Defect-Based Testing

Alexander Pretschner

Department of Informatics, Technische Universität München, Germany

Abstract. What is a good test case? One that reveals potential defects with good cost-effectiveness. We provide a generic model of faults and failures, formalize it, and present its various methodological usages for test case generation.

Keywords. Testing, model-based testing, risk-based testing, cost effectiveness, quality assurance, fault models, failure models, fault hypothesis, fault injection

1. Introduction

In his classic book [1, p.20], Myers stipulates a good test case to be “one that has a high probability of detecting an as yet undiscovered error.” Similar to the (managerial) perspective that “a good test case reveals a defect”, this definition necessitates the presence of a defect for a test case to be good. For a hypothetically correct program without any defects, there would hence be no good test cases. It then appears in order to modify this definition as follows: A good test case is one that reveals a potential defect. This solves the problem of the correct program and assures that test cases can be good even if a respective defect is not present. However, in practice the cost of the defect plays an important role, and it seems acceptable to modify our definition once more: A good test case reveals a potential defect with good cost-effectiveness. The cost of testing pertains to the cost of designing, executing, evaluating and managing the test suite; to the cost of fault localization and debugging; and to the cost of remaining defects in the field.

Our definition arguably is intuitive. However, it has a disadvantage: it is not directly operationalizable in the sense that it would provide a recipe for deriving tests. In this article, we provide such an operationalization on the grounds of defect models. Defect models capture the notion of potential defects and are designed for operationalization with test case generators.

Defects can be failures, errors or faults. With the exception of Myer’s quotation above, we stick to the following terminology in this article: A failure is the observed difference between actual and intended behaviors. This necessitates a program to be executed. An error is an incorrect internal state of the program. Because of error masking or fault tolerance mechanisms such as redundancy, errors may but do not necessarily lead to failures (but there is no failure without an error). Faults are the textual representation of what went wrong in a piece of code or any other artifact of the development process (let us call them behavior descriptions in the following): an incorrect line or an incorrect block of lines in code; incorrect statements in a user requirements specification, and so on. Because faults can be located in dead code, they do not necessarily provoke an error, but there is no error without a fault.
Our defect models capture both fault models and failure models. As we will see, the same formalism could be used to describe error models as well, but we refrain from doing so for the sake of simplicity.

As we have seen, faults are incorrect parts of a behavior description. Because we want to operationalize knowledge of faults for test case generation in various forms, it is convenient to define fault models as mappings from correct to incorrect behavior descriptions. This is in line with and generalizes Morell’s definition [2] who, theoretically speaking, proposes to perform fault-based testing by replacing single statements by other statements.

However, intuitively speaking, many testing approaches dubbed fault-based are in fact failure-based. We therefore complement the definition of fault models to be mappings from correct into incorrect programs by the additional notion of failure models which directly describe incorrect behaviors. Among other things, this is motivated by the defect model that justifies n-wise, or combinatorial, testing [3]: it is the interaction of a few rather than all input parameters which lead to a failure. This is a property of the domain rather than the textual representation of the program under test—and yet, it intuitively captures one essential part of what a defect model should be. As we will see, fault models naturally induce failure models (which in practice need to be approximated). However, as the example of the defect model underlying combinatorial testing shows, there are failure models for which we do not have fault models, or for which useful fault models may not even exist. In Section 3, we provide further examples for this phenomenon.

Contribution. This work is based on Morell’s cited work [2] and considerations published earlier [4]. The original contributions of this paper include (1) the explicit distinction between fault and failure models as well as (b) a description of the various possible methodological instantiation of our defect model.

We contrast our work with related work throughout the paper.

Structure. The remainder of this paper is organized as follows. In Section 2, we introduce faults and failures and show how to turn them into fault and failure models, argue that approximations are necessary, align our defect model with considerations about effectiveness and cost-effectiveness, and conclude with a few examples of test selection strategies that are not defect-based. In Section 3, we show how the theoretical framework has been instantiated in various forms, and how precisely these instantiations relate to the formalism. In Section 4, we discuss methodological issues pertaining to how to obtain and manage defect models. We conclude in Section 5.

2. Defects and Defect Models

2.1. Faults and Fault Models

We define faults and failures for possibly non-deterministic behavior descriptions, denoted by $\mathcal{B}$. Behavior descriptions can be code, models, architectures, or any kind of documents, provided they give rise to a semantics that maps input to output. Let us fix input and output domains $I$ and $O$, respectively. The semantics of behavior descriptions is provided by a partial function

$$\llbracket \cdot \rrbracket : \mathcal{B} \rightarrow (I \rightarrow 2^O),$$
where we require $\forall i \in \text{dom}([b]). [b](i) \neq \emptyset$: non-definedness is captured by partiality.

In the following, we introduce fault models and failure models. Fault models rely on the inner structure of a behavior description, and therefore can be associated with white-box testing strategies. Failure models, in contrast, exclusively rely on the externally visible interface of a system, and can therefore be associated with black-box testing strategies.

### 2.1.1. Definition

Faults are incorrect behavior descriptions in the sense that there are syntactic elements in a given behavior description that need to be replaced. Depending on the specification formalism, these syntactic elements can be single lines of code, blocks of code, blocks in a Simulink diagram, transitions in a state machine, etc. To capture that notion of incorrectness, we need to relate to a valid specification, valid in the sense that it encodes the actual requirements.

Faults are instances of fault classes. A fault class $K$, drawn from a set of failure and fault classes $\mathcal{K}$, 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. Sections 2.1.2 and 3 elaborate on these examples.

A fault model for fault class $K$ is a mapping $\alpha_K$ from a correct behavior description and a valid specification to a set of incorrect behavior descriptions,

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

The fact that $A_K$ operates on a (hypothetical) correct implementation is captured by the following requirement. Let $j$ be an implementation and $s$ a valid specification. We thus require

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

i.e., the first argument specifies an implementation that is correct w.r.t. the specification provided as the second argument.

$A_K$ is a transformation from correct to incorrect behavior descriptions and can be seen as a fault injection operator. A fault of class $K$ can of course occur multiple times in a given implementation. In order not to complicate formalism, we assume a function $\alpha_K$ that picks one arbitrary element from the range of $A_K$.

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

In the following, we will use $\alpha_K$ instead of $A_K$, knowing that a fault model for class $K$ describes many behavior descriptions with arbitrarily many instances of a fault of class $K$. Similarly, a class $K$ can be composed of multiple fault classes.

