A Generic Fault Model for Quality Assurance

A. Pretschner[1], D. Holling[1], R. Eschbach[2], and M. Gemmar[2]

[1] Technische Universität München, Germany
{pretschn,holling}@in.tum.de
[2] itk Engineering, Germany
{robert.eschbach,matthias.gemmar}@itk-engineering.de

Abstract. Because they are comparatively easy to implement, structural coverage criteria are commonly used for test derivation in model- and code-based testing. However, there is a lack of compelling evidence that they are useful for finding faults, specifically so when compared to random testing. This paper challenges the idea of using coverage criteria for test selection and instead proposes an approach based on fault models. We define a general fault model as a transformation from correct to incorrect programs and/or a partition of the input data space. Thereby, we leverage the idea of fault injection for test assessment to test derivation.
We instantiate the developed general fault model to describe existing fault models. We also show by example how to derive test cases.

1 Introduction

Partition-based testing [23] relies on the idea of partitioning the input domain of a program into blocks. For testing, a specified number of input values is usually drawn randomly from each block. The number of tests per block can be identical, or can vary according to a usage profile. Sometimes, the blocks of the partition are considered to be "equivalence classes" in an intuitive sense, namely in that they either execute the same functionality, or are likely to provoke related failures. Code coverage criteria, including statement, branch and various forms of condition coverage, naturally induce a partition of the input domain: in a control flow graph, every path from the entry to the exit node (or back to the entry node) of a program represents all those input data values that, when applied to the program, lead to the respective path being executed. Since this same argument also applies to different forms of condition coverage, coverage-based testing can be seen as an instance of partition-based testing.

More than twenty years ago, Weyuker and Jeng have looked into the nature of test selection based on input domain partitions [25]. They contrasted partition-based testing to random testing; more specifically, to test selection that uniformly samples input values from a program's input domain. To keep the model simple, their criterion to contrast these two forms of testing measures the probability of detecting at least one failure.

They show that depending on how the failure-causing inputs are distributed across the input domain, partition-based testing can be better, the same, or worse

than random testing. By means of example, let us assume an input domain of 100 elements out of which 8 are failure causing. Let us assume a partition of two blocks, and that we can execute two tests.

1. Assume the failure causing inputs are uniformly distributed across the input domain. Assume two blocks of the partition of size 50, each of which contains 4 (uniformly distributed) failure-causing inputs. Drawing one test from each block as opposed to sampling two tests from the entire domain yields the same likelihood of catching at least one failure-causing input: partition-based testing and random testing have the same effectiveness.
2. Assume one block of 8 elements, all of which are failure-causing. The probability of detecting at least one failure with partition-based testing then is 1, certainly more effective than random testing.
3. Assume one block with just 1 element which is not failure-causing, and a second block with 99 elements, out of which 8 are failure-causing. Intuitively, we are wasting one test in the small block; partition-based testing is less effective than random testing.

As we have seen, code coverage criteria induce a partition of the input space. However, in general, we do not know which of the cases (1)-(3) this induced partition corresponds to. In other words, in general, we simply do not know if test selection based on coverage criteria defined over a program's syntax will outperform random sampling. Worse, it may even be less effective than random testing! It is then questionable if this coverage-based approach to test selection can be justified.

Note that limit testing—the idea of picking tests from the "boundaries" of ordered data types—seems to contradict this reasoning. In fact, the contrary is true: limit testing is based on empirical (or at least anecdotal) evidence that things "do go wrong" more often at the ends of intervals than in other places. Therefore, partition-based testing based on limit analysis precisely increases the probability that a failure will be caused by an element of the limit blocks, which puts this scenario, on average, in the context of scenario (2) described above.

Also note that we are careful to distinguish between different methodological usages of coverage criteria. Zhu et al. speak of adequacy criteria that can be used as selection, stopping, or test suite assessment criteria [26]. If tests are *derived* using whatever magic selection criterion but *assessed* using coverage criteria, then coverage criteria have empirically been shown to lead to better quality: they point the tester to parts of the code that has not, or insufficiently, been tested yet [20, 17]. However, if an organization uses any kind of metric for assessment, then there always is the risk that this metric is optimized. In other words, it is possible that after some time, the magic selection criterion will be substituted by the coverage criterion used for test assessment.

