ct	-
6	ex + ct	ex + ct	ex + ct	ex + ct	ex + ct
7	-	-	-	-	-
8	-	-	-	-	-
9	-	-	-	-	-

and 2000/6150 rpm and a step from 0 to all values of the previous steps. For control systems 3 and 5, the tests cases involved moving the pin out by 2, 3, 5, 7 and 10 cm (see Table 7.1). Control system 5 was to move the pin out by 0, 0.5 and 1 m and control system 7 was to use a step from 500, 800, 2000 to 4000 rad/sec. The pump control system was to use a step from 2, 3, 5 to 7 m^3 with disabled and enabled feed forward. The worst case produced by the methodology on average was better than the worst case of the test cases provided by the control system experts for the training control systems 92.5% of the time. The median improvement by using the methodology was 44% compared to the experts' test cases. The best improvement was the smoothness of control system 4 being worsened by 8000%. This worst case also violated the requirements and was one of the violations only detected by our methodology as seen in Table 7.6. The requirements of the brushless motor controller were never violated although the control system experts did only propose 2 of 4 test cases leading to the worst cases. However, since this control system was extensively tested and is used in practice, we could increase the confidence of the control system experts w.r.t. the brushless motor controller functioning according to its requirements. The pump control system's requirement were also not violated, but the worst case was improved by 22% as decreasing the flow led to worse values than increasing. Using the sine failure mode, we could detect instability at 1 and 1.1 kHz for controllers 1 and 4. For all other controllers any instability when using the sine failure model up to a frequency of 10 kHz (given by the control system experts) was negligible. An interesting aspect of the sine failure model was the clear visibility of the increase of instability described as particularly interesting for new control system engineers by the control system experts. For the disturbance failure model and a trapezoidal ramp (given by the control system experts), the worst case was marginally different to the worst cases of the step failure model. This is an indication towards the ability of the control systems to withstand disturbances. Thus, we consider failure models to be as effective or better in comparison to manual testing.

7.4.3 Efficiency

RQ5: Is the search space exploration executable in reasonable time?

Concerning efficiency, one training control system's search space exploration using 8x8 regions and 5 points per regions took approximately 2 minutes, while the baseline took approximately 3 minutes. The largest configuration evaluated was 12x12 regions and 20 points per region taking 8 minutes per control system. For the complex control system, the computation lasted 10 minutes for 8x8 regions and 5 points per region, 28 minutes for the baseline and 70 minutes for 12x12 regions and 20 points per region. The pump control system's search space exploration lasted approximately 3 minutes for 8x8 regions and 5 points per regions and 6 minutes for the baseline. The time required for each test case was only determined by the time Matlab required for simulation as generation of test cases and measurements take negligible time. Thus, the employed failure model did not influence the overall execution time. Note that, the test case generation and execution is highly parallelizable. When presented with the execution times, the control system experts found the time adequate for practical purposes. Thus, we tentatively conclude to have high efficiency and a linear scalability w.r.t. the overall number of points to be used in our case study.

7.4.4 Sensitivity

To assess the sensitivity of the methodology, we performed experiments using configurations differing from the 10x10 regions with 10 points baseline as described above. In terms of the reproducibility of the search space exploration, we want to examine the correlation between the number of points, and number of regions, used during the search space exploration and their caused increase in reproducibility. In terms of the relative effectiveness of each configuration, we are interested in a correlation between the absolute worst case found for a quality criteria in each controller and the number of points and regions used during the search space exploration. For the evaluation of sensitivity, we use all configurations above for the evaluation with 10 repetitions.

RQ6: How does the reproducibility change when using different configurations?

The results of our experiments using the step failure model showed a configuration of 8x8 regions and 5 points per region to be sufficient for control systems 2, 3, 5 and 6 reaching 95% closeness 100% of the time. The same was true for control systems 1 and 4 in all quality criteria except for smoothness. The smoothness of control system 4 yielded the worst case, but almost the same values were measured for control system 1. Again, the decrease of reproducibility was marginal and did completely disappear when using either 12x12 regions or at least 20 points per region. Interestingly, an increase in reproducibility was observed when increasing the number of points per regions as seen in Figure 7.7, but not when increasing the number of overall points. For

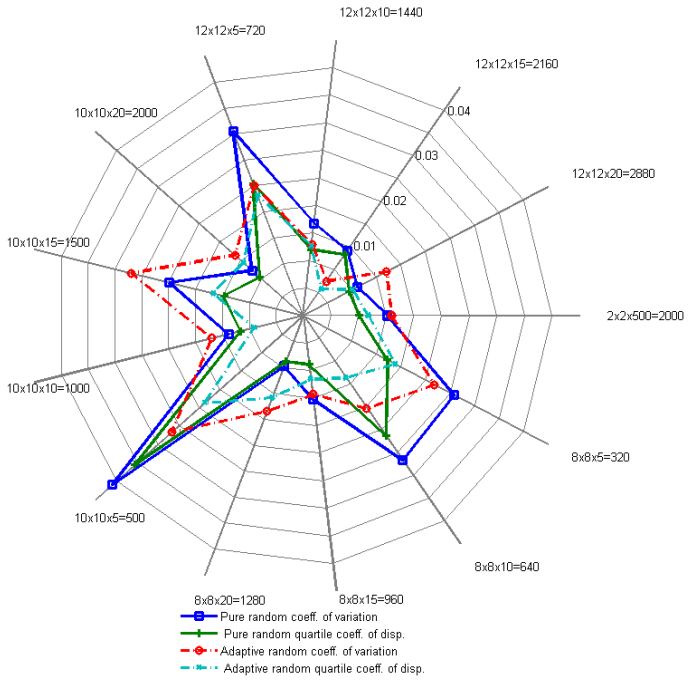
the complex control system, the reproducibility increased with the number of points used in 90% closeness in all quality criteria, but is completely given in 80% closeness. For the pump control system without feed forward enabled, 8x8 regions and 5 points per region are sufficient. With feed forward enabled, smoothness is impacted in 95% and 90% closeness, but is given in 80% closeness. Putting this into perspective, we examined 5 quality criteria for 9 controllers leading to 45 experiments out of which only four had a very small dispersion possibly the result of jitter.

For the sine failure model, a configuration of 8x8 regions and 5 points per region is sufficient for control systems 2, 3, 5 and 6 as argued above. Again, the decrease of reproducibility of control system 1 and 4 was marginal. Also the reproducibility increased when increasing the number of points per region. The results for the complex and pump control system is the same as with the step failure model.

For the disturbance failure model, a configuration of 8x8 regions and 5 points per region is insufficient only for the smoothness of systems 1 and 4. However, this insufficiency is resolved when using 10 instead of 5 points per region int the 95% closeness. 80% closeness is always given. For the complex control system, 90% closeness is impacted for the liveness. It is always given with 80% closeness, which is true for all other criteria as well.

Thus, we tentatively conclude all configurations to yield a high reproducibility. Therefore, a configuration of 8x8 regions and 5 points per region yields the highest cost-effectiveness due to its short run time. In addition, we speculate it to be better to increase the number of points per region and not the number of regions themselves based on our limited results.

Figure 7.7: Reproducibility of smoothness for control system 4 in the step failure model



To further examine whether the number of regions affects the reproducibility in the step failure model, we performed experiments using an overall number of 2000 points with 144 (12x12) and 4 (2x2) regions. As depicted in Figure 7.7, there is only a marginal difference when using this number of points as the maximum value is 0.04 and the minimum is 0. These results also applied to the sine and disturbance failure models. Putting our findings into the perspective of Weyuker and Jeng [150], we intuitively compare random to partition-based testing. Our comparison results give an empirical perspective of the theoretical model yielding (almost) the same results for both as the underlying landscape formed by the quality criteria is unknown. However, using more regions may improve usability for a control system engineer inspecting the heat map produced by the search space exploration.

Examining the complex control system using only 8x8 regions and 5 points per region in the step failure model, the coefficient of variation was 0.07 and the quartile coefficient of dispersion was 0.07 for stability. Again, these values show a marginal loss of reproducibility. When using 12x12 regions and 20 points per region (2880 points in total), the coefficient of variation and the quartile coefficient of dispersion had their best values also increasing the 80% closeness for all quality to 100%. Again, theses results also applied to the sine and disturbance failure models. Thus, we tentatively conclude the reproducibility to increase when using a larger number of points. This results in a rule of thumb to rather increase the number of point than the number of regions. However, this increase appears to be non-linear and correlated with the number of regions. Thus, a careful characterization of the search space in the future is required to draw a conclusion.

RQ7: How does the effectiveness change when using different configurations?

As the reproducibility for control systems 2, 3, 5, 6 and 8 was high in the last section in the step, sine and disturbance failure models, the median worst case was constant in all configurations for these control systems. Again, the same was true for control system 1 and 4 except for the smoothness quality criterion. However, as depicted in Figure 7.8, the median of the worst case of the smoothness of control system 4 in the step failure model stays in a range of 3.5% average and 5% maximum deviation yielding negligible differences. Examining the median worst case found using only 8x8 regions and 5 points per region for the complex control system, the ratio of median divided by the absolute worst case (median ratio) found already reaches a sufficient 87% for smoothness for compared to the baseline of 89%. When using 12x12 regions and 20 points per region, the ratio increases to 97% for smoothness being also the best ratio for this configuration. In the pump control system with feed forward, the median ratio is 75% for 8x8 regions and 5 points per region and 91% for the baseline. We speculate these results to be correlated with the use of feed forward control, but further examination is required.

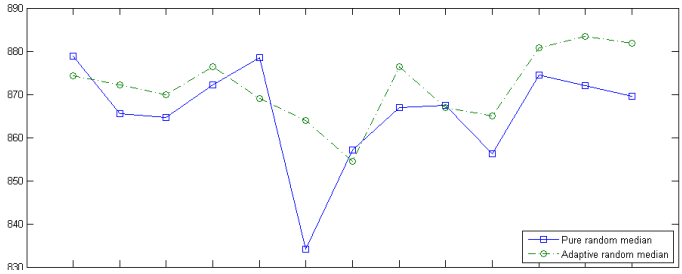


Figure 7.8: Relative effectiveness of smoothness for control system 4 in the step failure model

For the sine and disturbance failure model, the median ratio of the liveness for the complex control system is 77% for 8x8 regions and 5 points per region 90% for the baseline and 91% for the maximum of 12x12 regions and 20 points per region. For the pump control system with feed forward and the sine failure model, the median ratio is 82% for 8x8 regions and 5 points per region and 87% for the baseline. For the maximum of 12x12 regions and 20 points per region, the median ratio is 97%.

