Introducing Complexity to Testing

Ismael Rodríguez, Fernando Rosa-Velardo, and Fernando Rubio

Abstract—We develop a general theory to introduce asymptotic complexity to testing. Our goal is measuring how fast the effort of testing (measured in terms of the number of tests to be applied and their length) must increase to reach bigger levels of partial certainty on the correctness of the implementation under test (IUT). By recent works it is known that, for many practical testing scenarios, *any* partial level of correctness certainty less than 1 (where 1 means full certainty) can be reached by some finite test suite. However, how fast must these test suites grow as long as the target level gets closer to 1? More precisely, how do test suites grow with α, where α is the inverse of the distance to 1 (e.g. if $\alpha = 4$ then our target level is $0.75 = 1 - \frac{1}{4}$)? In this paper we develop a theory to measure this testing complexity. We use the theory to analyze the testing complexity of some general testing problems, as well as the complexity of some specific testing strategies for these problems, and discover they are within e.g. $\mathcal{O}(log\ \alpha)$, $\mathcal{O}(\alpha)$, $\mathcal{O}(\alpha\ log\ \alpha)$, or $\mathcal{O}(\sqrt{\alpha})$. Similarly as the computational complexity theory conceptually distinguishes between the complexity of *problems* and *algorithms*, tightly identifying the complexity of a testing *problem* will require reasoning about any testing *strategy* for the problem. The capability to identify testing complexities will provide testers with a measure of the *productivity* of testing: after applying n test cases, what is the utility of applying one more test? How closer would that additional test case get us to the (ideal) complete knowledge on its (in-)correctness? When should we stop testing, because applying one more test is not worth the effort?

Index Terms—Testing, complexity theory.

1 Introduction

COMPUTER Science has made strong efforts to identify its own limits. On the one hand, Computability provides techniques to know whether a given problem can be computed or not (e.g. diagonal arguments, many-one reductions, Rice theorem, etc). On the other hand, Complexity gives methods to identify whether a problem can be solved in reasonable (typically, polynomial) time. For some problems we know this can be done (we prove it by giving an efficient algorithm), for others we can just say that we *probably* cannot do it (e.g. NP-complete problems), and for others we can categorically affirm that we cannot do it (e.g EXPTIME-complete problems).

Although the application of formal methods to testing is a mature field [1], [2], [3], [4], [5] and formal testing frameworks have been proposed for many different kinds of underlying models [3], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], little efforts have been made to identify the *limits* of testing in the same way as the limits of Computation. Hereafter, let us say that a test suite is *complete* if, after applying it to the implementation under test (IUT), the IUT replies will let us know with certainty whether the IUT is correct or not with respect to a given specification. Why do *finite* complete test suites exist in some testing scenarios (typically, ideal ones), whereas they do not in others (typically, realistic ones)? Several factors are known to be important in this regard. For instance, the set of hypotheses assumed by the tester is known to play a key role in testing [4], [17], [18], and in some particular cases they even enable the existence of finite complete test suites [19], [20]. More generally, in [21] (later continued in [22]) the difficulty of testing was studied in terms of the problem of identifying the true definition of a given black-box IUT (out of a given set of possible definitions) just by interacting with it. However, note that the actual condition required to test up to *completeness* (that is, until we do know whether the IUT is correct or not) is weaker than that indeed: We can stop testing when we know with certainty whether the IUT belongs to the set of possible correct definitions or not —even if we still cannot identify which one of these definitions the IUT actually is.

To the best of our knowledge, the first general theory identifying the difficulty of testing, measured in terms of the sizes required by test suites to be complete, was given in [23] (this work completed an earlier version of the theory presented in [24]). Testing scenarios were classified according to the following testability hierarchy: there exists a finite complete test suite (`Class I`); for any partial level of knowledge on the IUT correctness, that level can be reached with a finite test suite (`Class II`); there exists a countable complete test suite (`Class III`); there exists a complete test suite (`Class IV`); and all cases (`Class V`). It was shown that each class is strictly included in the next, and that being included in some of the best testability classes does not imply that the assumed computation model is simple: Testing systems of some form written in a Turing-complete language, and testing some kind of machines depending on *continuous* magnitudes, were shown to be in `Class I`, whereas testing systems with some kind of rational-valued temporal time-outs, and testing systems defined under the *must* testing semantics of M. Hennessy's model [25], were shown to belong to `Class II`. Other issues, such as alternative characterizations, the complexity of computing the quality of incomplete test suites in terms of the *least hypothesis* that would make them complete, and reductions

This work has been partially supported by projects TIN2012-39391-C04-04, TIN2015-67522-C3-3-R, and S2013/ICE-2731.