Goal. The above remark on limit testing motivates the research presented in this paper. Limit testing relies on a fault model—an intuitive or empirically justified idea that things "go wrong" at the limits of intervals. If partition-based selection criteria are to be useful, then they must be likely to induce

partitions with blocks that have a high likelihood of revealing failures (or have a comparatively high failure rate). Our goal is (1) to precisely capture a general notion of fault models; (2) to show its applicability by instantiating the generic fault model to fault models known from the literature; and (3) to use these instantiated fault models for the derivation of tests that, by construction, have a high likelihood of revealing potential faults—or at least a higher likelihood than random testing.

Contribution. Both in testing practice and in the academic literature, many fault models have been presented. Yet, we are not aware of a unifying understanding of what a fault model, in general, is. This paper closes the gap. It formally defines a general fault model and shows how various fault models from the literature can be seen as instances of our generic fault model. We consider our work to yield a different viewpoint on fault models, partition-based testing and fault injection.

Structure. The structure of this paper is as follows: Section 2 puts our work in context. In Section 3 we introduce our general fault model; Section 4 presents fault models found in the literature and illustrates how they are instances of our general fault model. Section 5 concludes.

2 Related Work

Fault Models. Fault models are a common concept in the field of software and hardware testing. Many fault models have been described in the literature (among others, those of Section 4). These fault models usually describe what the fault is and how to test for it, or provide a method to define the input part of the test cases. Although the descriptions exist, we are not aware of a comprehensive definition that encompasses all existing fault models, and enables their unified instantiation. Martin and Xie state 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 [18]." Harris states that a fault model associates a potential fault with an artifact [12]. 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 to allow an operationalization of fault models in quality assurance. In contrast, the definition by Bochmann et. al. [4] is related to testing. It says 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 similar to our definition of α in the general fault model (see Section 3).

Effectiveness of partition-based testing. Gutjahr observed that the assumption of fixed failure densities (number of failure causing inputs divided by number of elements in the respective block) in Weyuker's and Jeng's model is rather unrealistic [11]: It seems impossible to state for a given program and a given partition what the failure rates of the single block are. He suggests to rather model failure rates as random variables. To do so, he picks one fixed specification and a corresponding input space partition, and considers a (hypothetical) class

of programs written to that specification. On the grounds of knowledge w.r.t. developers, organizations, programming language, implemented functionality, etc., it appears then possible to state, for each block of the partition, what the distribution of failure rates for the programs of the hypothetical class would be. This then gives rise to *expected* failure rates for each block of the partition. Using this construction, he can show that under specific assumptions, if the expected failure rate for all blocks is the same, then partition-based testing is, in general, better than or the same as random testing. While we consider the assumption of equal expected failure rates to be inacceptably strong in practice, we will use Gutjahr's construction as a pillar for defining fault models.

In the same paper, Gutjahr also formulates shortcomings of the model of Weyuker and Jeng. This includes "at least one detected failure" to be a questionable criterion for comparing testing strategies. He suggests alternatives that relate to faults rather than failures and to their severity, and can show that his results are not impacted by these modifications. For simplicity's sake, we stick to the model of Weyuker and Jeng here when assessing the effectiveness of fault models, and leave the formally precise generalization to future work.

Random testing. It is important not to misread the results of Weyuker and Jeng to be advocating uniform random testing; for a recent discussion, see the work of Arcuri et al. [1, 2]. In addition, the quality of tests is not only determined by their probability of detecting faults or failures, but also by their cost. This cost includes many factors, one of which is the effort of determining the fault that led to a failure. In our experience [24, 8], failures provoked by random tests are particularly hard to analyze because the way they execute a program does not follow any "intuitive" logic.

Coverage criteria. In spite of the above argument, one cannot overlook that coverage criteria for test case selection are popular. We conjecture there are two main related reasons. The first is that standards like DO-178B require coverage (MC/DC coverage) for a specific class of software, even though there is little evidence that MC/DC increases the quality of tests in terms of failure detection [13, 9]. Secondly, code (and model) coverage criteria naturally lend themselves to the automated generation of test cases, which then appears a promising field of research and development. We are also aware that coverage criteria provide numbers which are, from a management perspective, always useful. We would like to re-iterate that a-posteriori usages of coverage criteria have been shown to be useful [20, 17].

3 A Generic Fault Model

3.1 Preliminaries

Behavior descriptions and specifications: Programs, models, and architectures all are behavior descriptions. We assume they are 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. Given universal input and output domains I and O, both BDs and specifications give rise to associated semantics $[\![\cdot]\!] : I \to 2^O$.

Correctness of behavior descriptions: 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 $b \models s$.