Thus, all evaluated configurations yield a high relative effectiveness of 80% or more when using the baseline configuration. Using fewer regions and points per region influences the 80% closeness in medium and highly complex control system, but is sufficient in low complexity control systems.

7.4.5 Summary

In summary, we found pure random and adaptive random search space exploration to be reproducible and effective using the provided control systems. In particular, reproducibility and effectiveness of the search space exploration are essentially the same for both strategies. We see a larger increase when using more points per regions instead of using more regions and speculate there to be a correlation. This hints towards the number of regions to not affect the worst case, but only the readability of the resultant heat map. When examining the test cases given by the control system experts for the training control systems, the methodology yielded better worst cases 90% of the time with a maximum increase of 8000%. For the complex control system, the methodology provided better worst cases for 50% of the test cases with a maximum increase of 25%. We deem the methodology to be effective and efficient when compared with manual testing. In addition, search space exploration of all control systems was possible within acceptable time constraints using the baseline setting.

Although search space exploration and single-state search are automatic, there is no oracle for the test cases. Thus, there is an effort involved in manually inspecting the respective worst cases. However, this effort is minimal as only the worst case or all test cases voiding the requirements have to be inspected. We speculate these to be a single digit number of test cases. However, to gain generalizable results concerning the manual inspection effort, further (possibly large-scale case-study) research is required.

7.5 Related Work

This work is based on the automated testing of continuous controller methodology introduced by Matinnejad et al. [105, 106]. It aims to generalize and extend the approach by using failure models derived from the violation of quality criteria. A detailed description has already been given in Section 7.1 as it is necessary to understand the generalization and extension performed. For an overview of testing of continuous control systems see [106]. We see the main difference in our perspective that is based on failure models and that we deem to be more general, as witnessed by more quality criteria (Section 7.2) and more input constraints / scenarios (Section 7.3) that we provide in this paper. The employment of defect models for quality assurance is related to the idea of using defect taxonomies to improve software testing [55] and failure mode and effects analysis (FMEA) [124]. Due to the their nature, these methodologies deliberately focus on making defect knowledge explicit and allow the derivation of test cases specifically targeting these defects. This is similar to mutation testing, where the quality of test suites is assessed by their ability to detect explicit defects after injecting them into the system under test [79].

In the area of MIL control system testing tools, there are Simulink Design Verifier [108], Reactis [131] and TPT [146]. These tools allow the manual specification and automated execution of test cases. Design Verifier and Reactis can generate test cases for coverage. TPT uses a graphical test case notation abstracting from actual I/O in a keyword-driven way. Reactis is the only tool able to generate random test inputs. However, none of the tools use a search-based strategy or failure-based testing. To our knowledge, also no results w.r.t. reproducibility, effectiveness or efficiency have been published.

The effectiveness and reproducibility or predictability of random testing as well as its advantages and drawbacks have been largely discussed by Arcuri and Briand [4, 6]. In addition, they study various types of random testing and present potential practical application scenarios including tailorable evaluation methods. The effectiveness of random testing and particularly adaptive random testing is also discussed [5].

7.6 Conclusion

In this chapter, we have proposed to derive tests for continuous control systems on the grounds of potential failures in different scenarios. These scenarios are based on failure models. Failure models describe violations of various quality criteria together with explicit constraints on the input to form scenarios. We have shown that existing work [4], [5] can be cast in our methodology, and have provided additional failure models. Violations of some quality criteria may be safety-critical giving our approach the ability to detect such violations or increase the confidence in the absence of these violations.

As a second step, we have looked into the operationalization of failure models for test case generation. Again building on existing work, we have studied characteristics of a (re-implemented) test case generator [105, 106] for continuous controllers. These characteristics include reproducibility (because test generation in parts is random, results should not be fundamentally different in each run), effectiveness (because the tool should not yield worse results than a human expert), and efficiency (because results should be available quickly). If these characteristics are demonstrated, the operationalization can increase expert effort relief and front-load the detection of common and recurring failures.

In our experiments, we found both random and adaptive random search space exploration to yield essentially the same reproducibility and effectiveness. However, this depends on the ability of the search space exploration to capture the landscape produced by the quality criteria. In our experiments, we used several training and two fully fledged real-world complex control system. Thus, we are unable to form a conclusion about the mapping of the landscape in general. However, the worst cases found by the methodology were often better than the manual test case worst cases, yet mostly did not violate the quality requirements.

In the future, we need to characterize the search space using many different configurations. Approaches to visualize large search spaces including respective changes when varying configurations have already been introduced [66] and may be transformable into our context. We were able to provide initial insights concerning the number of regions (seems to be negligible) and points per region (seem to matter more) in our evaluation. However, it was impossible for us to come to a conclusion regarding the effect of increasing either regions or points per region. As an exemplary future use case, Controller Tester could be adapted to test autonomous systems (e.g. drones) and gain confidence in their correct behavior (e.g. for certification purposes). In addition, we want to research ways of detecting faults in control systems such as implicit data type conversions, wrong integrator modulations, divisions by zero and state variable overflows possibly re-using existing solutions (e.g. 8Cage in Chapter 5). In addition, the methodology can be extended to control systems with multiple inputs/outputs by extending the number of dimensions of the search space. However, this thesis deliberately focuses on providing an extensive and comprehensive library of failure models for control systems.

Finally, the presented methodology is not limited to MIL testing, but tests can be re-used on other levels such as SIL and HIL. However, when testing at the MIL level, the quality of the test results depends on the accuracy of the plant model.

8

Assessment and Maintenance

The previous chapters defined the foundations for a systematic and (semi-)automatic approach to defect-based quality assurance based on defect models, elicited and classified relevant defects and demonstrated the effectiveness/efficiency of their description and operationalization on all test levels in the domain of Matlab Simulink. The effectiveness/efficiency experiments in the respective chapters were an inherent assessment of the underlying defect models and their operationalizations. Generalizing these individual assessments, this chapter presents a framework for the assessment of defect models. Furthermore, a core aspect of the employment of defect models in quality assurance is their continuous improvement. This includes eliciting and classifying new typical defects, adjusting existing descriptions and operationalizations according to evolving contexts and continuously assessing the cost-effectiveness. Thus, a framework for a support process to perform this defect model maintenance is required, which is able to trigger respective lifecycle activities if needed. This chapter describes the frameworks for assessment and maintenance in the final step called controlling in the defect model lifecycle framework as seen in Figure 8.1.

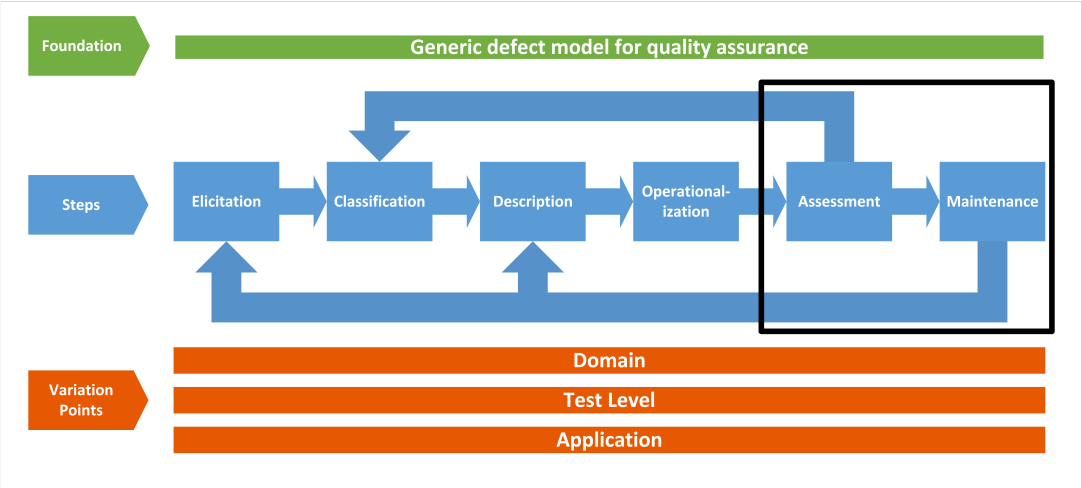
8.1 Assessment

In chapters 5, 6 and 7, the defect models were not only described and operationalized, but also assessed as part of the evaluation. The assessment consisted of evaluating effectiveness, efficiency and reproducibility of the different approaches as to demonstrate their practical applicability. To demonstrate effectiveness and efficiency of an operationalization in a context, two questions can be generalized from the previous assessment of defect model operationalizations: (1) does the operationalization detect the right defects and (2) is the operationalization capable of detecting the defects right.

defect model validation

The first question is concerned with the validation of defect models and related to validation of requirements, which constitutes quality assurance in the requirements engineering activities [140]. The task of validation is to assure the requirements of a system reflect the organizational needs. The validation of defect models in the

Figure 8.1: Position of assessment and maintenance in this thesis



assessment activity is to assure the description and operationalization of the respective defect model(s) is according to the defects elicited and classified earlier. This ensures the description and operationalization to be aligned with the organizational goals/aims. This means that from an organizational perspective, it is "useful" to develop these operationalizations. However, organizational goals/aims justifying the development of an operationalization may not necessarily reflect the needs of every individual project and a project-specific assessment (see below) before the employment of any developed operationalization is recommended. As is the case with any software development, defect model validation is recommended to be performed as early as possible as to avoid operationalization effort by using a "wrong" defect or designing the operationalization "wrong". Thus, the validation of defect models assures the first part of the definition of a good test case in Chapter 3 concerning the targeting of potential faults.

defect model verification

The second question is concerned with the verification of defect models. The description and operationalization of defects models yields a tool. Such a tool likely contains faults as it is developed using software engineering methods. These methods typically already include the verification of each development step eliminating these faults. Apart from removing faults, the evaluation of their practical applicability and cost-effectiveness as stated by the second part of the definition of a good test case in Chapter 3 is an important aspect of the verification of defect models. The evaluation of cost-effectiveness involves performing an experimental case study [141]. To assure cost-effectiveness, the case study to be performed in real projects using the operationalizations needs to demonstrate a decrease in expert workload and increase of the early detection of the described defects. It is recommended to perform such a case study using the same project as treatment and control project. However, such a resource-intensive case study may not be possible in industry.

project-specific assessment

A third and individual assessment may be performed before employing any operationalization in a project. This project-specific assessment is performed as part of test

management [53] and concerned with the suitability of an existing operationalization in the context of a project. An existing operationalization is suitable to be used in a project, if the described defects are potential defects in this project (1) and the variation points of domain, test level and application match the project (2). Both these factors then are an indicator of cost-effectiveness. This concurs with the definition of defect model effectiveness in Section 3.1.7 as the broader an operationalization is w.r.t. the variation points, the more it can be re-used in projects and the more cost-effective its development.