Faults in a given behavior specification (correct behavior descriptions with faults injected via $\alpha_K$) may or may not lead to failures. For instance, no failure will occur if the behavior description is a piece of code and the fault is located in dead code.
2.1.2. Examples

Examples for fault models and respective formalizations of \( \alpha_K \) are provided in the literature [4]. They include the following.

- A typical hardware defect is the stuck-at fault [5] where an input signal is constantly zero or one. Injecting this fault replaces a line or a gate in a circuit by a constant.
- Typical programming defects are division-by-zero and null pointer dereferencing. These can be modeled as faults where checks for a null pointer or a denominator equaling zero are removed. They can also be seen as failures by specifying that respective fatal exceptions should be thrown.
- Mutation testing aims at assessing the quality of test suites. To this end, small changes are introduced into a program. For instance, a \(<\) is replaced by \(\leq\), \(\text{protected}\) is replaced by \(\text{public}\), and so on. The existing test suite is run on these modified programs, called mutants, where the original program serves as an oracle. The ratio of detected mutants is a measure for the quality of the test suite. The applicability of mutation testing relies on two crucial hypotheses, the competent programmers hypothesis (programmers only introduce small defects of the said kind), and the coupling hypothesis which hypothesizes a correlation between the small changes and real defects, which is obvious for some mutation operators [6]. The mutation operators are a fault model in the sense of this paper.
- In terms of testing finite state machines, fault models describe unnecessary states and incorrect transitions. These can be injected by respective fault models \( \alpha_K \).
- Many things can go wrong when doing object-oriented testing [7]. When seeing objects as state machines, the above fault models can directly be applied. In terms of polymorphism and inheritance [8], let us focus on one example, a state definition anomaly that violates some class invariant in the definition of a method in a subclass. The fault model simply adds an assertion as the last line to the method; this assertion makes sure that the property is violated, e.g. by negating the assertion.
- Concurrency tends to provoke many typical defects, e.g. [9]. A fault model removes synchronizations.

2.2. Failures and Failure Models

2.2.1. Definition

Failures are differences between actual and intended behaviors. Failures are instances of failure classes. A failure class \( K \), drawn from the above set of failure and fault classes \( \mathcal{K} \), is a description of what can go wrong while executing a (hypothetical) set of behavior descriptions, as opposed to what can gone 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. Again, Sections 2.2.2 and 3 elaborate on these examples.

A failure model is a description of those behaviors that are incorrect, i.e., a characterization of the (generally non-computable) set

\[
\varphi(j,s) = \{ (i, [j](i)) : i \in \text{dom}(s) \land [j](i) \not\subseteq [s](i) \}.
\]
Failure models give rise to a partitioning of the input domain in two blocks. One block contains the input elements that lead to incorrect output, the so-called failure domain. The other block contains the elements that lead to correct output.

This definition does not distinguish between different failures that are the result of possibly different faults. It is useful to see that fault models $\alpha_K$ induce failure models. Let $(j, s) \in \text{dom}(\alpha_K)$. Then

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

is the failure model induced by $\alpha_K$. For each fault model, there is an induced failure model. We have already stated that there are also failure models without known fault models.

Note that the exact computation of $\varphi$ or $\varphi_K$ is impossible in general.

2.2.2. Examples

- The above defects in state machines can also be modeled as failure models where $\varphi_K$ generates test cases that satisfy various coverages of the state machine, including the W, U, I etc. methods, e.g. [10].
- Limit, or boundary, testing is an extremely successful test selection technique. In a behavior description, some data types are ordered, and for most of them relevant intervals are naturally defined by the behavior description: specifications, models, and implementations. Limit testing is based on the observation that failure inducing inputs often are clustered around the left or right boundaries of an interval—and this is where test cases are selected. The way we have presented limit testing, we obviously have a failure model. However, there also is a fault model underlying limit testing: in loops, for instance, the upper or lower bounds sometimes are not specified correctly, which comes in the form of off-by-one faults. A fault model for limit testing can then add or subtract 1 from the upper bounds, which is a transformation $\alpha_K$ as introduced above.
- We have introduced the idea of combinatorial testing [3] in the introduction: failures are the result of the interactions of few rather than all parameters. There are many tools and algorithms that ensure pairwise coverage on a multi-dimensional input domain. In general, there is not just one test suite that satisfies n-wise coverage, so these tools tend to pick one solution at random, or provide multiple solutions. This test case generator $\varphi_K$ implements the failure model that underlies n-wise testing; test cases are picked regardless of the structure of the behavior description.\footnote{The question of the failure model underlying n-wise testing relates to a property of a behavior description, as the casting as failure model suggests, or to a property of the domain in which the behavior description is located. Answering this question, and the ramifications it may have for our theory of defect-based testing, is the subject of future work.}

2.3. Approximations

In practice, $\alpha_K$ and $\varphi_K$ normally cannot be precisely specified—specifically, if we could compute $\varphi_K$, there would not be a need for testing. However, these mappings can be, and
in practice are, approximated by functions $\tilde{\alpha}_K$ and $\tilde{\phi}_K$. Let us postpone the formalization of our specific notion of approximation to Section 2.4. We will there see that a defect model is effective (and hence $\phi_K$ is a good approximation of $\phi_K$) if the average failure rate of the entire behavior description under validation is far smaller than the average failure rate in the approximation $\tilde{\phi}_K$. Put another way, there need to be “sufficiently large” overlaps between both domains and codomains of functions $\alpha_K$ and $\tilde{\alpha}_K$ on the one side, and between both domains and codomains of $\phi_K$ and $\tilde{\phi}_K$.

In terms of fault models, the intuition behind the need for approximations is that it seems hard to describe all those regions of a behavior description that would be mapped to all possible regions with all possible injections of fault class $K$. Similarly, in terms of failure models, the function defined by the induced failure model usually cannot be efficiently computed (or cannot be computed at all [2]). And even if the failure model is provided explicitly, the behavior descriptions often are too complex to be amenable to a fully automated analysis with approaches such as symbolic execution or model checking. The respective tools use so-called concolic execution or predicate abstractions for model checking which results in the need of computing approximations $\tilde{\phi}_K$ rather than $\phi_K$ itself.