- *Dpto. Sistemas Informáticos y Computación. Facultad de Informática. Universidad Complutense de Madrid. 28040 Madrid, Spain*
  E-mail: `{isrodrig,fernandorosa,fernando}@sip.ucm.es`

between testing problems, were also addressed in [23].[1]

`Class II` is probably the most interesting class in that hierarchy. It defines a kind of testability *in the limit*: as long as the size of the test suite to be applied increases, our knowledge on the IUT (in-)correctness converges to 1 (i.e. full).[2] More formally, given the set of definitions the black-box IUT may have (where some of them are considered as correct with respect to some specification, and some of them are considered incorrect), the measure we are requiring to converge to 1 is in fact a *distinguishing* rate: the proportion of all possible pairs *"(possible correct definition of IUT, possible incorrect definition of the IUT)"* which will necessarily be *distinguished* by the IUT outputs observed after applying the test suite under consideration (we say that a pair is distinguised if outputs produced by the IUT let us discard at least one element in the pair). Note that, if the distinguishing rate of a test suite is 1, then all pairs are distinguished, so this suite lets us properly classify the IUT as within the set of possible correct definitions or (exclusive-or) as within the set of incorrect definitions —even if we cannot identify it within its corresponding set. Hence, the test suite is *complete*. Note that the distinguishing rate (DR) notion is undefined for infinite sets, as this rate is a proportion. In order to let `Class II` deal with infinite sets of possible definitions of the IUT, *sequences* of *finite* sets converging to the original sets are considered, instead of the infinite sets themselves. In particular, a testing scenario belongs to `Class II` if, for some sequence like this and for all distinguishing rates being less than 1, there exists some finite test suite reaching that distinguishing rate for *all* sets in the sequence. For instance, a testing scenario would be in `Class II` if some test suite of size 15 reaches distinguishing rate 0.9 in *all* sets in the sequence, some test suite of size 450 reaches DR 0.99 in *all* sets in the sequence, and so on for all DR less than 1.

In this paper we construct a general testing complexity theory based on `Class II`. Let us consider a strategy that, given the target DR $d < 1$, produces a test suite reaching a DR equal to or higher than d for all sets in the considered sequence. The *testing complexity* of this strategy is the function denoting how fast the sizes of the test suites produced by the strategy grow with d. Borrowing a typical notation used e.g. in the definition of Computational Complexity's approximation class FPTAS, rather than measuring the sizes of test suites in terms of d, we will do it with respect to $\alpha = \frac{1}{1-d}$ (e.g. if $\alpha = 4$ then our target level d is $0.75 = 1-\frac{1}{4}$). Besides, we say that the testing complexity of a *testing problem* is within some asymptotic complexity (e.g. $\mathcal{O}(log\ \alpha)$) if it supports some strategy matching that complexity. For a given testing problem, we will be interested in finding strategies with a small asymptotic complexity, i.e. strategies where the sizes of the produced test suites grow slowly with α, as they provide more (partial) information on the IUT correctness with lesser testing effort. In order to illustrate the capability of the proposed testing complexity theory to discover new interesting information about the difficulty of testing in several frameworks, we identify the testing complexity of several testing strategies and problems, which turn out to be in e.g. $\mathcal{O}(log\ \alpha)$, $\mathcal{O}(\alpha)$, $\mathcal{O}(\sqrt{\alpha})$, or $\mathcal{O}(\alpha\ log\ \alpha)$.

The contributions of this paper are the following:

(a) converting the qualitative notion defined by `Class II` in [23], which essentially means *"complete testing can be reached in the limit"*, into a quantitative notion allowing us to measure the *speed* of that convergence towards completeness in each particular case;

(b) based on the previous notion, measuring the *productivity* of testing in terms of the utility of applying one more test after we have already applied n tests. Testers have the intuition that, for any high $n \in \mathbb{N}$, the n-th test case they apply has less fault detection capability than their first test cases —provided that the IUT passed all previous test cases. By *measuring* how this fault detection capability falls, we can decide when we should stop testing because the utility of applying one more test is negligible; and

(c) for some testing problems, identifying not only the testing complexity of applying some specific testing strategies (e.g. by applying some test suites, the problem can be tested in $\mathcal{O}(\alpha\ log\ \alpha)$ testing complexity), but also the testing complexity of the testing problem itself (i.e. proving that the testing complexity of *any* testing strategy for that testing problem cannot be better than $\mathcal{O}(\alpha\ \ log\ \ \alpha)$, so a $\mathcal{O}(\alpha\ \ log\ \ \alpha)$ testing strategy is optimal in asymptotic terms).

The rest of the paper is organized as follows. In the next section we introduce some basic testability notions primarily borrowed from [23]. Next, in Section 3 we introduce the new notions needed to construct a testing complexity theory, and the complexity of some testing strategies for some testing problems is shown in Section 4. The complexity of the considered problems themselves is addressed in Section 5. Finally, in Section 6 we present our conclusions and lines of future work.