In sum, the generalizations above yield a framework to assess defect model operationalizations. Defect model validation has been indirectly performed by assessing the effectiveness of defect models in the previous chapters. Defect model verification has been applied while developing the operationalizations, but has only been evaluated w.r.t. the effectiveness and efficiency of the operationalizations. The assessment as part of test management has not yet been performed. The parts not performed form parts of a framework yet to be evaluated. However, such an evaluation is infeasible in a Ph.D. dissertation.

8.2 Maintenance

Maintenance of defect models requires the continuous assessment of occurring defects and the employment of defect models in quality assurance within an organization. Thus, the maintenance of defect models is an organizational support process executed in parallel to the software engineering processes. In this section, we present a framework for such a process for completeness reasons, which however is yet to be instantiated and evaluated in practice. A prerequisite to the inclusion of defect models in software quality assurance is the acquisition and retainment of management and engineer support. Unless engineers are able to openly discuss defects and face no consequences by management, the description and operationalization of effective defect models is impossible. Thus, the maintenance process acts as bootstrapping process for the embedding of defect models in quality assurance and tailoring and initiating of the activities in the defect model lifecycle framework. The bootstrapping involves the assessment of existing quality assurance and the collection of possible employment scenarios for defect-based quality assurance based on defect models. After bootstrapping, the activity of elicitation is initiated as described in Chapter 4 and the respective follow up activities are performed. The maintenance process can be integrated in existing test or quality management, which is typically an already existing support process in quality assurance activities [53]. Since the selection of test strategies and their assessment is already considered in test management, defect-based quality assurance based on defect models is able to be seamlessly embedded. Alternatively, dedicated quality engineers can be employed for the assessment of quality including defect-based quality assurance as proposed by Steidl et al. [142]. These engineers

need to monitor and adjust the instantiated activities of the defect model lifecycle framework.

Elicitation and classification (in Chapter 4) determine occurring defects and enable the decision to describe and operationalize defect models. Performing elicitation and classification (e.g. using DELICLA) gives a snapshot according to the chosen context of the elicitation. This snapshot is limited to (1) the past and (2) the context (e.g. systems developed in Java with framework X). To overcome the limitation of only portraying the past, elicitation and classification can be repeated as required to monitor / predict defects and their respective occurrence/severity rates. Concerning the limitation of context w.r.t. the snapshots, the elicitation and classification is context-independent (see Chapter 4) and can be performed in all relevant contexts. In addition, future defects may be anticipated by scouting for technology, organizational or domain changes. These changes can frequently occur and the defect model maintenance process must monitor/anticipate such changes proactively. As an example, if prototype projects for the familiarization to new technology exist, elicitation and classification for new defect models may be started within these projects to gain knowledge of common and recurring defects.

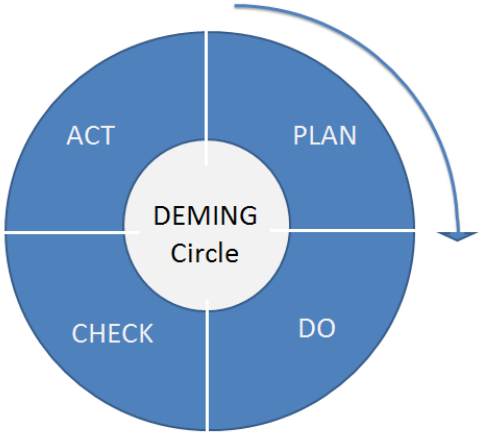
The defect model maintenance process must maintain existing operationalizations of defect models by continuously assessing their effectiveness and efficiency. Concerning the effectiveness, we speculate there to be a decrease w.r.t. each operationalized defect model over time due to personal learning [134]. If a fault is detected by the operationalization of a defect model multiple times, the responsible engineer is subject to learning and is expected to learn from the automated detection. We hypothesize this personal improvement to decrease the effectiveness of the respective defect model as the described faults are avoided rather than detected. However, individuals and organizations tend to also forget [67]. In addition, a change in technology such as programming language may render operationalizations of defect models useless. Concerning the efficiency, the increasing complexity of systems may decrease or obliterate the efficiency of operationalizations of defect models. However, improvement in technology may improve tool efficiency and increased processing power may also increase efficiency. Thus, the defect model maintenance process must monitor/anticipate learning, changes, complexity and tool improvements proactively.

To satisfy the requirements of the defect model maintenance process above, existing continuous quality improvement methodologies can be re-used to give a framework. However, they need to be adjusted and possibly combined to fit to the project and organizational context. In the following, we describe the relevant existing methodologies to possibly base the maintenance process on.

PDCA

One of the earliest continuous quality improvement methodologies is PDCA by Deming [45] conceived in 1986. PDCA is an iterative four-step process shown in Figure 8.2 also known as Deming circle and short for PLAN-DO-CHECK-ACT. During PLAN, quality goals, a plan to reach them and measures for their assessment are created. DO

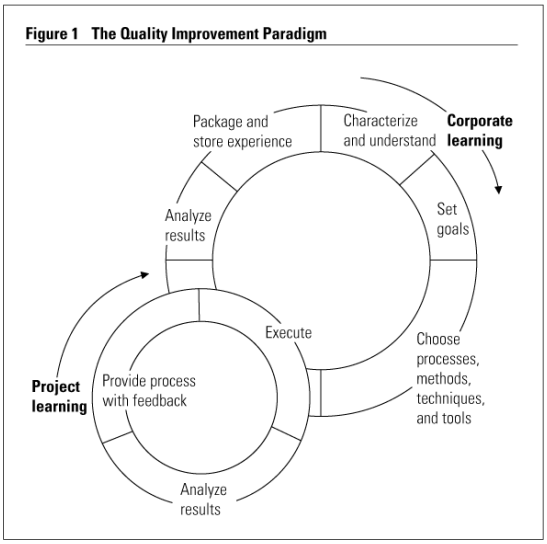
Figure 8.2: PLAN-DO-CHECK-ACT cycle



Source: https://commons.wikimedia.org/wiki/File:Deming_PDCA_cycle.PNG

refers to the implementation of the plan. CHECK assesses the execution of the plan in DO and compares expected to actual quality. ACT gives a retrospective concerning the plan, its implementation and assessment. If the quality goals are reached, PDCA is terminated. In case the quality goals are not reached, a further PDCA cycle is started using the insights obtained by the previous PDCA cycle.

Figure 8.3: Quality Improvement Paradigm (QIP) [13]



In the field of continuous quality improvement methodologies in software engineering, Basili et al. created two well-known approaches called QIP [14] based on PDCA and Experience Factory [12]. QIP is the quality improvement paradigm shown in Figure 8.3 and an iterative process. On the organizational level (also called capitalization cycle), there are six sequential steps, out of which one is shared with the project level. The project level (also called control cycle) has three steps. In the first step of characterize and understand, the context of the organization is modeled. In the

QIP

second step, the quality goals to achieve are set. “To satisfy the goals relative to the current environment, it chooses processes, methods, techniques, and tools, tailors them to fit the problem, and executes them”) [13]. The execution is performed within one or multiple projects. Within these projects, results are analyzed according to respective measurements and feedback is provided to the organizational level. Depending on the outcome, execution may be performed multiple times. On the organizational level, the results of the execution are analyzed and packaged / stored for future re-use. If quality goals were not achieved, QIP reiterates using the results from the current iteration.

experience factory

The experience factory acts as a storage of data and lessons learned from previous projects. It contains various forms of models and measures to instigate re-use of the knowledge and experience of previous projects. The experience factory always includes the contexts of the gathered data. In addition, it also requires a support process within the organization to keep its contents up to date. This process is required to be separate from the development activities. This is also a requirement for the maintenance of defect models and defect models can be seen as one instance of data and lessons learned stored in the experience factory.

Although the experience factory requires such a support process, it does not define an explicit maintenance process to adjust to changing contexts. An approach specifically targeting the maintenance of software is continuous software quality control [142]. In continuous software quality control, tools are used to determine the state of quality of a software product. These tools are adaptable and anticipate learning, changes, complexity and tool improvements in a reactive way. By having a support process using quality engineers maintaining the employed software metrics, the process and tool are able to be proactive.

While the defect model lifecycle framework can be seen as an instance of PDCA and QIP in addition to knowledge management, the maintenance of defect models can re-use continuous software quality control methods. Maintenance of defect models has similar requirements and can use test metrics to be recorded and analyzed rather than source code metrics in continuous software quality control. Exemplary metrics are the number of defects found by employing defect model including a break down to each defect model and the time to find these defects. However, as is the case with continuous software quality control, there is no one size fits all metric and they will have to be tailored on an organizational and project level. This tailoring directly fits into the instantiation/tailoring of the defect model lifecycle framework to the project/organization.

A purely process-based quality improvement approach usable for constructive quality assurance and re-usable for the assessment of the maintenance process is

CMMI

CMMI [143]. CMMI is short for Capability Maturity Model Integration and hypothesizes quality improvement to be possible by pure development process improvement. CMMI has five maturity levels called initial, managed, defined, qualitatively managed and optimizing. Every development process is at least on the initial level and can gain levels

by employing certain methods. These methods are from process areas reaching from project planning on the defined level to risk management on the managed level and causal analysis and resolution on the optimizing level. A development process in an organization then gets audited certified to have a certain maturity level. Defect models and the defect model lifecycle framework can be used on all maturity levels. However, defect models require organizational investments, which are typically only performed on the defined, qualitatively managed and optimizing levels. Particularly, the support processes desired for defect model maintenance are typically already performed for other purposes in the qualitatively managed and optimizing levels. Thus, a defect model maintenance process can highly profit from re-using and tailoring existing resources. In the advancement of CMMI called SPICE [76], defect models and the defect model lifecycle framework can be classified in the software support processes of software quality assurance and software verification.

TMMI

A derivative of CMMI is TMMI [145], which gives a guideline and reference framework to improve test processes. TMMI uses the maturity levels of managed, defined, measured and optimizing to assess test processes and propose process improvements. Defect models and the defect model lifecycle framework fall into the category of defect monitoring and prevention in the optimizing level and are able to integrate into the activities on this level. Similarly, TPI NEXT [139] defines core areas of test processes and categorizes them into the levels of initial, controlled, efficient and optimizing. One of these areas is defect management (K10), which is concerned with the monitoring of defect lifecycles and particularly how the project/organization deals with defects.