In the following, we acknowledge that in practice, $\alpha_K$ and $\phi_K$ can rarely be specified and/or computed to capture the entirety of a fault class $K$. That said, however, if the approximations are not too coarse, we see good reasons trade precision for computability and scalability if this allows us to actually compute defect-based test suites.

Notice that the approximation $\tilde{\phi}_K$ can be seen as a selection strategy for tests that target faults of class $K$. Also notice that in contrast $\phi_K$ which is computed on the grounds of $\alpha_K$, we do not require $\tilde{\phi}_K$ to be necessarily computed using $\tilde{\alpha}_K$. In other words, failure models $\phi_K$ can also be explicitly provided. In this case, of course, the oracle problem materializes: our definition of failure models and approximated failure models also includes the expected output, i.e., the oracle, which often is hard to specify. However, without further formal elaboration, the oracle can often be formulated as an assertion with various degrees of precision, ranging from “no crash” to characterizations sets of values (e.g., “positive number”) to concrete single values. As we will see in Section 3, there are some (approximated) failure models for which we do not fully understand the underlying faults yet.

2.4. Effectiveness: tackling typical defects

Fault models hypothesize what could go wrong. In order to be useful, these hypotheses should be close to what does go wrong in a specific domain in practice. Remember that approximate failure models can be seen as test selection strategies. To simplify matters, let us compare these defect-based test selection strategies to random testing, and require that effective defect models perform significantly better than random testing. This idea has been proposed in the literature before [4].

To do so, we use a model introduced by Weyuker and Jeng [11] that contrasts test selection strategies w.r.t. their ability to detect at least one failure (for a criticism of this model see, among others, the work of Gutjahr [12]). Let us randomly (uniformly) sample $n$ elements from the input space of a behavior description $b$ that has been written to specification $s$.

Given a fault class $L$ and using the size of the failure domain $\text{dom}(\phi_L(j,s))$, the probability of causing at least one failure with $n$ tests (with redrawal) as a result of some fault class $L$ (which may well be an aggregation of multiple possible faults here) is [11]
\[
P_{\text{rnd}}(b, s, n, L) = 1 - \left(1 - \frac{|\phi_L(b, s)|}{|\text{dom}([b])|}\right)^n,
\]

under the (unrealistic yet irrelevant) assumption of uniformly distributed failure causing inputs. We will contrast this probability with the probability of provoking at least one failure with defect-based testing as follows. If we knew how to compute \(\phi_L(b, s)\), either directly or by referring to an exact fault model, then there would not be any need for testing. However, we have argued above that we can only hope for approximations \(\tilde{\phi}_L\) and \(\tilde{\alpha}_L\).

When doing defect-based testing, we only sample from \(\tilde{\phi}_L(b, s)\), and we therefore compute the probability of provoking at least one failure with defect-based testing as

\[
P_{\text{defect}}(b, s, n, L) = 1 - \left(1 - \frac{|\tilde{\phi}_L(b, s) \cap \phi_L(b, s)|}{|\tilde{\phi}_L(b, s)|}\right)^n.
\]

A defect model for defect class \(L\) is effective for (or applicable to) a given behavior description \(b\) and a given specification \(s\) if random testing is significantly worse than defect-based testing. Formally, this is the case iff

\[
P_{\text{rnd}}(b, s, n, L) \ll P_{\text{defect}}(b, s, n, L),
\]

or equivalently, iff

\[
\frac{|\phi_L(b, s)|}{|\text{dom}([b])|} \ll \frac{|\tilde{\phi}_L(b, s) \cap \phi_L(b, s)|}{|\tilde{\phi}_L(b, s)|},
\]

which matches 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 \(\tilde{\phi}_L\). If this equation holds, \(\tilde{\phi}_L\) is a “good” approximation of \(\phi_L\) in the context of one specific behavior description that has been written w.r.t. one specific specification, as anticipated in Section 2.3. As one would expect, this equation is independent of the number of test cases being drawn. Consequently, in the following, we will assume some \(n\) to be fixed, and omit it as a parameter for \(P_{\text{rnd}}\) and \(P_{\text{defect}}\).

It seems intuitive that not every class of faults is relevant for every set of specifications. Divisions by zero are unlikely to occur in text processing contexts. 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 \(S\). For instance, a program implementing quicksort cannot realistically be expected to have been written w.r.t. the specification of an engine control. Our construction is the same as Gutjahr’s when he models failure rates as random variables rather than deterministic values [12]. According to [4]

\[
n_S = \left|\left\{s \in S : \left|\left\{b \in B_S : P_{\text{prl}}(b, s, L) \gg P_{\text{rnd}}(b, s, L)\right\}\right| \gg \left|\left\{b \in B_S : P_{\text{prl}}(b, s, L) \nless P_{\text{rnd}}(b, s, L)\right\}\right|\right\}\right|
\]

is the number of specifications from \(S\) for which the number of behavior descriptions that can effectively be tested using defect-based testing via \(\tilde{\phi}_L\) is significantly higher than the
number of behavior descriptions for which random testing is performing at least equally well. A specific defect model is effective if this number is “high” for a given class of specifications \( S \) that define a domain of interest.

Finally, in order to define an effective defect model, we say that \( n_S \) must be far larger than the number of specifications from \( S \) for which the defect model is not effective (this defines the notion of \( n_S \) being “high”):

\[
n_S \gg \left| \left\{ s \in S : \left\{ b \in B_S : P_{\text{defect}}(b, s, L) \gg P_{\text{rnd}}(b, s, L) \right\} \gg \left\{ b \in B_S : P_{\text{defect}}(b, s, L) \not\gg P_{\text{rnd}}(b, s, L) \right\} \right\} \right|.
\]

This definition of defect models is based on the 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. If used retrospectively, this notion is thus ideally based on empirical evidence that a specific defect class is relevant in a specific setting.

However, it is noteworthy that this idea of a defect model can, without any modifications, also be used prospectively for one behavior description and therefore without empirical evidence about many behavior descriptions: If it is decided that an instance of a fault class may be present in a specific behavior description, 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.

Finally, also note that we could model failure rates as random variables, in the spirit of Gutjahr’s work [12]. 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 behavior descriptions.

We are aware that because this definition uses \( \varphi_L \), it is theoretical in that it does not allow us to compute, for a given fault or failure model, if it is effective. However, this formalization does provide a precise definition of effectiveness that we can (a) strive to achieve and (b) try to assess using empirical experiments.