Faults and Failures: *Faults* are textual (or graphical) differences between an incorrect and a correct BD. Faults may or may not lead to *failures* in the semantics of BDs, which are observable differences between specified and actual behaviors. We do not consider *errors* here, that is, incorrect states of a BD.

Faults as textual mutations; fault classes: Because faults are defined as textual or graphical differences between incorrect and correct BDs, we may assume that there is a textual or graphical transformation $\alpha : \mathcal{B} \to 2^{\mathcal{B}}$ that maps a BD that is correct w.r.t. a specification s into the set of all BDs that are not correct w.r.t. specification s. These latter incorrect BDs in the codomain of α may contain one or multiple faults; these faults may or may not lead to failure; and some of the faults in one BD may be of the same class (see below).

Furthermore, we assume for a subset of all BDs that parts of α can be characterized w.r.t. a set of recurring problems. α_K describes textual or graphical transformations on $\mathcal{B}$ w.r.t. a *fault class* K. Examples for fault classes include textual problems ("> where $\geq$ would have been correct") and typical faults such as problems at the boundaries of loops or ordered data types. Note that this definition does not say how to define K or α_K—it is possible (yet arguably not too useful) to have K capture all possible faults.

$$B_K^s = \bigcup_{b \in \{b' \in \mathcal{B} : b' \models s\}} \alpha_K(b)$$

is the set of all BDs, written to specification s, that contain instances of fault class K. Note that B_K^s is of a hypothetical nature, similar to the construction used by Gutjahr [11]. The elements in $\alpha_K(b)$ may contain one or multiple instances of fault class K.

Failure domains and induced failure domains: BDs have failure domains. Let a BD b be written to a specification s. The *failure domain* of b, $F^{b,s} \subseteq \mathrm{dom}([\![s]\!])$, consists of precisely those inputs that cause incorrect outputs, $i \in F^{b,s}$ iff $i \in \mathrm{dom}([\![s]\!]) \wedge [\![b]\!](i) \not\subseteq [\![s]\!](i)$.

$$\varphi(\alpha_K, s) = \bigcup_{b \in B_K^s} F^{b,s}$$

then defines the failure domain induced by fault class K on specification s. Note that the induced failure domain is independent of any specific BD but rather defined by specification s and fault class K (or rather α_K which is needed for the computation of $F^{b,s}$). $\varphi(\alpha_K, s)$ is the set of all those inputs that potentially provoke a failure related to a fault of class K for *any* BD written to s. The failure domain of every specific BD is a subset of this $\varphi(\alpha_K, s)$, and this failure domain may very well be empty or contain only very few elements of $\varphi(\alpha_K, s)$.

3.2 Fault models

Fault models: Intuitively, a fault model is the understanding of "specific things that can go wrong when writing a BD." In a first approximation, we define fault models to be descriptions of the differences with correct BDs that contain instances of fault class K. More precisely, for a class of specifications $S \subseteq \mathcal{S}$, we define a fault model for class K to be descriptions of the computation of α_K. Sometimes, α_K cannot be provided but the respective induced failure domain can, using the definition of $\tilde{\alpha}_K$ (see below). A *fault model* for class K therefore is a description of the computation of α_K or a direct description of the failure domains induced by α_K, $\varphi(\alpha_K, s)$ for all $s \in S$.

Approximated fault models: In general, a precise and comprehensive definition of the transformations α_K or the induced failure domains $\varphi(\alpha_K, s)$ is not possible and can only be approximated. For a given BD b and a specification s, let $\tilde{\alpha}_K$ describe an approximation of α_K, and let $\tilde{\varphi}$ describe an approximation of the computation of failure domains. 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 $\alpha_K(b')$. Similar to α_K, $\tilde{\alpha}_K$ gives rise to a partition of the input domain of every BD w.r.t. fault class K. Because $\tilde{\alpha}_K$ is an approximation, these induced partitions may or may not be failure domains w.r.t. the considered specification.

Example: Approximations $\tilde{\alpha}_K$ can be over-approximations that may contain mutants that do not necessarily contain faults or even are equivalent to the orginal 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 afterthoughts. Then, the set of BDs created by $\tilde{\alpha}_k$ is smaller than the set of BDs created by α_k.