TPI next

In sum, there exist requirements for a support process for the maintenance of defect models in the form of proactive anticipation of changes to steer defect model creation as well as usage. Particularly, this support process continuously assesses the cost-effectiveness of the employment of defect-based quality assurance based on defect models throughout the organization. Cost-effectiveness of an existing operationalization is measured by the effort involved in employing the operationalization and the cost of a late or in the field removal of the defects it detected. There exist several methodologies catering to knowledge management (and re-use) as well as defect monitoring and prediction. If already employed or planning to be employed in the organization, these methodologies provide parts re-usable by the support process for defect model maintenance. Note that, we have only provided the requirements for the maintenance of defect models as defect-based quality assurance based on defect models has not yet been employed in practice. Thus, we are only able to provide a framework yet to be evaluated. However, such an evaluation is infeasible in a Ph.D. dissertation.

8.3 Conclusion

The assessment and maintenance activities are part of the controlling step in the defect model lifecycle framework. The main purpose of this step is to monitor and adjust the employment of defect models and proactively anticipate changes in technology, organization or domain. This allows an optimal and cost-effective employment of defect models and their operationalization is software quality assurance.

For the assessment of effectiveness and efficiency of defect model operationalizations, their requirements have to be validated to match with elicited and classified defects in the planning step. Defect model maintenance is a support process tailored to the project/organization. The maintenance process framework is able to re-use existing knowledge and experience management in software engineering (e.g. PDCA, QIP and the experience factory), if already employed in the organization. However, these approaches do not include continuous assessment. Thus, re-use of process and resource aspects of continuous software quality control and constructive quality assurance (e.g. CMMI, SPICE, TMMI and TPI NEXT) complete the framework to tailor the maintenance of defect models. The assessment and maintenance activities have yet to be evaluated in a large scale case study, which was infeasible as part of a Ph.D. dissertation.

The activities of assessment and maintenance are the last two activities in the defect model lifecycle framework. However, they are of most importance w.r.t. the organizational aspects of defect-based quality assurance with defect models. These processes concern the tailoring, introduction, operationalization and evolution of defect models within an organization. In addition, these activities bring together the cost aspects involved in quality assurance and defect-based quality assurance with defect models. Thus, they are at the core of the strategic business decision towards the usage of the presented approach.

Recalling that defect models as such are programming language/paradigm/methodology independent, a central question in the integration into existing quality assurance processes concerns their process-independence, particularly if they can be integrated independently of the employed software engineering process. Two prominent process models are the waterfall [132] (classic) and scrum [135] (agile) process models. Although these process models are said to be on opposite sides of the process model spectrum, the quality assurance activities performed are very similar. The original idea in the waterfall process model is to have a dedicated phase after implementation has been completed to deliberately perform the testing. Other analytical quality assurance activities (e.g. reviews / inspections, formal proofs, static analysis, etc.) can be performed after the creation of all artifacts at the end of a phase. Scrum aims to perform testing on each increment and leaves it up to the development team how and when to perform quality assurance. Using other agile software development techniques (e.g. test-driven development [16]), this may be throughout the implementation or completely at the end of each sprint. As for defect-based quality assurance based

on defect models, it can be integrated into any of these process models by being utmost flexible due to the defect model lifecycle framework. The lifecycle framework structures the integration into existing processes by allowing the tailoring of defect models and their operationalizations to the process models used in the organization. Particularly, the elicitation and classification of defect models is flexible w.r.t. the context (see Chapter 4). The generality of the defect model (see Chapter 3) allows the tailoring of the description to the context according to the variation points. The operationalization has the greatest dependence on the process as the point in time the operationalization is used during the process may be vital to its cost-effectiveness. In a waterfall process model, careless mistakes (e.g. divisions by zero) may still be present when the testing phase starts (see Section 5.1) whereas these defects may have already been detected in a combined implementation/testing effort using test-driven development. Assessment and maintenance are again process-independent due to their pure supportive nature. Thus, the description and operationalization in the defect model lifecycle framework are the only activities depending on the process and must be carefully tailored, if defect-based quality assurance based on defect models is to be employed cost-effectively.

When employing defect models in quality assurance on the basis of the defect model lifecycle framework, there are also synergy effects w.r.t. process improvement. The DELICLA approach for the elicitation and classification of defects can be used to elicit process-related defects (see Table 4.2). Potential process improvements to prevent/mitigate defects in the process/organization could then be taken into constructive quality assurance considerations. Otherwise, they could be detected by other means of analytic quality assurance yielding a comprehensive quality assurance approach for such defects.

9

Conclusion

At the beginning of this thesis, we found the existence of common and recurring defects in a field study across project, organizational and even domain contexts. For their detection, software engineers tend to use either (1) use their own knowledge and experience or (2) implicitly re-use the encapsulated knowledge and experience in certain test selection strategies to derive test cases directly targeting these defects. These test cases have at least anecdotal evidence of their effectiveness, but are either engineer-dependent or based on test case selection strategies possibly not generally effective. Using our definition of a good test case being one that finds a potential fault with good cost-effectiveness (contrary to the pertinent literature), we find these test cases to potentially operationalize this abstract definition.

To systematically and (semi-)automatically create good test cases based on the personal or encapsulated knowledge and experience, our approach defines a generic defect model for the description of the personal and the investigation of the encapsulated knowledge. The described knowledge and experience relates to faults (captured in fault models) and failures (captured in failure models) comprising defect-based quality assurance based on defect models. By operationalization, the described defect models yield (semi-)automatic test case / check list generators targeting directly the described defects. Thus, they have the possibility to yield good test cases / check lists as the first part of the definition of a good test case is inherently operationalized and cost-effectiveness is possible.

To arrive at a defect knowledge repository containing defect models, we use knowledge management. Knowledge management aims to make knowledge and experience of software engineers available in project/organizational contexts. It consists of four steps to (1) acquire, (2) transform, (3) systematically store, disseminate and evaluate knowledge as well as (4) its application in new contexts (see Chapter 1). We present these knowledge management steps in a defect model lifecycle framework and instantiate them. The lifecycle framework gives a structure for the employment of defect-based quality assurance based on defect models and consists of three steps

containing 2 activities each (see Section 1.4). The first step of planning comprises the activities of elicitation and classification. These activities aim to give a comprehensive overview of occurring defects in specific contexts and enable the strategic decision making to describe some defects in defect models. The second step of method execution includes the description and operationalization of defect models. In this step, defect models for respective defects are described and operationalized by (semi-)automatic test case generators directly targeting the described defects. The third step of controlling includes the assessment and maintenance of defect models. This step aims to assess the previously created defect models w.r.t. their effectiveness and efficiency. In addition, the defect models are to be maintained by anticipating future changes in technology, organizational or domain causing further/different defects and adjust to personal/organizational learning effects. The defect model lifecycle framework is flexible w.r.t. how and when these activities are to be performed. However, there are a number of triggers described with the respective activities.

The planning step consists of the elicitation and classification of defects. Our approach DELICLA (see Chapter 4) is based on qualitative interviews for data collection and grounded theory for its analysis. Using a field study revealed DELICLA to be able to elicit and classify technical and process-related defects. It also identified cause effect relationships between some defects and demonstrated them to be context-independent. All defects are stored in a defect knowledge repository for their later use.

The method application step consists of description and operationalization, where some of the defects in the defect knowledge repository are formally described in defect models and operationalized. Operationalization is performed in three instances in software testing to represent the test levels of unit, integration and system testing in the domain of Matlab Simulink systems. All (semi-)automatic operationalization of the defect models demonstrates their effectiveness and efficiency in software testing.

The defect models on the unit test levels describe faults related to over-/underflows, divisions by zero, comparisons/rounding, Matlab Stateflow faults and loss of precision in single blocks / block combinations. The described failures relate to exceeding of I/O ranges. These defect models are operationalized in a tool called 8Cage (see Chapter 5). After configuration, 8Cage performs an automatic detection of the defects above consisting of smell detection, test data generation and test case execution.

On the integration testing level, the defects relate to the failures of superfluous or missing functionality and untested exception/fault handling in Matlab Simulink systems. These failure models are operationalized in a (semi-) automatic tool named OUTFIT (see Chapter 6). OUTFIT re-uses high coverage unit tests or creates them automatically using symbolic execution in an automatic three step process. The three steps (1) convert the units/components into an integrated system, (2) create test cases for the units/components as well as the integrated system and (3) execute the test cases to retrieve their achieved coverage. A manual inspection then reveals superfluous or missing functionality. By choosing exception/fault handling components as units/-

components, a manual inspection of the test cases reveals untested exception/fault handling.

The defect models on the system testing level are related to the quality requirements of control systems in Matlab Simulink. By negating the requirements and constraining the input space of the control system, our operationalization Controller Tester (see Chapter 7) aims to find their worst case behavior in different scenarios. Controller Tester is based on and generalizes existing work [105, 106] in the area of (semi-)automatic control system testing. We create a comprehensive library of defect models and quality criteria for control system testing. The results of all selected test cases are displayed in a heat map allowing the identification of the worst case as well as areas/input values close to the worst case/voiding the requirements.

In the assessment and maintenance step, we generalize the assessment of the operationalizations to be an evaluation of effectiveness, efficiency and reproducibility in multiple case studies. Effectiveness concerns the ability of an operationalization to detect the defects described in the respective defect models, while efficiency concerns the time and resources used for the detection. Since all of the operationalizations in this thesis use some form of non-determinism, reproducibility characterizes the ability of an operationalization to give the same results multiple times. 8Cage, OUTFIT and Controller Tester demonstrated effectiveness, efficiency and reproducibility within the constraints given by our industrial project partner. The maintenance of defect models is presented as a framework for a support process encompassing the employment of defect models in quality assurance. It performs the bootstrapping, introduction, operationalization and continuous assessment of defect models to adjust to personal/organizational learning and anticipates changes in technology, organizational or domain. The maintenance process can re-use existing knowledge and experience management in software engineering. Particularly, continuous software quality control and constructive quality assurance (e.g. CMMI, SPICE, TMMI and TPI NEXT) can be tailored to the project/organizational needs for the maintenance of defect models. In addition, the maintenance process is not only assessable by constructive quality assurance, but is also able to share defect knowledge with static analysis and constructive quality assurance responsible for process improvement. The assessment and maintenance activities have yet to be evaluated in a large scale case study, which was infeasible as part of a Ph.D. dissertation.