2.5. Cost-effectiveness: tackling relevant defects

While the above considerations do consider the frequency of specific defects, they do not distinguish between failures that are considered “grave” (e.g., crash of a system with resulting loss of money or life) and failures that are a mere nuisance (e.g., typos or GUI problems). Theoretically, our defect model can easily be extended by notions of risk-based testing as follows.

For simplicity’s sake, let us first assume that the effect of failures can be classified, e.g., by using a set \( \mathcal{C} = \{ \text{uncritical}, \text{annoying}, \text{critical}, \text{catastrophic} \} \). This classification simplifies matters (1) seemingly because we see failures as single output values and (2) actually because in most cases, this decision cannot be taken without further context information. In terms of (1), it is noteworthy that we did not impose any constraints on input set \( I \) and output set \( O \), so they can range over (sets of) single values—-transformative
systems; discrete sequences of values—discrete reactive systems; or continuous streams of variables—continuous reactive systems. This allows us to assign a criticality assessment not only to pairs of single input values and sets of output values, but also to histories, or streams, of \((i,O)\) pairs.

In any case, we can assign fault and failure classes to criticalities via a function

\[ c : \mathcal{H} \times 2^B \rightarrow C \]

which needs to be parameterized by a set of behavior descriptions. This is because different fault or failures will have different criticalities for different behavior descriptions. Then we only need to observe that our fault and failure classes \(\mathcal{H}\) can be composed of other fault and failure classes. This means that we are free to define a fault and failure class \(\mathcal{H}' \subseteq \mathcal{H}\) for a set of behavior descriptions \(B \subseteq \mathcal{B}\) under scrutiny such that

\[ c(k,B) \in \{\text{catastrophic, critical}\} \text{ for all } k \in \mathcal{H}'. \]

Once again theoretically speaking, this allows us to prioritize our testing activities, and thus to perform risk-based testing, in a very simple manner.

This notion of risk tackles one aspect of the cost of testing: the cost of remaining defects in the field. It does not cover other forms of cost yet, namely those of deriving, running, evaluating and maintaining tests, as well as the debugging cost. Theoretically, it is trivial to assign a cost value, drawn from set \(\mathcal{M}\), to any element of \(\phi_K(j,s)\) via some function

\[ \text{cost} : I \times 2^O \rightarrow \mathcal{M}, \]

and then predict the cost of some subset \(T \subseteq \phi_K(j,s)\) for a relevant class of faults \(K\) by

\[ \sum_{(i,O) \in T} \text{cost}((i,O)), \]

or find suggestions for sets \(T' \subseteq \phi_K(j,s)\) such that

\[ \sum_{(i,O) \in T'} \text{cost}((i,O)) < M \]

for some upper cost limit \(M\).

Both in terms of risk and cost, we are aware that these considerations are more of a theoretical than a practical nature. This is because in practice, it is usually extremely hard to characterize (that is: anticipate!) those (sequences of) inputs and outputs that lead to critical or catastrophic consequences. Similarly, predicting the cost of debugging, for instance, is more an art than a science.

2.6. Counterexamples

One may wonder if there are test selection strategies that are not defect-based. We have seen one such example: Uniform random testing certainly is not defect-based. While the practical usefulness is a subject of debate [13], particularly when considering the cost
of test minimization, fault localization, and debugging, there is another selection criterion that is more popular in practice: partition-based testing. Partition-based testing uses, among other things, requirements [14], coverage criteria, or knowledge about historical or hypothesized defects to partition the input space into blocks and then samples test cases uniformly (or randomly by maximizing some kind of data coverage in the context of adaptive random testing [15,16]) from each of the blocks.

Since we want to understand the nature of test selection that is not based on defects, we obviously need to restrict ourselves to partition strategies that are not based on defects. A partition of a behavior description’s input domain \( I \) is a set of pairwise disjoint sets \( I_1, \ldots, I_m \) with \( I = \bigcup_{j=1}^{m} I_j \). In the sequel, we call these sets blocks of the partition. If possibly different numbers of tests \( n_j \) are uniformly drawn from each block \( j \), then the likelihood of detecting at least one failure is [11]

\[
P_{\text{prt}}(b, s) = 1 - \prod_{j=1}^{m} \left( 1 - \frac{|I_j \cap \varphi^1(b, s)|}{|I_j|} \right)^{n_j}
\]

where \( \varphi^1(b, s) \) is the projection of \( \varphi(b, s) \) to the first component, that is, the input part. \( \frac{|I_j \cap \varphi^1(b, s)|}{|I_j|} \) clearly is the average failure rate of the \( j \)-th block.

Weyuker and Jeng have shown that \( P_{\text{prt}} \) can be larger than, smaller than, and equal to \( P_{\text{rnd}} \), depending on how the blocks of the partition are chosen [11,4]. They also non-exclusively show conditions under which either of the strategies is better. In any case, in order to predict if testing based on one specific partition is more effective than random testing, the failure rates of the different blocks need to be known in advance. This, however, normally is not the case (and if it was, these blocks would naturally be tested more intensely than the others).

Gutjahr [13] extends Weyuker and Jeng’s model by modeling failure rates as random variables, and by using different criteria for comparing selection strategies, including number of faults, criticality, etc. Under the assumption that the expectations of the failure rates are similar, he can show that partition-based testing is, in general, better than random testing. The assumption of equal expectations is crucial, of course.

Now, if the choice of blocks is not based on defect hypotheses (which is the case, for instance, for limit testing), then it cannot be predicted if partition-based testing will be better than random testing. From an engineer’s perspective, if any test selection strategy sometimes is better than, sometimes worse than, and sometimes equal to random testing, then the effort of building partitions can hardly be justified.

As an example, among others, this is the case when selecting tests based on coverage of a behavior description that does not explicitly encode defects, e.g. code coverage or coverage on state machines that model systems [4]. It is generally important to distinguish the methodological usage of adequacy criteria [17]: coverage used for test selection may yield different results than coverage used for test assessment, in which latter case the quality of a test suite may stem from a superior selection criterion. However, there are no guarantees in either direction [18,19,20], and there always is a problem that every metric in an organization will eventually be optimized, known as the “meeting the numbers” phenomenon.

Note that the above is not to be seen as advocating random testing. While indeed often effective at provoking failures in practice, fault localization tends to be extremely
hard because random tests, even after minimization, exercise the system in a way that often is not congruent with the intuition of how the system works. Moreover, randomly picking input values alone does not relieve us from the task of defining oracles.