2 Testability concepts

In order to make a self-contained presentation, next we briefly introduce some basic notions borrowed from the general testability theory given in [23]. These notions let us define the set of possible definitions of the IUT, as well as the subset of those which are considered as correct by the tester. Each possible definition is denoted just as a function which, for each possible input, returns the set of possible outputs the system could return (the system non-deterministically chooses which one it returns). Besides, the model lets the tester explicitly denote which pairs of outputs can be distinguished by means of observation. In [23], abundant examples illustrate the expressivity of this model despite its simplicity. These examples include simple models representing testing deterministic and non-deterministic *finite state machines* (FSMs), where functions representing

1. Note that the *computational* complexity of some testing tasks was analyzed in [23], in particular the time complexity of some test suite designing tasks (e.g. the NP-completeness of constructing the smallest complete test suite in a finite testing setting). On the contrary, this paper will address the *testing* complexity, i.e. the testing effort itself.

2. This notion should not be confused with the more straightforward notion of *testing in the limit* mentioned in [22], which basically requires that some *countable* test suite lets us uniquely identify the IUT. Letting aside that the testing notion in [22] requires unique identification rather than just distinction between correct and incorrect, that view of testing in the limit would be more related to `Class III` than to `Class II`.

possible IUT definitions actually denote possible FSMs behaviors, and each possible input of these functions actually denotes a *sequence* of machine inputs —and the same for outputs. Other models denote programs written in any Turing-complete language and temporal systems. The latter illustrate how non-functional properties (in this case, the elapsed times) can be trivially codified into the inputs and outputs of the model together with the actual system inputs and outputs (e.g. buttons and lights, respectively). Besides, some examples of distinguishing relations illustrate cases where we cannot distinguish e.g. terminating behaviors from non-terminating ones, as well as close values under the presence of imprecise measuring.

Definition 1. Let O be a *set of outputs*.

- A *distinguishing relation* for O is an anti-reflexive symmetric binary relation $\not\doteq$ over O. We denote the complementary of $\not\doteq$ by $\doteq$.
- A *set of distinguished outputs* is a set O of outputs with a distinguishing relation $\not\doteq$.
- We extend the relation $\not\doteq$ to sets of outputs as follows. Let $\mathcal{O}', \mathcal{O}'' \subseteq O$. We say that $\mathcal{O}'$ and $\mathcal{O}''$ *are distinguishable*, denoted by $\mathcal{O}' \not\doteq \mathcal{O}''$, iff $o' \not\doteq o''$ for all $o' \in \mathcal{O}'$ and $o'' \in \mathcal{O}''$. Again, the relation $\doteq$ is the negation of $\not\doteq$.

Hereafter, when no distinguishing relation is explicitly given, we will assume the trivial inequality: $o_1 \not\doteq o_2$ iff $o_1 \neq o_2$. □

Next we introduce the notion of computation formalism, which denotes the set of all possible behavioral definitions the IUT could have (each one denoted by a function). We assume that 2^A denotes the powerset of set A.

Definition 2. Let I be a set of inputs and O be a set of distinguished outputs. A *computation formalism* C for I and O is a set of functions $f : I \to 2^O$ where for all $i \in I$ we have $f(i) \neq \varnothing$. □

Given a function $f \in C$, $f(i)$ represents the set of outputs we can obtain after applying input $i \in I$ to the computation artifact represented by f. Since $f(i)$ is a set, f may represent a non-deterministic behavior. Besides, C, I and O can be infinite sets. Representing the behavior of systems by means of functions avoids to implicitly impose a specific structure to system models (e.g. states, transitions). Still, elaborated behaviors can be represented (see the examples commented above in [23]).

Computation formalisms are used to represent the set of possible implementations we are considering in a given testing scenario. Implicitly, a computation formalism C represents a *fault model* (i.e. the definition of what can be wrong in the IUT) as well as the *hypotheses* about the IUT the tester is assuming. For instance, if the IUT is assumed to be a deterministic FSM and to differ from a given correct FSM in at most one transition, then only functions denoting the behaviors of such FSMs (including the correct one) are in C. Alternatively, if we only assume that the IUT is represented by a deterministic FSM, then C will represent all deterministic FSMs.

Computation formalisms are also used to represent the subset of specification-compliant implementations. Let C represent the set of possible implementations and $E \subseteq C$ represent the set of implementations fulfilling the specification. The goal of testing is interacting with the IUT so that, according to the collected responses, we can decide whether the IUT actually belongs to E or not. Typically, we apply some tests (i.e., some inputs $i_1, i_2, \ldots \in I$) to the IUT so that observed results $o_1 \in f(i_1)$, $o_2 \in f(i_2)$, … allow us to provide a verdict.