By creating the defect model lifecycle framework, we provide a structure of activities to include defect-based quality assurance into existing quality assurance processes and map it to knowledge management. These activities are tailorable to organizational and project contexts and are able to yield a defect, defect model and defect model operationalization repository. This repository is maintained by a support process and included in the planning of quality assurance activities. Thereby, the defect model approach can re-use defect knowledge systematically and (semi-)automatically within an organization or project.

By presenting a comprehensive and systematic approach to defect-based quality assurance with defect models and demonstrating its effectiveness and efficiency, we have contributed a systematic and (semi-)automatic way to employ tacit defect knowledge of software engineers and knowledge encapsulated in existing test techniques for quality assurance. In turn, this enables the systematic and (semi-)automatic operationalization of the definition of good test cases. In detail, the contribution is a generic model to characterize faults and methods to provoke failures (1), a context-insensitive qualitative interview method (DELICLA) to elicit and classify common and recurring defects (2), the defect model operationalizations 8Cage, OUTFIT and Controller Tester on unit, integration and system testing level including their demonstration of effectiveness, efficiency and reproducibility (3) and the defect model lifecycle framework to structure the integration of defect-based quality assurance into existing quality assurance processes (4).

9.1 Limitations

Although we have presented a systematic and (semi-)automatic approach to perform defect-based quality assurance based on defect models, it does not substitute but complement other quality assurance activities. As described earlier, defect-based quality assurance is to be integrated with existing quality assurance as it only detects defects known a-priori to testing. As a variety of defects (known, common and recurring as well as unknown) may occur when performing quality assurance, it is additional to existing field-tested and cost-effective analytical and constructive quality assurance methods. Concerning the cost-effectiveness of our approach, we have limited ourselves to the evaluation of effectiveness, efficiency and reproducibility in case studies. Larger studies involving (parts of) organizations using defect-based quality assurance based on defect models will have to be performed to demonstrate cost-effectiveness.

The operationalizations of defect models presented in this thesis are (semi-)automatic test case generators. Although test automation for the generated test cases is desirable, the automatic derivation of the required inputs may be infeasible (e.g. undecidable by algorithm). The user is then required to complete test inputs. In addition, the automated creation of oracles (see Section 2.1.2) is still an active area of research. Thus, the operationalizations of defect models may use a simple robustness oracle (i.e. no crash). In case a more complex oracle is required, its automated creation may be infeasible and manual inspection of test results is required.

In this thesis, we have only provided operationalizations for software testing of embedded systems in the Matlab Simulink domain. Although further operationalizations were performed for software testing of information systems, it was mostly infeasible to create (semi-)automatic tools for the generation of test cases in these systems. The generation of test cases requires deriving input values of a program, such that variables are assigned a certain value at a certain point of execution of the program

(i.e. semantic or dynamic constraints). These constraints are typically inserted into the system under test as the expression of a conditional statement. Upon reaching the conditional branch, the execution is halted and a variable assignment is derived. If this branch of the program is reachable is undecidable in general. However, heuristic-based techniques such as symbolic execution (see Section 2.2) or fuzzing (see Section 2.1.2) are able to reach these branches depending on the complexity of the program. The programming guidelines of embedded systems targeting their safety (e.g. MISRA-C [114]) yield big advantages to symbolic execution as no pointers or memory allocation must be taken into account. When symbolically executing information systems, symbolic execution is typically not as effective [25]. Particularly, unbounded and recursive data types (e.g. strings) or objects as inputs to methods in object-oriented programs yield typically unsolvable problems for symbolic execution. Without the creation of a test case and providing evidence to an actual fault to be present, defect-based testing using defect models is only able to yield lint-like [41] smell detection results including false positives. Moreover, we have not operationalized defect models in reading techniques, but only referred to existing work [54, 55]. Particularly, interesting is a comparison of effectiveness and efficiency of defect-based reviews/inspections and testing.

Each of the operationalizations also has its own limitations. First and foremost, each operationalization was evaluated in a case study and, therefore, the results may not generalize. The three systems and three integrated systems used in the evaluation of 8Cage and OUTFIT respectively may not be representative of all systems developed in Matlab Simulink. The control systems used in the ControllerTester evaluation may also not be representative of control systems in Matlab Simulink or control systems in general.

For 8Cage, we were only able to find overflows in the examined units, but 8Cage detected all other injected faults. Albeit, the performed injection could have been unrealistic (i.e. not representative of real-world faults). Also, the control systems in the evaluation turned out to be symbolically executable by KLEE, which is likely not the case for all control system in Matlab Simulink. Additionally, the extraction of units with global variables was problematic due to the possible involvement of its value in triggering a failure.

For OUTFIT, there are the limitations of the inability to create the $T_{A \leftarrow B}$ from T_B, scalability of symbolic execution and manual inspection effort. Recall that, to create a test suite $T_{A \leftarrow B}$ for component B in the integrated system based on its test suite T_B, the inputs required for the test cases in T_B must be the outputs of component A. In our experiments, the creation of $T_{A \leftarrow B}$ was hard and sometimes infeasible. The scalability of symbolic execution was a well known limitation found in 8Cage and OUTFIT as both use symbolic execution for test case generation. Also recall that, OUTFIT requires a manual inspection as the last step. The effort for this inspection was minimal in our case study, but these results likely will not generalize.

For Controller Tester, the limitations relate to one controlled variable, the manual inspection effort and the availability of a plant model. We restrict ourselves to one controlled variable and so called single-input single-output systems. However, our methodology can be transferred to multiple controlled variables and so called multiple-input and multiple-output systems in the future. The manual inspection effort of the test results produced by Controller Tester has not yet been evaluated. However, we believe it to be in the single digit numbers as only the worst case system behavior has to be inspected. One fundamental limitation in a broad-scale evaluation was the availability and accuracy of a plant model. Some engineers use dummy plants or no plant model at all when developing controllers making MIL testing impossible. However, the test cases generated by Controller Tester may also be used on other levels such as SIL and HIL.

In this thesis, we did not provide a concrete process for the maintenance of defect models, but only a framework to tailor/adjust it to projects/organizations. Since the maintenance of defect models requires human resources as well as the active employment of defect models in practical quality assurance, we were only able to determine the prerequisites for defect model maintenance. Thus, we can only speculate about its effectiveness and efficiency and future case studies need to be performed.

9.2 Insights gained and lessons learned

While working on this thesis, the primary lesson learned was the realization of the capabilities of symbolic execution. Although there are a plethora of publications with promising progress in the area of symbolic execution, the problem of efficient SAT/SMT solving remains. When first experimenting with symbolic execution for the operationalization 8Cage, the results looked promising and generalizable to further programming languages/paradigms/methodologies. However, reproducing the results of KLEE on the GNU coreutils in [25] already turned out to be non-trivial. When experimenting with other symbolic execution tools, such as java path finder [148] and its symbolic execution extension [3], the insufficient capabilities of the solvers immediately became clear.

A realization attained after creating 8Cage and before creating Controller Tester and OUTFIT was the primary usage of fault models on the unit testing level. Unit testing is the only form of testing, which is typically performed in a white-box fashion. Thus, there is knowledge about faults as software engineers look into the implementation of the unit when testing. In integration and system testing, the testing is mostly performed black-box style and without the knowledge of the underlying faults (except for directly visible faults such as interface incompatibilities). However, software engineers are well aware of effective methods to provoke failures leading to failure models on these levels.

An interesting note during the discussion of defect-based testing with defect models was the purpose of testing. The purpose of testing is not only to “identify as many faults as possible so that those faults are fixed at an early stage of the software development” [119] as presented in the relevant literature. It is also about an increase into the confidence of a system to fulfill its requirements. Following this reasoning also gives rise to the argument that solely using defect-based quality assurance for software quality assurance is not sufficient as an extensive description of all possible faults is impossible a-priori to testing and would also void the need for it.

9.3 Future Work

The future work is concerned with the extension of our approach to autonomous systems in the embedded domain and to other domains, a large-scale evaluation, usage in other parts of software testing and further defect elicitation techniques.

In the future, the approach of Matinnejad et al. [105, 106] for continuous control systems including our generalization can be extended to autonomous systems. Autonomous systems (e.g. a drone) typically have an operational range to execute a mission. This operational range creates a search space similar to the search space created by continuous control systems. This search space needs to be explored to (1) find the worst case behavior of the autonomous system and (2) possibly address the safety certification of the autonomous system (e.g. FAA certification by DO-178 [48]). To find the worst case behavior, the search space and quality criteria need to be adjusted to the context. If the autonomous system is a drone, exemplary scenarios could be all possible changes in position and all possible changes in altitude using respective initial and final way points. The quality criteria of stability, liveness, responsiveness and reliability can easily be adjusted towards using these way points, but further quality criteria may be required. To address the safety certification, the created heat maps may be used to ascertain confidence that the system behavior is safe. However, there is a caveat concerning self-learning autonomous systems. If the system behavior changes during or after performing testing, the results are not reproducible and other methods of testing or abstraction are required.

In this thesis, we have limited ourselves to the embedded domain for operationalization of defect models. The most prominent rationale behind this decision was the inability of symbolic execution (or any other related techniques) to yield test cases targeting a specific (set of) fault(s). To make our defect-based approach usable in the information system area, further research in the area of symbolic execution and program exploration is required to deal with unbounded and recursive data types and object-oriented concepts (e.g. polymorphism). In addition, we did not create an operationalization in the area of manual static analysis. A possible operationalization would be a (semi-)automatic defect-based check list generator. Felderer et al. [54, 55] describe using defect taxonomies in reviews of requirements to specifically target

common and recurring defects in the specification of these requirements. Creating a database of these defects including their classification would yield the input to such a (semi-)automatic defect-based check list generator. The check list generator is then a selector of the relevant defects for an artifact. A large-scale case study could then assess the effectiveness and efficiency.

The instantiation of all lifecycle framework activities in one project/organization is also still to be performed. Although we have performed a field study concerning the elicitation and classification of defects, a field study concerning the description, operationalization, assessment and maintenance is yet required to be performed to generalize our findings. Particularly the maintenance of defect models still requires to be instantiated in an organizational context, where our approach is used in a/multiple project(s).