Finally note that this also does not mean that requirements-based testing—which often is partition-based testing either implicitly or explicitly, using the category-partition method [14]—should not be performed. Indeed, a certain amount of requirements-based testing always seems advisable, for various reasons. The main reason likely is that requirements-based tests make it comparably easy to specify the expected values, which is in contrast to random testing, as just discussed. A second reason likely is that explicitly stated requirements often correspond to those parts of the input domain that are most often applied to a behavior description. Because these inputs are used more often, they can be seen to incur a higher risk (which is the idea behind statistical testing [21,22]). That said, however, in terms of failure detection effectiveness, requirements-based testing indeed can in general not be expected to be better than random testing.

3. Instantiations

The characterization of fault models via functions \( \hat{\alpha}_K \) does not require that the fault models can be operationalized directly. In fact, there are multiple variants for the operationalization, depending on parameters such as

- Availability of a specification;
- Availability of a model that is used for the generation of code or tests;
- Availability of an implementation for which correctness is not known;
- Availability of an explicit fault or failure model;
- Availability of an oracle.

In the following, we provide five different ways of instantiating our defect model, thereby demonstrating its generality.

3.1. Security Testing with Vulnerability Injection

In this scenario, we want to test an implementation \( I \) for security vulnerabilities. To this end, a model \( M \) is built that reflects the functionality of \( I \). For a given set of security properties \( S \), such as confidentiality, availability, integrity, absence of XSS or SQLI attacks, we work on the model until \( M \models S \) is satisfied. In some cases, this can be decided using a model checker; in most cases this is not directly possible for deciding \( I \models S \). Now we apply mutations \( \hat{\alpha}_K \) to the model. These mutations reflect vulnerabilities at the code level, for instance, the absence of specific validation procedures. Using a model checker, we can derive counterexamples \( t_i \) if \( \hat{\alpha}_K (M) \not\models S \), and use these traces after suitable concretization \( \gamma \) as test case for the implementation. Running the test decides \( \gamma(t_i) \not\models I \) and hence potentially \( I \not\models S \). If this is the case, a vulnerability has been found in the implementation. And even if no vulnerability is found, the test case still is good because it has tested for a potential vulnerability.

In sum, in this scenario, we have

- a model \( M \) (and thus an explicitly provided oracle),
an implementation $I$ that is not generated from $M$,
• a specification of interesting security properties $S$ with $M \models S$,
• a fault model $\tilde{\alpha}_K$ that comes in the form of mutation operators at the model level, and
• a model checker as test case generator, yielding $\phi_K$.

A respective tool is described in the literature [23,24,25].

3.2. Testing for common faults in Simulink diagrams

In this scenario, we want to test if (the code generated from) a Matlab Simulink model $M$ suffers from typical faults such as division by small values and resulting overflows, absolute values for minimum or maximum numbers, threshold violations, or numerical problems that are the result of performing a multiplication before a division rather than vice versa. There usually is no specification. We hence assume an implicit specification $S$ that requires that no minimum value is passed to a block that computes an absolute value; that no denominator of a division block is very small; that specific thresholds are not surpassed; and so on. The implementation $I = \gamma(M)$ is generated using some code generator $\gamma$.

We then use our defect model as follows. $\tilde{\alpha}_K$ removes checks for valid ranges of input and output signals of specific blocks (abs, div, etc.). $M$ is understood as the result of having removed these checks in a hypothetical model $M'$ with

$$M' \models S \text{ and } \tilde{\alpha}_K(M') = M.$$ 

$M'$ can be seen as the “repaired” version of $M$ in that it contains the missing checks. To generate test cases, the idea is to look for smells in $M$. These smells represent blocks, or combinations of blocks, that do not exhibit respective range checks and that, as a consequence, may lead to the above typical faults. Once these blocks have been identified in $M$, we trace them back to respective code parts in $I$, and then use symbolic execution of $I$ to find input values to these (combinations of) blocks that would lead to range violations.\(^2\) If respective traces can be found, i.e., if $\tilde{\phi}(I,S) \neq \emptyset$, $I$ and consequently $M$ suffer from the fault encoded by $\tilde{\alpha}_K$: there is no check of the validity of ranges.

In sum, in this scenario, we have

• a model $M$,
• an implementation $I = \gamma(M)$ that is generated from the model,
• an implicit specification $S$ (and thus an implicit oracle) that is the result of negating the effects of faults of class $K$,
• a fault model $\tilde{\alpha}_K$ that describes the lack of guards for the input and output ports of specific blocks,
• a symbolic execution engine as test generator to compute $\phi_K$.

A respective tool, 8Cage, is described in the literature [26].

\(^2\)Strictly speaking, this leads to the notion of error models here because the internal inputs to specific blocks correspond to the system’s internal state and cannot be observed from the outside.
3.3. Testing continuous closed-loop controllers

Given an intended value provided by the user of a system, continuous controllers attempt to make sure that a variable of the controlled system (the plant) reaches that intended value (there can be multiple variables to be controlled). In this scenario, we want to test if the trajectories defined by a continuous controller $M$ specified in Matlab Simulink satisfy standard properties such as smoothness, responsiveness, stability, precision, and so on. These properties come as functions $s_\ell$ that map trajectories $T$ of $M$ to quality values $Q$.

The specification $S$ states that for each trajectory $t \in T$, specific minimum values are not reached, $s_\ell(t) > \text{min}_\ell$. While $D \models S$ for the differential equations $D$ that correspond to $M$ can in principle be decided using standard control theory, we are interested in computing test cases for the generated code $I = \gamma(M)$ that show where and how badly these properties are violated, for subsequent assessment of the engineer.

In this case, there is no fault model. Instead, we make use of a failure model which encodes the violation of $S$ and hence is derived from $S$:

$$\varphi(M, S) = \{ t \in T : s_\ell(t) \leq \text{min}_\ell \}.$$  

For scalability’s sake, and for the sake of restricting the system’s search space for analysis, we add information about situations in which these failures may occur, e.g., by stipulating that there are only two intended values subsequently to be set (see below).

For simplicity’s sake, we assume that the controller starts with an intended value of 0, then tries to make the system reach an intended value $i_1$, and after some more time—the amount of which is to be determined—another intended value $i_2$ (hence $i_1$ and $i_2$ form a step as input to the controller). In other words, we restrict the set of all potentially relevant traces to those where we set the intended value twice, in a predefined temporal distance. Using this simplification, we compute the approximation $\tilde{\varphi}$ as follows.