Approximated induced failure domains and test selection strategies: The intuition behind the input space partitions induced by an approximated $\tilde{\alpha}_K$ is that it computes hypotheses about input blocks with associated failure rates that, overall, lead to good test effectiveness (formally defined below). While φ implicitly computes two blocks for every BD—one where every input is potentially causing a failure related to class K, and another one where

every input certainly is not—the approximations of φ, called $\tilde{\varphi}$ in the sequel, may compute multiple of these blocks. In order to capture relevant fault models from the literature, we augment the definition by stipulating that $\tilde{\varphi}$ computes a number of tests to be drawn from each block (if this number is not known, a constant number n of tests may be assumed for every block). $\tilde{\varphi}$ then is a test selection strategy.

Formally, for the set $\mathcal{J}$ of index sets and appropriate $J \in \mathcal{J}$, we require $\tilde{\varphi}_K : \mathcal{B} \to (J \to 2^I \times \mathbb{N})$ to define a partition of the input domain of a BD together with the number of tests to be drawn from each block.

3.3 Effective Fault Models

Comparing fault models: So far, fault models describe anything that *could go wrong*. They arguably are more useful if they capture problems that *do go wrong* in practice. In order to define useful fault models, we will simplify matters: We will compare testing strategies w.r.t. the likelihood of detecting at least one failure. We can now use the model introduced by Weyuker and Jeng that we described in Section 1. When randomly (uniformly) sampling n elements from the input space of a BD b written to specification s, the probability of causing at least one failure with n tests (with redrawal) is [25]

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

Let $\downarrow_1()$ and $\downarrow_2()$ denote the left and right projections on pairs. Again using the model introduced in Section 1, in terms of partition-based testing, if possibly different numbers of tests are drawn from each block defined by a given selection strategy $\tilde{\varphi}_K(b) = \pi$ with $\pi : J \to 2^I \times \mathbb{N}$ for some index set $J \in \mathcal{J}$, then the likelihood of detecting at least one failure is [25]

$$P_{prt}(b, s, \pi) = 1 - \prod_{j=1}^{|\mathrm{dom}(\pi)|} \Big(1 - \frac{|F^{b,s} \cap \downarrow_1(\pi(j))|}{|\downarrow_1(\pi(j))|}\Big)^{\downarrow_2(\pi(j))}$$

because $\frac{|F^{b,s} \cap \downarrow_1(\pi(j))|}{|\downarrow_1(\pi(j))|}$ is the failure rate of the j-th block of the partition.

Effectiveness: Not every class of faults is relevant for every set of specifications. For instance, rounding issues are unlikely to occur in text processing contexts. We therefore characterize effective fault models for a domain-, company- or technology-specific set of specifications $S' \subseteq \mathcal{S}$ by using a (hypothetical) set of BDs $B_{S'} \subseteq \mathcal{B}$ written to these specifications. We want to capture the fact that a fault model – $\tilde{\alpha}_K$ and the induced $\tilde{\varphi}$, or some provided $\tilde{\varphi}$ – is useful. We do this by defining when a fault model is applicable in the sense that the respective fault class typically happens in practice. When comparing partition-based testing to random testing, it is reasonable to overall use the same number of test cases,

i.e., $n = \sum_{j \in \mathrm{dom}(\pi)} \downarrow_2 (\pi(j))$.

$$n_{S'} = \Big| \Big\{ s \in S' : \big| \{ b \in B_{S'} : P_{prt}(b, s, \tilde{\varphi}_K(b)) \gg P_{rnd}(b, s, n) \} \big| \\ \gg \big| \{ b \in B_{S'} : P_{prt}(b, s, \tilde{\varphi}_K(b)) \not\gg P_{rnd}(b, s, n) \} \big| \Big\} \Big|$$

is the number n'_S of specifications from S' for which the number of BDs that can effectively be tested using partition-based testing via $\tilde{\varphi}_K$ is significantly higher than the number of BDs for which random testing is performing equally well or better. If $\tilde{\varphi}_K$ is defining a fault model or is induced by some α_K, the respective specifications are those to which the fault model is applicable. A fault model is effective if this number is high.