Although the generic effectiveness of defect models contains the notion of risk-based quality assurance (see Section 3.1.7), we have not further investigated or evaluated any risk-based aspects. In the future, such investigations could be made to evaluate the contribution of defect-based testing in the risk-based testing part of software testing. Particularly interesting are the notion of Myers, who proposes to execute test cases that detect "most errors found" [117] inherently combining defect-based and risk-based testing. Moreover, it could be interesting to evaluate whether defect models enable risk assessment as far down as the unit testing level. Such risk attribution would yield a contribution to risk-based testing as all test cases on all levels can be associated with risks and enable early considerations of risk management. The cost-effectiveness of defect-based testing based on defect models could also be evaluated at the same time as quantitative risk assessment has the inherent notion of cost. Moreover, defect models can be used to categorize test cases by the defects they target. Typically test case classification is performed by using coverage metrics to categorize them by the parts of the system and the functionality these part represent [47]. (Semi-)automatically adding the targeted defects using defect models for the categorization may yield a vital contribution to the risk assessment.

We have only presented DELICLA for elicitation and classification of defects for defect models. As discussed in Section 4.2, there are also other methods for the elicitation and classification of defects. A method particularly interesting in the future is orthogonal defect classification (see Section 4.1) as it enables the extraction of the required information from existing bug trackers. A further approach for defect elicitation and classification in the future is using machine learning techniques on defect removals. When software engineers remove defects from systems, they inherently apply the inverse function to α to the system. Thus, by using machine learning on the reverse image and image, the respective α could be created. As the inverse of α represents a form of program healing / repair [115], other application areas in quality assurance exist.

Finally, defect models are only able to capture known and (possibly) predictable defects. Any new and unrelated occurring defects not elicited, classified, described and operationalized will not be found using defect-based quality assurance. Thus, research on new methods to characterize these defects must be performed, if they are to be detected with defect-based quality assurance. Particularly interesting is the characterization of search spaces discussed in Section 7.6. As systems become more and more complex, it is important to find potentially problem-causing and less explored areas of these search spaces.

Bibliography

[1] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001. ISBN: 0769510000 (cited on pages 31–34).

[2] Nadia Alshahwan and Mark Harman. "Coverage and Fault Detection of the Output-uniqueness Test Selection Criteria". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pages 181–192. ISBN: 9781450326452. DOI: 10.1145/2610384.2610413 (cited on page 33).

[3] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. "JPF-SE: A Symbolic Execution Extension to Java PathFinder". In: *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'07. Braga, Portugal: Springer-Verlag, 2007, pages 134–138. ISBN: 9783540712084 (cited on pages 37, 172).

[4] Andrea Arcuri and Lionel C. Briand. "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering". In: *Softw. Test., Verif. Reliab.* 24.3 (2014), pages 219–250. DOI: 10.1002/stvr.1486 (cited on pages 130, 154).

[5] Andrea Arcuri and Lionel C. Briand. "Adaptive random testing: an illusion of effectiveness?" In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 2011, pages 265–275 (cited on page 154).

[6] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. "Random Testing: Theoretical Results and Practical Implications". In: *IEEE Trans. Software Eng.* 38.2 (2012), pages 258–277. DOI: 10.1109/TSE.2011.121 (cited on pages 130, 154).

[7] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. "Use of a Taxonomy of Security Faults". In: *NIST-NCSC*. July 1996, pages 551–560 (cited on page 67).

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions* (Jan. 2004). ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2 (cited on page 67).

[9] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. "Dependability and Its Threats: A Taxonomy". In: *Building the Information Society*. IFIP. Springer, 2004 (cited on page 67).

[10] S.E. Baker and R. Edwards. *How many qualitative interviews is enough?* NCRM, Southampton, Mar. 2012 (cited on pages 74, 77).

[11] Imran Bashir and Raymod A. Paul. “Object-oriented integration testing”. English. In: *Annals of Software Engineering* 8.1-4 (1999), pages 187–202. ISSN: 1022-7091. DOI: 10.1023/A:1018975313718 (cited on page 128).

[12] V. R. Basili, G. Caldiera, H. D. Rombach, Marciniak, and J. John. “Experience Factory”. In: *Encyclopedia of Software Engineering* vol .1 (1994), pages 469–476 (cited on page 161).

[13] Victor R. Basili and Gianluigi Caldiera. “Improve Software Quality by Reusing Knowledge and Experience”. In: *Sloan Management Review* Fall (1995), pages 55–64 (cited on pages 161, 162).

[14] Victor R. Basili and Gianluigi Caldiera. “Improve software quality by reusing knowledge and experience”. In: *MIT Sloan Management Review* 37.1 (1995), page 55 (cited on page 161).

[15] Victor R. Basili, Scott Green, Oliver Laitenberger, Forrest Shull, Sivert Sørumgård, and Marvin V. Zelkowitz. *The Empirical Investigation of Perspective-based Reading*. Technical report. College Park, MD, USA, 1995 (cited on page 36).

[16] Kent Beck. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530 (cited on page 164).

[17] Boris Beizer. *Software testing techniques (2nd ed.)* New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN: 0442206720 (cited on pages 17, 32, 33, 43, 67, 128).

[18] J. S. Bendat and A. G. Piersol. *Engineering applications of correlation and spectral analysis*. John Wiley & Sons, 1980 (cited on page 116).

[19] Alexej Beresnev, Bernhard Rumpe, and Frank Schoven. “Automated Testing of Graphical Models in Heterogeneous Test Environments”. In: *CoRR* abs/1409.6590 (2014) (cited on page 128).

[20] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley, 1999 (cited on page 55).

[21] D. L. Bird and C. U. Munoz. “Automatic generation of random self-checking test cases”. In: *IBM Systems Journal* 22.3 (1983), pages 229–245. ISSN: 0018-8670. DOI: 10.1147/sj.223.0229 (cited on pages 61, 130).

[22] Gregor von Bochmann, Anindya Das, Rachida Dssouli, Martin Dubuc, Abderrazak Ghedamsi, and Gang Luo. “Fault Models in Testing”. In: *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1992, pages 17–30. ISBN: 0444895175 (cited on pages 46, 55).

[23] M. Büchler, J. Oudinet, and A. Pretschner. “Semi-Automatic Security Testing of Web Applications from a Secure Model”. In: *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. June 2012, pages 253–262. DOI: 10.1109/SERE.2012.38 (cited on pages 58, 61).

[24] J. Burnim and Koushik Sen. “Heuristics for Scalable Dynamic Test Generation”. In: *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. Sept. 2008, pages 443–446. DOI: 10.1109/ASE.2008.69 (cited on page 37).

[25] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In: *Proc. OSDI*. 2008, pages 209–224 (cited on pages 37, 61, 97, 99, 116, 119, 120, 122, 129, 171, 172).

[26] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. “Symbolic Execution for Software Testing in Practice: Preliminary Assessment”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pages 1066–1071. ISBN: 9781450304450. DOI: 10.1145/1985793.1985995 (cited on page 37).

[27] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Commun. ACM* 56.2 (Feb. 2013), pages 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795 (cited on page 127).

[28] David N. Card. “Defect Analysis: Basic Techniques for Management and Learning”. In: edited by M. Zelkowitz. Advances in Computers. 2005 (cited on pages 66, 72).

[29] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. “A qualitative comparison of three aspect mining techniques”. In: *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. May 2005, pages 13–22. DOI: 10.1109/WPC.2005.2 (cited on page 56).

[30] K. Charmaz. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. 2006 (cited on page 69).

[31] S. R. Chidamber and C. F. Kemerer. “A Metrics Suite for Object Oriented Design”. In: *IEEE Trans. Softw. Eng.* 20.6 (June 1994), pages 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895 (cited on page 35).

[32] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. “Orthogonal Defect Classification-A Concept for In-Process Measurements”. In: *IEEE Trans. Softw. Eng.* 18.11 (Nov. 1992), pages 943–956. ISSN: 0098-5589. DOI: 10.1109/32.177364 (cited on pages 26, 67).

[33] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. ISBN: 9780262032704 (cited on pages 34, 35).

[34] P. Collingbourne, C. Cadar, and P.H.J. Kelly. “Symbolic Crosschecking of Data-Parallel Floating-Point Code”. In: *Software Engineering, IEEE Transactions on* 40.7 (July 2014), pages 710–737. ISSN: 0098-5589. DOI: 10.1109/TSE.2013.2297120 (cited on page 97).

[35] Peter Collingbourne, Cristian Cadar, and Paul Kelly. “Symbolic Crosschecking of Floating-Point and SIMD Code”. In: *Proc. EuroSys*. 2011 (cited on page 108).

[36] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’77. Los Angeles, California: ACM, 1977, pages 238–252. DOI: 10.1145/512950.512973 (cited on pages 34, 35).

[37] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The ASTRÉE analyzer”. In: *Proc. PLAS*. 2005, pages 21–30 (cited on page 107).

[38] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. 1st edition. Addison-Wesley Professional, 2009. ISBN: 0321534468, 9780321534460 (cited on page 31).

[39] Maximiliano Cristiá, Joaquín Mesuro, and Claudia Frydman. “Integration Testing in the Test Template Framework”. English. In: *Fundamental Approaches to Software Engineering*. Edited by Stefania Gnesi and Arend Rensink. Volume 8411. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pages 400–414. ISBN: 9783642548031. DOI: 10.1007/978-3-642-54804-8_28 (cited on page 128).

[40] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. “Model-based testing in practice”. In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. May 1999, pages 285–294. DOI: 10.1145/302405.302640 (cited on page 19).

[41] I.F. Darwin. *Checking C Programs with Lint*. A Nutshell handbook. O’Reilly & Associates, 1991. ISBN: 9780937175309 (cited on pages 34, 171).

[42] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pages 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0 (cited on page 37).

[43] M. E. Delamaro, J. C. Maidonado, and A. P. Mathur. “Interface Mutation: an approach for integration testing”. In: *Software Engineering, IEEE Transactions on* 27.3 (Mar. 2001), pages 228–247. ISSN: 0098-5589. DOI: 10.1109/32.910859 (cited on page 128).

[44] R.A. Demillo, R.J. Lipton, and F.G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *Computer* 11.4 (Apr. 1978), pages 34–41. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136 (cited on page 33).

[45] W.E. Deming. *Out of the Crisis*. Massachusetts Institute of Technology, Center for Advanced Engineering Study, 2000. ISBN: 9780262541152 (cited on page 160).

[46] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Mike Papadakis, Axel Legay, and Pierre-Yves Schobbens. “A Variability Perspective of Mutation Analysis”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pages 841–844. ISBN: 9781450330565. DOI: 10.1145/2635868.2666610 (cited on page 56).

[47] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. “Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system”. In: *Software Testing, Verification and Reliability* 25.4 (2015), pages 371–396. ISSN: 1099-1689. DOI: 10.1002/stvr.1572 (cited on page 174).

[48] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), 1982 (cited on pages 59, 173).

[49] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007. ISBN: 0321336380 (cited on pages 98, 109).