A realistic range $\{\text{min}, \ldots, \text{max}\}$ for the control variable (and thus for the intended values $i_1$ and $i_2$) yields the set of possible combinations of $i_1$ and $i_2$. We want to know for which combination of $i_1$ and $i_2$ the properties in $S$ are dissatisfied (or even better, compute the worst case). Trying all combinations is impossible. Instead, we partition the range of $i_1$ and $i_2$ into $n$ blocks for some $n$ (say, $n = 10$). This yields $n^2$ combinations of subranges. Graphically speaking, the ranges for $i_1$ and $i_2$ form a square which is partitioned into a grid with $n^2$ cells. From each cell, we randomly sample $m$ (say, $m = 5$) values for $i_1$ and $i_2$, and compute the quality characteristics in $S$ for each of the $m$ points in the cell. In sum, we compute $m \cdot n^2$ trajectories and their quality characteristics. Cells with low minimum/average/maximum qualities for any of the quality criteria specified in $S$ are subject to further scrutiny by using heuristic search algorithms to approximate local minima. If these violate $S$, we have derived test cases $T'$ the input part of which consists of two intended values for the controlled variable, $i_1$ and $i_2$. This procedure defines

$$\tilde{\varphi}(M, S) = T' \subseteq \varphi(M, S).$$

In sum, in this scenario, we have

- a model $M$,
- an implementation $I = \gamma(M)$ that is generated from the model,
• an explicit specification $S$ (and thus a derived oracle) that stipulates lower bounds for various quality characteristics,
• a failure model $\varphi$ that describes trajectories that do not meet the quality characteristics in $S$, and
• systematic random sampling and heuristic search as test generator to compute $\tilde{\varphi}$.

Respective tools are described in the literature [27,28].

3.4. Testing legacy information systems

One practical recurring problem with legacy information systems written in COBOL or RPG is the way they handle internal state. The GUIs of these systems often consist of a sequence of text-based screens where values can be entered, and aggregate values are computed. The screens are supposed to be entered in a well-defined ordering: first screen first, second screen second, etc. This assumption leads to efficient incremental updates of the internal data state. However, if screens are visited in a different order, then the assumptions behind the definition of the incremental updates do not hold anymore, and the data state gets corrupted. In order to find out if a given legacy system $I$ suffers from this problem, we build a model $M_1$ in the form of an extended finite state machine. Each control state $s_\ell$ corresponds to one screen, and the EFSM's data state is some abstraction of the data state of $I$, say, a meaningful aggregate value $a_\ell$ for which the corruption of the data state is relevant, for instance, the currently computed value of an insurance premium. The transitions in $M_1$ correspond to actually implemented navigations in-between the screens.

Similar to the example of continuous controller testing, in this case we may need no fault, but only a failure model. The failure model describes all those sequences of transitions $t_1, \ldots, t_m$ of $M_1$ that lead to values of $a_\ell$ that are not the intended values, $a'_\ell$. The specification $S$ states that in each state $s_\ell$, $a_\ell$ should equal $a'_\ell$. Since $M_1$ is an approximation, this yields

$$\tilde{\varphi}(M_1, S) = \{ (t_1, \ldots, t_m) : a_m \neq a'_m \}$$

which can be computed using classical algorithms for state machines.

In this case, we obviously generate tests using a failure model. It is also possible to do so using a fault model. Assume a model $M_2$ with states $s_1, \ldots, s_n$ to reflect the text screens, and the only possible transitions are from $s_\ell$ to $s_{\ell+1}$ for $\ell < n$. This model reflects the programmer’s understanding of what the users should do. A fault model $\tilde{\alpha}_K$ now introduces arbitrary transitions in-between pairs of states in $M_2$, and we can again use classical algorithms to generate tests that exercise all transitions of the model, leading to states that may or may not violate the data invariants.

In sum, in this scenario, we have

• models that we derive manually (i.e., abstract roughly) from our understanding of the implementation and the application domain. $M_1$ reflects the actual transitions of the system, some of which may lead to erroneous states. $M_2$ contains only the transitions that correspond to the programmer’s—thus not necessarily the user’s—understanding of how the system should be used,
• an implementation $I$ that is the COBOL or RPG legacy code,
• a hand-crafted specification $S$ that stipulates invariants on the model’s data state, and hence a derived oracle, and
• in case of model $M_1$: a failure model $\phi$ that describes traces that end in each state of $M_1$, together with respective classical test case generation technology, or
• in case of model $M_2$: a fault model $\alpha_K$ that injects hypothetically incorrect transitions into $M_2$ and then a test case generator $\phi_K$ that covers all transitions (or pairs of transitions or using any other criterion).

3.5. Code coverage via symbolic execution and model checking

The last instantiation of our framework is given by tools such as JavaPathFinder [29,30] or KLEE [31] or SAGE [32]. The idea is to generate a test suite with high coverage for the program under analysis. Coverage itself is not a fault model—and in fact generally does not correlate with failure detection, see Section 2.6. Instead, many methodological uses of these tools use a defect model that is either given implicitly (“the program crashes”) or can be encoded as an explicit assertion for the violation of which the respective tool is supposed to find a trace. This instantiation of our defect model is similar to the example in Section 3.2 (and the 8Cage tool makes use of KLEE) but (1) is applied to code rather than to a model and (2) uses a failure rather than a fault model. The specification $S$ is a set of properties such as “does not crash” or the negation of one of the assertions described in the failure model$^3$, the implementation $I$ is the provided code, the failure model $\phi$ describes specific assertions or the fact that the program should not crash, and $\phi$ is the symbolic execution tool that is run on the code either with the explicit assertions or with the goal of maximum coverage and possible crash of the program.

In sum, in this scenario, we have
• no model,
• an implementation $I$ that is x86 binary or Java or C code,
• a specification $S$ that stipulates that the failures captured by the failure model should not materialize (and hence an implicit oracle),
• a failure model $\phi$ that describes traces that lead to the violation of explicitly stated assertions or that make the program crash, and
• systematic concolic execution as test generator to compute $\phi$.