In order to define an effective fault model, we say that $n_{S'}$ must be far larger than the number of specifications from S' to which the fault model is not applicable:

$$n_{S'} \gg \Big| \Big\{ s \in S' : \big| \{ b \in B_{S'} : P_{prt}(b, s, \tilde{\varphi}_K(b)) \gg P_{rnd}(b, s, n) \} \big| \\ \not\gg \big| \{ b \in B_{S'} : P_{prt}(b, s, \tilde{\varphi}_K(p)) \not\gg P_{rnd}(b, s, n) \} \big| \Big\} \Big|.$$

Remark I: This definition of fault models is based on the intuition that a fault model is "better" if it is more generally applicable, that is, if many realistic BDs potentially contain an instance of the respective fault class. If used retrospectively, this notion is thus ideally based on empirical evidence that a specific fault class is relevant in a specific setting. However, it is noteworthy that this idea of a fault model can, without any modifications, also be used prospectively for *one BD* and therefore without empirical evidence about *many BDs*: If it is decided that an instance of a fault class *may* be present in a specific BD, then this fault class can be tested for. The effectiveness of this model is then based on a notion of likelihood that is not based on frequency in the past ("typical fault") but rather on the possibility that the fault may occur. This insight could have been gained on the grounds of a hazard analysis, for instance.

Remark II: We could model failure rates as random variables, in the spirit of Gutjahr's work [11]. We could then compute their expectations, and also the expectations of the probabilities. We do not do this here. Note, however, for the *characterization* of the effectiveness of fault models, it does not really matter which precise numbers we use—the point is rather about comparing the cardinality of different sets of specifications and BDs.

4 Instantiation

To demonstrate the usefulness of our general fault model, we now show existing fault models to be an instance of it. We performed a literature survey and considered existing fault models explicitly stated as such. This list is not intended to be exhaustive but to demonstrate the instantiation process using examples.

The instantiationsare described using the respective α, φ, and their approximations. $\tilde{\alpha}$ describes a fault as a (possibly higher order) mutant. $\tilde{\varphi}$ defines possible input space partitions induced by $\tilde{\alpha}$. We assume that the BDs are correct w.r.t. their specification before the transformation $\tilde{\alpha}$ and incorrect afterwards. Intuitively, this reflects a transformation of *specifications* rather than BDs and can be seen as a transformation of the system model in a model-based engineering approach: test case derivation is then performed at the level of the models. By using our definition of specification, both the correct and incorrect BD can be derived.

Since the failure domain varies due to functions applied to the input before the faulty part of the BD is executed, no general partition of the input space can be given. However, a general schema to derive the partition can be given. The generic definition of φ will partition the input space into one block of inputs for which the output is different after the application of α, and one block for which this is not the case. In practice, the applied functions may only be approximated using the approximation $\tilde{\varphi}$.

α and $\tilde{\alpha}$ are set-valued functions. Let α' and $\tilde{\alpha}'$ denote modifications of these functions that pick one arbitrary element from the codomain of α and $\tilde{\alpha}$, respectively. If α' or $\tilde{\alpha}'$ occur more than once in one definition, then each instance is supposed to pick the same element.

4.1 Stuck-at

The stuck-at fault model [19] 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 creates one block of inputs $\{i : [\![f]\!](i) \neq [\![\alpha'(f)]\!](i)\}$ and one block of inputs $\{i : [\![f]\!](i) = [\![\alpha'(f)]\!](i)\}$ (and it does this for each element of the codomain of α that is picked by α'). Consequently, $\tilde{\varphi}(f) = \{1 \mapsto (\{i : [\![\alpha'(f) \text{ XOR } f]\!](i) == 1\}, n_1), 2 \mapsto (\{i : [\![\alpha'(f) \text{ XOR } f]\!](i) == 0\}, n_2)\}$ where the overall number n of tests is assumed to be fixed, $n = n_1 + n_2$ and $n_1 = n$ if $n \leq |\{i : [\![\alpha'(f) \text{ XOR } f]\!](i) == 1\}|$ and $n_1 = |\{i : [\![\alpha'(f) \text{ XOR } f]\!](i) == 1\}|$ otherwise. 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 that the first block of the input partition is $(a = 0, b = 1, c = 1)$ with one test to be drawn, and the second block of all remaining valuations with $n - 1$ tests to be drawn.

4.2 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 from BD p and induces two blocks of inputs for $\tilde{\varphi}$. The first block of inputs is $\{i : [\![p]\!](i) \neq [\![\tilde{\alpha}'(p)]\!](i)\}$ and the second block is $\{i : [\![p]\!](i) = [\![\tilde{\alpha}'(p)]\!](i)\}$. Note that for the first block of inputs the divisor will be 0 at the point of the division, while for the second it will be different from 0. Thus, $\tilde{\varphi}(p) = \{1 \mapsto (\{i : [\![p]\!](i) \neq [\![\tilde{\alpha}'(p)]\!](i)\}, n_1), 2 \mapsto (\{i : [\![p]\!](i) = [\![\tilde{\alpha}'(p)]\!](i)\}, n_2)\}$ where the overall number n of tests is assumed to be fixed, $n = n_1 + n_2$ and $n_1 = n$ if $n \leq |\{i : [\![p]\!](i) \neq [\![\tilde{\alpha}'(p)]\!](i)\}|$ and $n_1 = |\{i : [\![p]\!](i) \neq [\![\tilde{\alpha}'(p)]\!](i)\}|$ otherwise.