[50] George Ellis. *Control System Design Guide*. Academic Press, 2004 (cited on pages 41, 136, 138, 140, 141, 147).

[51] A. Elsafi, D.N.A. Jawawi, and A. Abdelmaboud. “Inferring approximated models for integration testing of component-based software”. In: *Software Engineering Conference (MySEC), 2014 8th Malaysian*. Sept. 2014, pages 67–71. DOI: 10.1109/MySec.2014.6985990 (cited on page 128).

[52] M. E. Fagan. “Design and Code Inspections to Reduce Errors in Program Development”. In: *IBM Syst. J.* 15.3 (Sept. 1976), pages 182–211. ISSN: 0018-8670. DOI: 10.1147/sj.153.0182 (cited on pages 34, 36).

[53] Peter Farrell-Vinay. *Manage Software Testing*. Software engineering. CRC Press, 2008. ISBN: 9781420013849 (cited on page 159).

[54] M. Felderer and A. Beer. “Using defect taxonomies for requirements validation in industrial projects”. In: *Requirements Engineering Conference (RE), 2013 21st IEEE International*. July 2013, pages 296–301. DOI: 10.1109/RE.2013.6636733 (cited on pages 61, 171, 173).

[55] M. Felderer and A. Beer. “Using Defect Taxonomies for Testing Requirements”. In: *IEEE Software* 32.3 (May 2015), pages 94–101. ISSN: 0740-7459. DOI: 10.1109/MS.2014.56 (cited on pages 61, 154, 171, 173).

[56] Justin E. Forrester and Barton P. Miller. “An Empirical Study of the Robustness of Windows NT Applications Using Random Testing”. In: *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*. WSS’00. Seattle, Washington: USENIX Association, 2000, pages 6–6 (cited on page 130).

[57] D.A. Garvin. “What does Product Quality really mean?” In: *MIT Sloan Management Review* 26.1 (1984), pages 25–43 (cited on page 66).

[58] B.G. Glaser and A.L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967 (cited on pages 69, 71).

[59] G.C. Goodwin, S.F. Graebe, and M.E. Salgado. *Control System Design*. 2001. ISBN: 9780139586538 (cited on pages 40, 41, 136, 139–141, 143).

[60] J.O. Grady. *System Engineering Planning and Enterprise Identity*. Systems Engineering. Taylor & Francis, 1995. ISBN: 9780849378324 (cited on page 38).

[61] J.F. Gubrium and J.A. Holstein. *Handbook of Interview Research: Context and Method*. SAGE Publications, 2001. ISBN: 9781483365893 (cited on pages 68–71, 82).

[62] Walter J. Gutjahr. “Partition Testing vs. Random Testing: The Influence of Uncertainty”. In: *IEEE Trans. Softw. Eng.* 25.5 (Sept. 1999), pages 661–674. ISSN: 0098-5589. DOI: 10.1109/32.815325 (cited on page 52).

[63] J. D. Hagar, T. L. Wissink, D. R. Kuhn, and R. N. Kacker. “Introducing Combinatorial Testing in a Large Organization”. In: *Computer* 48.4 (Apr. 2015), pages 64–72. ISSN: 0018-9162. DOI: 10.1109/MC.2015.114 (cited on page 61).

[64] Ian G. Harris. “Fault Models and Test Generation for Hardware-Software Covalidation”. In: *IEEE Des. Test* 20.04 (July 2003), pages 40–47. ISSN: 0740-7475. DOI: 10.1109/MDT.2003.1214351 (cited on page 46).

[65] M.J. Harrold and M.L. Soffa. “Selecting and using data for integration testing”. In: *Software, IEEE* 8.2 (Mar. 1991), pages 58–65. ISSN: 0740-7459. DOI: 10.1109/52.73750 (cited on page 128).

[66] Blake Haugen and Jakub Kurzak. “Search Space Pruning Constraints Visualization”. In: *Second IEEE Working Conference on Software Visualization*. 2014 (cited on page 155).

[67] Pamela R. Haunschild, Francisco Polidoro Jr., and David Chandler. “Organizational Oscillation Between Learning and Forgetting: The Dual Role of Serious Errors”. In: *Organization Science* 26.6 (2015), pages 1682–1701. DOI: 10.1287/orsc.2015.1010 (cited on page 160).

[68] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pages 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259 (cited on pages 34, 35).

[69] Dominik Holling. “A Fault Model Framework for Quality Assurance”. In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*. *the authors own work*. 2014, pages 235–236. DOI: 10.1109/ICSTW.2014.44 (cited on page 27).

[70] Dominik Holling, Daniel Méndez Fernández, and Alexander Pretschner. “A Field Study on the Elicitation and Classification of Defects for Defect Models”. In: *To appear in Product-Focused Software Process Improvement - 16th International Conference, PROFES 2015*. *the authors own work*. 2015 (cited on page 66).

[71] Dominik Holling, Andreas Hofbauer, Alexander Pretschner, and Matthias Gemmar. “Profiting from Unit Tests For Integration Testing”. In: *Ninth IEEE International Conference on Software Testing, Verification and Validation, ICST 2016 Proceedings, April 10 - April 15, 2016, Chicago, Illinois, USA*. *the authors own work*. 2016 (cited on page 113).

[72] Dominik Holling, Alexander Pretschner, and Matthias Gemmar. “8Cage: lightweight fault-based test generation for simulink”. In: *ASE 2014*. *the authors own work*. 2014, pages 859–862. DOI: 10.1145/2642937.2648622 (cited on pages 22, 29, 95).

[73] S.E. Hove and B. Anda. “Experiences from conducting semi-structured interviews in empirical software engineering research”. In: *Software Metrics, 2005*. Sept. 2005. DOI: 10.1109/METRICS.2005.24 (cited on page 68).

[74] D. Huizinga and A. Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley, 2007. ISBN: 9780470165164 (cited on page 17).

[75] "IEEE Standard for Software Reviews and Audits". In: *IEEE STD 1028-2008* (Aug. 2008), pages 1–52. DOI: 10.1109/IEEESTD.2008.4601584 (cited on pages 34, 35).

[76] "Information technology – Process assessment". In: *ISO/IEC 33002* (2015) (cited on page 163).

[77] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pages 435–445. ISBN: 9781450327565. DOI: 10.1145/2568225.2568271 (cited on pages 45, 112).

[78] Florentin Ipate and Mike Holcombe. "An integration testing method that is proved to find all faults". In: *International Journal of Computer Mathematics* 63.3-4 (1997), pages 159–178 (cited on page 128).

[79] Yue Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Trans SW Eng* (2011) (cited on page 154).

[80] Zhenyi Jin and A. J. Offutt. "Coupling-based criteria for integration testing". In: *Software Testing Verification and Reliability* 8.3 (1998), pages 133–154 (cited on page 128).

[81] S. C. Johnson. "Lint, a C Program Checker". In: *COMP. SCI. TECH. REP*. 1978, pages 78–1273 (cited on pages 61, 107).

[82] Paul C. Jorgensen and Carl Erickson. "Object-oriented Integration Testing". In: *Commun. ACM* 37.9 (Sept. 1994), pages 30–38. ISSN: 0001-0782. DOI: 10.1145/182987.182989 (cited on page 128).

[83] Seifedine Kadry. "Advanced Automated Software Testing: Frameworks for Refined Practice". In: edited by Izzat Alsmadi. IGI Global, 2012. Chapter On the Improvement of Cost-Effectiveness: A Case of Regression Testing, pages 68–88. DOI: 10.4018/978-1-4666-0089-8.ch004 (cited on page 17).

[84] M. Kalinowski, Guilherme H. Travassos, and David N. Card. "Towards a Defect Prevention Based Process Improvement Approach". In: *SE&AA*. 2008 (cited on pages 71, 78).

[85] Marcos Kalinowski, Emilia Mendes, David N. Card, and Guilherme H. Travassos. "Applying DPPI: A Defect Causal Analysis Approach Using Bayesian Networks". In: *PROFES*. 2010 (cited on pages 66, 68, 72, 74, 75, 78, 83).

[86] Susanne Kandl and Martin Elshuber. "A Formal Approach to System Integration Testing". In: *CoRR* abs/1404.6743 (2014) (cited on page 128).

[87] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pages 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252 (cited on pages 36, 37).

[88] B. Kitchenham and S. P. Pfleeger. “Software Quality: The Elusive Target”. In: *IEEE Software* 13.1 (1996), pages 12–21. ISSN: 0740-7459 (cited on page 66).

[89] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. “Software Fault Interactions and Implications for Software Testing”. In: *IEEE Trans. Softw. Eng.* 30.6 (June 2004), pages 418–421. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.24 (cited on pages 32, 59).

[90] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. New York, NY, USA: Cambridge University Press, 1986. ISBN: 0521246652 (cited on page 35).

[91] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. “A Taxonomy of Computer Program Security Flaws”. In: *ACM Comp* (1994) (cited on page 67).

[92] Carl E. Landwehr, Alan R. Bull, John P. Mcdermott, William, and S. Choi. “A Taxonomy of Computer Program Security Flaws”. In: *ACM Computing Surveys* (1994) (cited on page 67).

[93] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1992. ISBN: 0387822968 (cited on pages 16, 30).

[94] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis transformation”. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. Mar. 2004, pages 75–86. DOI: 10.1109/CGO.2004.1281665 (cited on page 38).

[95] Y. Le Traon, T. Jeron, J. Jezequel, and P. Morel. “Efficient object-oriented integration and regression testing”. In: *Reliability, IEEE Transactions on* 49.1 (Mar. 2000), pages 12–25. ISSN: 0018-9529. DOI: 10.1109/24.855533 (cited on page 128).

[96] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. “IPOG: A General Strategy for T-Way Software Testing”. In: *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. ECBS ’07. Washington, DC, USA: IEEE Computer Society, 2007, pages 549–556. ISBN: 0769527728. DOI: 10.1109/ECBS.2007.47 (cited on page 59).

[97] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. “Classification and Evaluation of Defects in a Project Retrospective”. In: *J. Syst. Softw.* (2002) (cited on pages 67, 74, 75).

[98] W. E. Lewis. *Software Testing and Continuous Quality Improvement*. CRC Press, 2000. ISBN: 9781420048124 (cited on pages 16, 18).

[99] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pages 329–339. ISBN: 9781595939586. DOI: 10.1145/1346281.1346323 (cited on page 57).

[100] Li Ma and Jeff Tian. “Analyzing Errors and Referral Pairs to Characterize Common Problems and Improve Web Reliability”. In: *ICWE*. 2003 (cited on page 67).