3.6. Summary

The different methodological usages of our defect model can be summarized as follows (see Table 1):
• model explicitly provided, specification explicitly provided, model correct, fault model injects vulnerabilities $K$ at model level ($\alpha_K$), model checker generates tests ($\phi_K$):
• model explicitly provided, specification derived from fault model $\alpha_K$ that disallows a specific internal state, implementation is generated simulation or production code, only $\text{cod}(\alpha_K)$ is used for “smell detection”, $\phi_K$ induced by fault model;

$^3$Again, we could introduce the notion of an error model here because the assertions may well access the internal state of the program under analysis.
• model explicitly provided, specification available, implementation is generated simulation or production code, no fault model, failure model negates specification;
• model that encodes intended behavior derived from code and additional knowledge, specification simple invariants, implementation provided, fault model $\alpha_K$ injects unintended transitions in the derived model, failure model $\phi_K$ induced by $\alpha_K$;
• model that encodes actual behavior derived from code, specification simple invariants, implementation provided, no fault model, failure model $\phi_K$ covers all transitions;
• no model, specification consists of assertions in the code (or “should not crash”), no fault model, failure model negates specification.

It is interesting to see that defect-based testing does not necessarily require an explicit oracle: the oracle often is derived from a simple general specification or a simple model. In the above examples, the derivation of an explicit specification (and thus oracle) was needed only in the case of testing legacy systems. The reason is that this scenario is specific to a class of applications, whereas all the others are specific to a domain for which a general defect model has been designed.

4. Methodology

So far, we have considered defect-based testing from a theoretical and a technological perspective. In Section 3, we have seen that our theoretical model of defect-based testing can be instantiated in various ways. For instance, fault injection can be shown to instantiate the model; so can smell detection with subsequent test case generation to “exercise” the smells; and structured simulation of continuous systems is another possible instance.

Yet, we have not discussed the provenance of our defect models (or more precisely: the defect classes that are turned into defect models) so far. There are several standard defect classes that seem to be applicable almost universally: division-by-zero, null pointer dereferencing, and other defects that are targeted by limit testing strategies; absence of deadlock, livelock and various forms of race conditions; invalid state changes in a system; and so on. These classes are universal in that they do not apply to one specific domain, but rather to almost all domains.

The examples in Section 3 show that some fault classes are domain-specific: responsiveness of a continuous controller obviously is specific to the domain of continuous controllers; and the problem of text screens being used in an order different from what the programmer has intended is, as a first approximation, specific to the class of systems that do work with text-based GUIs with different screens. We will argue that domain-specific fault classes can be elicited and managed systematically. This, however, necessitates a brief discussion of the nature of a “domain”.

Domains relate at least to problem classes, organizations and their system development culture, technology, and applications targeted at a similar functionality (let us refrain from clearly defining what “similar” means here—what we mean is that a dishwasher controller and a tumbler controller target similar functionality, whereas quick-sort and matching software for advertisements do not). Defect-based testing seems particularly promising in situations where defect models can be re-used, that is, where
<table>
<thead>
<tr>
<th>Test Name</th>
<th>Specification S</th>
<th>Model M</th>
<th>Implementation I</th>
<th>Environment Model</th>
<th>Fault Model $\alpha_K$</th>
<th>Failure Model $\phi_K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecTest (§3.1)</td>
<td>confidentiality, integrity, non-repudiation properties</td>
<td>explicit Aslan++ model that satisfies the specification</td>
<td>any binary web application</td>
<td>other communicating web apps</td>
<td>mutations $\alpha_K$ inject vulnerabilities into the model</td>
<td>induced by $\alpha_K$</td>
</tr>
<tr>
<td>8Cage (§3.2)</td>
<td>derived from failure model</td>
<td>Simulink</td>
<td>generated production or simulation code</td>
<td>possible, but not necessary</td>
<td>code($\alpha_K$) describes smells that indicate missing checks</td>
<td>induced by $\alpha_K$</td>
</tr>
<tr>
<td>ContTest (§3.3)</td>
<td>criteria for controller quality including precision, responsiveness, stability</td>
<td>controller modeled in Simulink</td>
<td>generated production or simulation code</td>
<td>Simulink plant model for MiL test</td>
<td>none</td>
<td>negation of specification</td>
</tr>
<tr>
<td>legacy (M₁ in §3.4)</td>
<td>specific values correct</td>
<td>EFSM derived from implementation: actual behavior</td>
<td>COBOL/RPG code</td>
<td>none</td>
<td>none</td>
<td>$\phi_K$ tries covers all transitions</td>
</tr>
<tr>
<td>legacy (M₂ in §3.4)</td>
<td>specific values correct</td>
<td>EFSM derived from implementation: behavior intended by programmer</td>
<td>COBOL/RPG code</td>
<td>none</td>
<td>$\alpha_K$ injects transitions into $M_I$ that are not intended by the programmer</td>
<td>induced by $\alpha_K$</td>
</tr>
<tr>
<td>SymbEx (§3.5)</td>
<td>explicit assertions, &quot;no crash&quot;</td>
<td>none</td>
<td>Java, C, x86</td>
<td>none</td>
<td>none</td>
<td>symb. execution to compute traces that violate spec.</td>
</tr>
</tbody>
</table>
there are recurring, or typical, defect classes; see also our discussion on effectiveness in Section 2.4. The causes for defects can be manifold: because a specific subproblem is extremely hard; because new technology—such as IDEs, programming languages, or sensors—is used; because there are no or bad requirements descriptions; because quality assurance activities are badly managed; because subsystems are used in a context for which they have not been designed and this is not carefully checked; because requirements or programmers are too junior or outright incapable or given problematic incentives; and so on. This variety of reasons has been acknowledged for a long time [33].

Arguably, there are contexts, i.e., domains, in which similar characteristics of these kinds can be identified. In this case, it seems sensible to capture the respective defect class, turn it into a defect model, and operationalize it, be that by testing or by code reading techniques (Section 5) or any other means.

The idea to capture and operationalize defect classes is not new. There exists a multitude of defect classifications, even standardized, and also techniques to classify defects [33]. In some cases, standard defect classifications can be used. In other cases, domain-specific defects need to be elicited. This can be done on the grounds of interviews with stakeholders including developers, project managers, testers, and product managers; on the grounds of careful analysis of bug trackers, versioning system logs; and so on. In our experience, classes of recurring defects exist in every organizational context.