For a BD p_d with input parameter i and $p_d = f_x(i)/f_y(i)$ developed 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 $\tilde{\varphi}(p_d) = \{1 \mapsto (\{i : [\![f_z^{-1}]\!](i) == 0\}, 1), 2 \mapsto (\{i : [\![f_z^{-1}]\!](i) \neq 0\}, n-1)\}$ where the overall number n of tests is assumed to be determined. 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.

4.3 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 [16] 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 4.2 with the two blocks $\{i : [\![p]\!](i) \neq [\![\alpha'(p)]\!](i)\}$ and $\{i : [\![p]\!](i) = [\![\alpha'(p)]\!](i)\}$ also apply here. Consequently, symbolic execution tools are promising for computing φ.

As an example, take a program p_m with 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 $\tilde{\varphi}(p_m) = \{1 \mapsto (\{10\}, 1), 2 \mapsto (I - \{10\}, n-1)\}$ for the input domain I.

4.4 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 4.1: 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 [4]. 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 defines two blocks of input sequences. The first block, $\{i \in \Sigma^* : [\![M]\!](i) \neq [\![\alpha'(M)]\!](i)\}$, defines all those traces that are different in M and $\alpha(M)$ – these are the traces that exhibit faults. The second block is the set of traces for which no difference can be observed: $\{i \in \Sigma^* : [\![M]\!](i) = [\![\alpha'(M)]\!](i)\}$.

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 [3] 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 $S' \supseteq S$.

4.5 Object-oriented testing

In addition, there are fault models catering to subtyping and polymorphism [22] 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 4.2, 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 $p^C_{sda}(x) = f(x)$; $s := x\{s \neq NULL\}$; with post condition $s \neq NULL$ for some instance variable s,

and a method $q^C_{sda}(x,z) = p^C_{sda}(x); if\ z\ then\ \{s \neq NULL\}\ g(s)$; 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 $s \neq NULL$, then the inherited $q^{C'}_{sda}(x,z) = p^{C'}_{sda}(x); if\ z\ then\ \{s \neq NULL\}\ g(s)$; 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 $p^C_{sda}(x); if\ z\ then\ \{s == NULL\}\ g(s)$; 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, test cases representing the second block of $\tilde{\varphi} = \{1 \mapsto (\{(x \mapsto i, z \mapsto false) : i \in \mathbb{N}\}, n-1), 2 \mapsto (\{(x \mapsto i, z \mapsto true) : i \in \mathbb{N}\ and\ s == NULL\ before\ g\ is\ executed\ from\ within\ q^{C'}_{sda}(x))\}, 1)\}$ 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.

4.6 Aspect-oriented testing

The use of AOP has been shown to induce specific faults [6]. 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 4.5).

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 = try$\{f_x(x);\}$ catch (Exception e)$\{f_e(e);\}$ try$\{f_y(x);\}$ catch (Runtime−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 $\tilde{\alpha}'(p_a) = \{try\{f_x(x); f_y(x);\}$ catch (RuntimeException ex$\{f_{ex}(ex);\}$ 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 first block of $\tilde{\varphi}$ must contain 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. The second block contains all other inputs.

4.7 Performance testing

One fault model—there are multiple others—for performance testing [21] 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 4.4.

4.8 Concurrency testing

Fault models used in testing concurrent systems regard atomicity and order violations [15], 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 first block of $\tilde{\varphi}$ contains those inputs for which the output is different when only using a different schedule. The second block contains all other inputs.

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 first block of φ aims to break the assumption by executing all schedules different from s_0 and checking whether the semantics have changed. Thus, the first block of $\tilde{\varphi}$ contains all inputs for which the output is different when only a different schedule is used. The second block contains all other inputs.

4.9 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. [5]. The transformation α is reflected in semantic mutation operators (see Section 4.3) 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 input space partition 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 the first block of $\tilde{\varphi}$ also contains these unknown traces. Thus, $\tilde{\varphi}$ can be constructed in the same way as in Section 4.4.