[101] Li Ma and Jeff Tian. “Web Error Classification and Analysis for Reliability Improvement”. In: *J. Syst. Softw.* (2007) (cited on page 67).

[102] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt. “Inter-class mutation operators for Java”. In: *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*. 2002, pages 352–363. DOI: 10.1109/ISSRE.2002.1173287 (cited on page 55).

[103] Evan Martin and Tao Xie. “A fault model and mutation testing of access control policies”. In: *Proceedings of the 16th international conference on World Wide Web*. WWW ’07. Banff, Alberta, Canada: ACM, 2007, pages 667–676. ISBN: 9781595936547. DOI: 10.1145/1242572.1242663 (cited on page 46).

[104] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. “Effective test suites for mixed discrete-continuous stateflow controllers”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 2015, pages 84–95. DOI: 10.1145/2786805.2786818 (cited on page 130).

[105] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann, and Claude Poull. “Automated Model-in-the-Loop Testing of Continuous Controllers Using Search”. In: *SSBSE 2013*. 2013, pages 141–157. DOI: 10.1007/978-3-642-39742-4_12 (cited on pages 133, 135, 137, 138, 140, 141, 144, 154, 155, 169, 173).

[106] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. “Search-based automated testing of continuous controllers: Framework, tool support, and case studies”. In: *Information and Software Technology* 57 (2015), pages 705–722. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2014.05.007 (cited on pages 133, 135, 136, 140, 144, 154, 155, 169, 173).

[107] MATLAB. *Consult the Model Advisor*. Natick, Massachusetts: The MathWorks Inc., 2014 (cited on pages 107, 109).

[108] MATLAB. *Simulink Design Verifier*. MathWorks, 2014 (cited on pages 107, 154).

[109] MATLAB. *The Language of Technical Computing*. MathWorks, 1984 - 2016 (cited on pages 38, 90).

[110] T. J. McCabe. “A Complexity Measure”. In: *Software Engineering, IEEE Transactions on* SE-2.4 (Dec. 1976), pages 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837 (cited on page 34).

[111] E. J. McCluskey and Frederick W. Clegg. “Fault Equivalence in Combinational Logic Networks”. In: *Computers, IEEE Transactions on* C-20.11 (Nov. 1971), pages 1286–1293. ISSN: 0018-9340. DOI: 10.1109/T-C.1971.223129 (cited on page 54).

[112] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678 (cited on pages 59, 60, 62).

[113] Harlan D Mills. “Function Semantics for Sequential Programs.” In: *IFIP Congress*. 1980, pages 241–250 (cited on page 47).

[114] MISRA Ltd. *MISRA-C:2004 Guidelines for the use of the C language in Critical Systems*. MISRA, Oct. 2004 (cited on pages 34, 90, 171).

[115] Martin Monperrus. “A Critical Review of ”Automatic Patch Generation Learned from Human-written Patches”: Essay on the Problem Statement and the Evaluation of Automatic Software Repair”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pages 234–242. ISBN: 9781450327565. DOI: 10.1145/2568225.2568324 (cited on pages 62, 174).

[116] L. J. Morell. “A theory of fault-based testing”. In: *Software Engineering, IEEE Transactions on* 16.8 (Aug. 1990), pages 844–857. ISSN: 0098-5589. DOI: 10.1109/32.57623 (cited on pages 16, 17, 20, 47).

[117] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN: 0471469122 (cited on pages 15, 17, 18, 32, 33, 43, 174).

[118] Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. “Using fault injection and modeling to evaluate the performability of cluster-based services”. In: *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS’03. Seattle, WA: USENIX Association, 2003, pages 2–2 (cited on page 57).

[119] S. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2008. ISBN: 9780470382837 (cited on pages 30–33, 173).

[120] Norman S Nise. *Control System Engineering*. John Wiley & Sons, 2007 (cited on page 140).

[121] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. “A Fault Model for Subtype Inheritance and Polymorphism”. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering*. ISSRE ’01. Washington, DC, USA: IEEE Computer Society, 2001, pages 84–. ISBN: 0769513069 (cited on page 56).

[122] T. J. Ostrand and M. J. Balcer. “The Category-partition Method for Specifying and Generating Fuctional Tests”. In: *Commun. ACM* 31.6 (June 1988), pages 676–686. ISSN: 0001-0782. DOI: 10.1145/62959.62964 (cited on pages 15, 32, 58).

[123] M. A. Ould, C. Unwin, and British Computer Society. Working Group on Testing. *Testing in Software Development*. British Computer Society Monographs in Informatics. Cambridge University Press, 1986. ISBN: 9780521337861 (cited on page 128).

[124] N. Ozarin and M. Siracusa. “A process for failure modes and effects analysis of computer software”. In: *Reliability and Maintainability Symposium*. 2003 (cited on page 154).

[125] M. Papadakis and N. Malevris. “Automatic Mutation Test Case Generation via Dynamic Symbolic Execution”. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. Nov. 2010, pages 121–130. DOI: 10.1109/ISSRE.2010.38 (cited on page 62).

[126] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh. “An integrated test generation tool for enhanced coverage of Simulink/Stateflow models”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. Mar. 2012, pages 308–311. DOI: 10.1109/DATE.2012.6176485 (cited on page 130).

[127] Alexander Pretschner. “Compositional Generation of MC/DC Integration Test Suites”. In: *Electr. Notes Theor. Comput. Sci.* 82.6 (2003), pages 1–10. DOI: 10.1016/S1571-0661(04)81020-X (cited on pages 43, 127).

[128] Alexander Pretschner. “Defect-Based Testing”. In: *Dependable Software Systems Engineering*. Edited by Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner. *work in partial collaboration with the author*. 2015, pages 224–245. DOI: 10.3233/978-1-61499-495-4-224 (cited on pages 17, 18, 46–49, 52, 53, 60).

[129] Alexander Pretschner, Dominik Holling, Robert Eschbach, and Matthias Gemmar. “A Generic Fault Model for Quality Assurance”. In: *Proc. MODELS*. *work in collaboration with the author*. 2013, pages 87–103 (cited on pages 17, 18, 46, 52, 53, 112).

[130] Polyspace Code Prover. *Static Analysis with Polyspace Products*. Mathworks, June 2014 (cited on page 107).

[131] Reactis. *Reactis Product Suite*. Reactive Systems, 2014 (cited on pages 61, 107, 128, 154).

[132] W. W. Royce. “Managing the Development of Large Software Systems: Concepts and Techniques”. In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE ’87. Monterey, California, USA: IEEE Computer Society Press, 1987, pages 328–338. ISBN: 0897912160 (cited on page 164).

[133] P. Runeson and M. Höst. “Guidelines for Conducting and Reporting Case Study Research in Software Engineering”. In: *EMSE* (2009) (cited on page 72).

[134] Kurt Schneider. *Experience and Knowledge Management in Software Engineering*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3540958797, 9783540958796 (cited on pages 16, 19, 160).

[135] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130676349 (cited on page 164).

[136] Sina Shamshiri, René Just, José M. Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. “Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. Lincoln, NE, USA, Nov. 2015 (cited on page 18).

[137] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. An Alan R. Apt book. Prentice Hall, 1996. ISBN: 9780131829572 (cited on page 39).

[138] Forrest Shull, Ioana Rus, and Victor Basili. “How perspective-based reading can improve requirements inspections”. In: *Computer* 33.7 (July 2000), pages 73–79 (cited on pages 62, 79).

[139] Sogeti. *TPI Next - Business Driven Test Process Improvement*. UTN Publishers, 2009 (cited on page 163).

[140] Ian Sommerville. *Software Engineering*. International computer science series. Addison-Wesley, 2007. ISBN: 9780321313799 (cited on page 157).

[141] Robert E. Stake. *The Art of Case Study Research*. 1st edition. Tousand Oaks, CA: Sage Publications, 1995. ISBN: 080395767X (cited on page 158).

[142] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler. “Continuous Software Quality Control in Practice”. In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. Sept. 2014, pages 561–564. DOI: 10.1109/ICSME.2014.95 (cited on pages 159, 162).

[143] CMMI Product Team. *CMMI for Development, Version 1.3*. Technical report CMU/SEI-2010-TR-033. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2010 (cited on page 162).

[144] Nikolai Tillmann and Jonathan De Halleux. “Pex: White Box Test Generation for .NET”. In: *Proceedings of the 2Nd International Conference on Tests and Proofs*. TAP’08. Prato, Italy: Springer-Verlag, 2008, pages 134–153. ISBN: 3-540-79123-X, 978-3-540-79123-2 (cited on page 37).

[145] TMMI Foundation. *Test Maturity Model Integration*. 2008 (cited on page 163).

[146] TPT. *TPT - Time Partition Testing*. Piketec Gmbh, 2014 (cited on pages 61, 107, 128, 154).

[147] Mark Utting, Alexander Pretschner, and Bruno Legeard. *A Taxonomy of Model-based Testing*. Technical report. Apr. 2006 (cited on page 34).

[148] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. “Test Input Generation with Java PathFinder”. In: *SIGSOFT Softw. Eng. Notes* 29.4 (July 2004), pages 97–107. ISSN: 0163-5948. DOI: 10.1145/1013886.1007526 (cited on pages 37, 172).

[149] Stefan Wagner. “Defect Classification and Defect Types Revisited”. In: *Defects in Large Software Systems*. ACM, 2008 (cited on page 67).

[150] E. J. Weyuker and B. Jeng. “Analyzing partition testing strategies”. In: *IEEE Trans. Software Eng.* 17.7 (July 1991), pages 703–711. ISSN: 0098-5589. DOI: 10.1109/32.83906 (cited on pages 43–45, 51, 52, 112, 152).

[151] James A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. 1st. Addison-Wesley Professional, 2009. ISBN: 0321636414, 9780321636416 (cited on pages 32, 59).

[152] James A. Whittaker and Michael G. Thomason. “A Markov Chain Model for Statistical Software Testing”. In: *IEEE Trans. Softw. Eng.* 20.10 (Oct. 1994), pages 812–824. ISSN: 0098-5589. DOI: 10.1109/32.328991 (cited on page 34).

[153] Laurie Williams. “Integrating Pair Programming into a Software Development Process”. In: *Proceedings of the 14th Conference on Software Engineering Education and Training*. CSEET ’01. Washington, DC, USA: IEEE Computer Society, 2001, pages 27–. ISBN: 0769510590 (cited on page 35).

[154] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn. “An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pages 242–251. DOI: 10.1109/ICST.2013.35 (cited on page 59).

Indices

List of Figures

List of Tables

List of Listings