Let us now assume that in a given context, defect classes have been elicited and turned operational by whatever means of test case generation or code reading. We are not saying that this is a simple undertaking. Quite to the contrary—if employees need to fear the consequences of defects being closely investigated, there are significant odds that they may not be willing to cooperate. This of course depends on an organization’s culture and on how information on defects is used and communicated. In any case, assuming that a given organization’s culture embraces a culture of defects, it is likely that after some time, the “typical” defects will change. This is because the organization learns about historical defects and in some cases will not introduce them again into their products, or at least less frequently.

Then, defect models need to be updated: there must be a lifecycle for defect model management [34]. In other words, the above activities to initially elicit defect classes must be performed on a recurring basis: some defects will be weeded out, some new defects will appear, some defects will be recognized to not have been so relevant in the first place, for some defect classes, their operationalization will turn out to be ineffective or inefficient, and so on. How exactly this life cycle is managed is beyond the scope of this article, but we do not want to suggest that defect models only need to be built once and for all.

In addition, the above considerations are not meant to imply that all defects in a system are typical and recurring. In some cases, specific failure situations may be considered so problematic that a defect model and its operationalization are built only for this specific situation: the respective defect models are then not referring to typical defects in the sense of Section 2.4 but rather to grave defects in the sense of Section 2.5.

In terms of the operationalization of defect models, this article has taken the perspective of testing only. However, several reading techniques are based on checklists, and it seems a very natural way to turn fault classes or fault models into checklists. Designing good check lists is an art, however, and there certainly is a need for assessing there
effectiveness and efficiency, both in absolute terms and in comparison with defect-based testing.

5. Conclusions

The starting point is our question of what makes a test case a good test case: its ability to provoke likely failures with good cost-effectiveness. To capture this notion, we have presented a theoretical framework to capture defects, that of defect models. Defect models formally capture the essence of defect classes such as “division-by-zero” or “smoothness of the controller in a specific range cannot be guaranteed”.

Defects can be faults, in which case fault models are formally defined as mappings from correct to incorrect behavior descriptions. Defects can also be failures, in which case failure models are defined as characterizations of those inputs (and expected outputs) of a system that do not conform with a specification. Fault models naturally induce failure models, but there are examples of failure models for which no useful fault models are known.

At a theoretical level, we have shown how defect models can be qualified in terms of cost-effectiveness, by relating them to risk and cost. We acknowledge that in practice, the respective assessments of risk (what can go wrong, and how critical is something that went wrong) and cost (what is the cost of testing, fault localization, and debugging; what is the cost of remaining defects in the field) are far from being trivial.

Defect models are supposed to be built on the grounds of observations in a specific domain, to the end of capturing relevant and typical and highly likely or highly expensive defects. In these situations, defect models can be re-used. Sometimes, defect models will have to be built for one specific hypothetical fault, without re-use or potential of re-use. In which situations this is an economically viable option is the subject of future work.

We would like to re-iterate that fault-based testing is not meant to be the only form of testing but should be complemented by other techniques, including requirements-based testing, regression testing, statistical testing, random testing, and so on.

We have presented several examples of how our theoretical framework can be interpreted to capture existing test case generators. We believe that this provides some preliminary evidence that the model is sufficiently general. We deem it interesting that the instantiations differ widely from each other: sometimes the domain of an $\alpha$ is used for the sake of fault injection, sometimes its codomain for the sake of detecting smells, sometimes an induced failure model for test case generation based on knowledge of faults, sometimes explicit (that is: not induced by a fault model) failure model to capture potentially defective situations.

Finally, we have argued that living fault models must be managed, and that this certainly is not a technological problem only. We have hinted that knowledge of defect classes and defect models can not only be turned operational in terms of test case generators, but also in terms of check lists to be used in reading techniques.

The ideas in this article are rooted in a plethora of related ideas, some of which have been referenced throughout the paper. Specifically, we have seen that many defect models are known in the literature and/or used in practice. We refrain from a full literature survey, which would make for an own article, but rather relate to some relevant fields, including the following:
• requirements engineering (which leads to defects by negation, or sometimes by explicitly stating anti-requirements which directly give rise to defects);
• smell detection (which we use for detecting potentially failure-causing pieces of code);
• coding guidelines (which try to avoid smells);
• defect classifications (which are a basis for defect classes and then defect models);
• fault injection (which is one instance of the possible operationalizations of our model);
• defect modeling (both in general—to understand the relationship between errors, faults, and failures and hence maybe the notion testability and the cost of fault localization—and specifically, in which case it can directly lead to fault models);
• fault trees (which help structuring defects and building subclasses);
• FMEA (which help identify faults and failures) and FMECA (which help identify critical faults and failures);
• hazard analysis (which can be used as or turned into a failure model);
• mutation testing (which performs fault injection, albeit of a restricted kind without necessarily correlating with real-world defects);
• metamorphic testing (which is one testing technique that does not need to provide explicit oracles, similar to some instantiations of our defect model);
• risk-based testing (which aims at prioritizing tests for the sake of cost-effectiveness);
• model-based testing (which relies on models of a system under test, its environment, and/or potential faults);
• random testing (which is not defect-based yet effective at finding faults but arguably not necessarily cost-effective) and adaptive random testing;
• partition-based testing (which is the strategy that most test selection strategies boil down to, including failure-based testing);
• process improvement (which can be based on observations of currently relevant defect classes);
• the software experience factory and process improvement models (which embody the idea of measuring and feeding back observations into the development process);
• symbolic execution, model checking, and SAT solving as well as various forms of search (which are commonly used techniques for test case generation);
• analytical techniques to prove properties of a system or a model of this system (which can be seen as an alternative to testing based on defect models—but properties to be proved often turn out to be negations of fault hypotheses, e.g., “no deadlock”);
• knowledge management (which is concerned with the general problem of cost-effectively and usably capturing and disseminating knowledge in an organization);
• check lists (which can be a different way of capturing fault models, for the sake of code reading techniques); and
• domain-specific engineering (which, like our defect classes, aims at exploiting idiosyncrasies of a specific domain).

As future work, we plan to investigate the effectiveness and efficiency of check lists for code reading techniques when they are derived from defect classes or defect models. Additionally, in open contexts such as the internet with services being composed at run-
time, it seems interesting to use the negation of defect models as (partial) specifications, to be checked statically or dynamically, for components that are integrated into a system at runtime.

6. Acknowledgments

The ideas in this article have greatly benefited from discussions with the lecturers and student participants of the 2014 Marktoberdorf summer school. Numerous discussions with Dominik Holling and Matthias Büchler have led to the understanding of defect models described in this article.

References