4.10 Limit testing

The well-known fault model of boundary value analysis (based on the category partition method [23]) is underlying limit testing and, like the fault model of Section 4.5, differs from the rest of the introduced fault models in that the transformation α is unknown (or, analogously, models all those possibilities to get a BD's treatment of limit values wrong). It is, however, possible to create the partition of the input space φ. This creation requires a partition γ, which can use control flow or data flow-based criteria for example. φ then takes the blocks of γ and splits them such that a new block is created for each block boundary including its closest inputs.

An integer block containing the number 1 to 100 would, for example, be split into 3 blocks. The first block contains 0 and 1, the second block contains the number 2 to 98 and the third block contains 99 and 100. Note that, in some cases -1 and 101 are included in the first and last block respectively. This fault model is considered useful, as there is anecdotal evidence that the failure causing inputs are more likely to be in the first and third blocks.

4.11 Combinatorial testing

The fault model of combinatorial, or n-wise, testing [14] 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). The transformation α is again unknown, but the partition of the input space $\tilde{\varphi}$ (i.e. the partition of the 2-way, 3-way or n-way interactions) can be derived by adhering to the parameter combinations. $\tilde{\varphi}$ can be computed using known algorithms on the grounds of Latin squares, for instance. One block of the partition will contain a minimal set of test cases covering all 2-way, 3-way or n-way interactions and the other block will cover all other possible 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).

As an example, reconsider function f from Section 4.1. 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.

5 Conclusion

The contribution of this paper is a general characterization of fault models that encompasses fault models found in practice and the literature. The aim of using

fault models in testing is to derive good test cases. We consider a test case to be good if it detects a potential, or likely, fault.[3] Our fault models consist of syntactic transformations (higher-order, or semantic, mutants), and/or an input space partition. Using several different technologies, this allows us to derive test cases that address the potential faults. We have instantiated our generic fault model to several fault models found in the literature. We do not claim that we capture all fault models but consider our choice to be representative.

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. In addition, we introduced an experience model creating a relationship between classes of systems and class of faults, which we consider helpful in creating adequate fault models, improving risk assessment and test derivation for fault tolerance mechanisms.

Although we have not evaluated the effectiveness of operationalized fault models yet, we see multiple advantages with respect to risk assessment in testing. Coverage criteria and random testing are unable to state whether the system still contains a class of faults after testing. The use of fault models can increase the probability of a particular targeted class of faults to not be present in the system after testing. In addition, fault tolerance can be evaluated by using fault models that target faults handled by the fault tolerance systems. It is also noteworthy that classes of faults in fault models can be associated with the impairment of quality attributes in the system. Thereby, testing using these fault models can reduce the risk of impairment in the final product. Lastly, when using fault models with an inherent transformation, the fault localization effort can be estimated and reduced since the transformations describe what to look for and where. The cost effectiveness of fault models cannot be determined *in general* as it varies from instance to instance. The cost factors involved, however, can be named and are the test level, class of systems (including its context) and the likely class of faults.

We elaborated fault models in testing, but have not limited our general definition to it. Fault models may also be used in other quality assurance techniques such as reviews or inspection performed on non-executable artifacts. Because specifications are BDs themselves, it is straightforward to generalize the transformation α to the level of specifications, thereby allowing it to transform requirements, architecture and design artifacts among others.

We do not believe that the use of fault models is the silver bullet. By its very definition, faults for which no model exists cannot be targeted with our approach. However, for a class of recurring and typical faults in specific contexts, we consider our work to be useful in practice.

Future Work. Our current research focuses on creating a prototype to (semi-) automatically generate test cases using underlying fault models, using the different mentioned technologies. In the future we wish to complete this prototype and to empirically evaluate our introduced fault models with it. In

[3] In fact, it should do so with good cost effectiveness, including debugging cost, but this is not the subject of this paper.

addition, we are also exploring the sources of experience data for fault models. We see many promising areas inside and outside software testing. A promising method to gain knowledge common faults in software testing is orthogonal defect classification [7]. Using this method faults can be classified according to criteria, which in turn can be leveraged to select faults to test for. We also plan to investigate whether fault models can be created from faults found during reviews and inspections [10]. This investigation will particularly focus on the comparison of effectiveness of quality assurance techniques per created fault model.

References

1. Arcuri, A., Iqbal, M., Briand, L.: Random testing: Theoretical results and practical implications. Software Engineering, IEEE Transactions on 38(2), 258–277 (2012)
2. Arcuri, A., Briand, L.: Formal analysis of the probability of interaction fault detection using random testing. IEEE Transactions on Software Engineering 38(5), 1088–1099 (2012)
3. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Object Technology Series, Addison-Wesley (1999)
4. Bochmann, G.v., Das, A., Dssouli, R., Dubuc, M., Ghedamsi, A., Luo, G.: Fault models in testing. In: Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV. pp. 17–30. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (1992), http://dl.acm.org/citation.cfm?id=648126.747577
5. Buchler, M., Oudinet, J., Pretschner, A.: Semi-automatic security testing of web applications from a secure model. In: Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. pp. 253–262 (June 2012)
6. Ceccato, M., Tonella, P., Ricca, F.: Is aop code easier or harder to test than oop code? In: In on-line Proceedings of the First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005) (March, 2005)
7. Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.Y.: Orthogonal Defect Classification-A Concept for In-Process Measurements. IEEE Trans. Softw. Eng. 18(11), 943–956 (Nov 1992), http://dx.doi.org/10.1109/32.177364
8. Ciupa, I., Pretschner, A., Oriol, M., Leitner, A., Meyer, B.: On the number and nature of faults found by random testing. Softw. Test., Verif. Reliab. 21(1), 3–28 (2011)
9. Dupuy, A., Leveson, N.: An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In: Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th. vol. 1, pp. 1B6/1–1B6/7 vol.1 (2000)
10. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Syst. J. 15(3), 182–211 (Sep 1976), http://dx.doi.org/10.1147/sj.153.0182
11. Gutjahr, W.J.: Partition testing vs. random testing: The influence of uncertainty. IEEE Trans. Softw. Eng. 25(5), 661–674 (Sep 1999), http://dx.doi.org/10.1109/32.815325
12. Harris, I.G.: Fault models and test generation for hardware-software covalidation. IEEE Des. Test 20(04), 40–47 (Jul 2003), http://dx.doi.org/10.1109/MDT.2003.1214351

13. Heimdahl, M., Whalen, M., Rajan, A., Staats, M.: On mc/dc and implementation structure: An empirical study. In: Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th. pp. 5.B.3–1–5.B.3–13 (2008)
14. Kuhn, D.R., Wallace, D.R., Gallo, Jr., A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. 30(6), 418–421 (Jun 2004), http://dx.doi.org/10.1109/TSE.2004.24
15. Lu, S., Park, S., Seo, E., Zhou, Y.: 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. pp. 329–339. ASPLOS XIII, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1346281.1346323
16. Ma, Y.S., Kwon, Y.R., Offutt, J.: Inter-class mutation operators for java. In: ISSRE. pp. 352–366 (2002)
17. Malaiya, Y., Li, M., Bieman, J., Karcich, R.: Software reliability growth with test coverage. Reliability, IEEE Transactions on 51(4), 420–426 (2002)
18. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proceedings of the 16th international conference on World Wide Web. pp. 667–676. WWW '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1242572.1242663
19. McCluskey, E., Clegg, F.W.: Fault equivalence in combinational logic networks. Computers, IEEE Transactions on C-20(11), 1286–1293 (Nov 1971)
20. Mockus, A., Nagappan, N., Dinh-Trong, T.T.: Test coverage and post-verification defects: A multiple case study. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. pp. 291–301. ESEM '09 (2009)
21. Nagaraja, K., Li, X., Bianchini, R., Martin, R.P., Nguyen, T.D.: 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. pp. 2–2. USITS'03, USENIX Association, Berkeley, CA, USA (2003), http://dl.acm.org/citation.cfm?id=1251460.1251462
22. Offutt, J., Alexander, R., Wu, Y., Xiao, Q., Hutchinson, C.: A fault model for subtype inheritance and polymorphism. In: Proceedings of the 12th International Symposium on Software Reliability Engineering. pp. 84–. ISSRE '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=851028.856258
23. Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating fuctional tests. Commun. ACM 31(6), 676–686 (Jun 1988), http://doi.acm.org/10.1145/62959.62964
24. Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One evaluation of model-based testing and its automation. In: Proceedings of the 27th international conference on Software engineering. pp. 392–401. ICSE '05 (2005)
25. Weyuker, E., Jeng, B.: Analyzing partition testing strategies. Software Engineering, IEEE Transactions on 17(7), 703 –711 (jul 1991)
26. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (Dec 1997)