Definition 3. Let C be a computation formalism. A *specification* of C is a set $E \subseteq C$. Given a distinguishing relation $\not\doteq$, a *testing scenario* is a triple $(C, E, \not\doteq)$. □

For the sake of notation simplicity, we will write testing scenarios $(C, E, \not\doteq)$ just as pairs (C, E), as $\not\doteq$ will be deducible from the context (when not mentioned, we will always assume the trivial relation $o_1 \not\doteq o_2$ iff $o_1 \neq o_2$).

If $f \in E$ then f denotes a *correct* behavior, whereas $f \in C \backslash E$ denotes that f is incorrect. Thus, a specification implicitly denotes a correctness criterion. For example, let $f, f' \in C$ be two functions such that for all i we have $f(i) = \{a\}$ and $f'(i) = \{b\}$. Then, $E = \{f, f'\}$ denotes that only machines producing *always* a or *always* b are considered correct. We can also construct E in such a way that a given underlying semantic relation is considered (e.g. bisimulation, testing preorder, traces inclusion, conformance testing, etc). For instance, given some $f \in C$, let us consider that f is correct and we wish to be consistent with respect to a given semantic relation $\preceq$, where $A \preceq B$ means that A is correct with respect to B. Then we could define E as follows: $E = \{f' | f' \preceq f \ \wedge \ f' \in C\}$.

Next we identify test suites and complete test suites. By applying tests cases in a *test suite* (i.e. a set of inputs $\mathcal{I} \subseteq I$) to an IUT, collected outputs should (ideally) let us determine whether the IUT is correct or not, i.e. whether the function denoting its behavior belongs to a given specification set $E \subseteq C$. Unfortunately, in black-box testing we do not have access to the IUT definition, so the only thing we can do is to observe the outputs given by the IUT when tests are applied, and collect produced outputs. If observed outputs can be produced by some possible correct behavior (i.e. some $f \in E$) but *cannot* be produced by some possible incorrect behavior (i.e. some $f \in C \backslash E$), then we know for sure that the IUT is correct; otherwise, we cannot assure the IUT correctness. If a test suite $\mathcal{I} \subseteq I$ is such that observed outputs will necessarily let us decide the (in-)correctness of the IUT, then we say that the test suite is *complete*. This requires that, for any pair of correct and incorrect functions (i.e. $f \in E$ and $f' \in C \backslash E$, respectively), there must be a test $i \in \mathcal{I}$ that distinguishes f and f'. Let f'' be the actual behavior of the IUT. When tests $i_1, i_2, \ldots \in \mathcal{I}$ are applied to f'', some outputs $o_1, o_2, \ldots \in O$ are (in general non-deterministically) chosen by f'' and observed ($o_1 \in f''(i_1), o_2 \in f''(i_2), \ldots$). If $\mathcal{I}$ is complete then, for each pair $f \in E$ and $f' \in C \backslash E$ of possible correct and incorrect behaviors of the IUT, some of these outputs, say o_j, must let us distinguish between f and f'. Assuring this property *a priori*, i.e. regardless of non-deterministic choices actually taken by the IUT, requires that, for some i_j, *all* possible non-deterministic output responses of f must be pairwise distinguishable from all possible non-deterministic output responses of f'. In this case, outputs collected by applying

$\mathcal{I}$ to the IUT will let us determine the (in-)correctness of the IUT in any case.

Definition 4. Let C be a computation formalism for I and O, $E \subseteq C$ be a specification, and $\mathcal{I} \subseteq I$ be a set of inputs. We say that $f \in E$ and $f' \in C \backslash E$ are *distinguished* by $\mathcal{I}$, denoted by $\mathtt{di}\,(f, f', \mathcal{I})$, if there exists $i \in \mathcal{I}$ such that $f(i) \not\doteq f'(i)$.
We say that $\mathcal{I}$ is a *complete test suite* for C, E if for all $f \in E$ and $f' \in C \backslash E$ we have $\mathtt{di}\,(f, f', \mathcal{I})$. □

Note that, in the previous definition, $\not\doteq$ is applied to *sets* of outputs, so all outputs from a set are required to be distinguishable from all outputs from the other set.

Observations collected by complete test suites precisely determine whether the IUT is correct or not. Let $\mathcal{I}$ be a complete test suite for the computational formalism C and the specification E. Then, we trivially conclude that:

(a) if $f \in E$, then there is no $f' \in C \backslash E$ such that $f'(i) \doteq f(i)$ for all $i \in \mathcal{I}$, and
(b) reciprocally, if $f \in C \backslash E$ then there does not exist $f' \in E$ such that $f'(i) \doteq f(i)$ for all $i \in \mathcal{I}$.

That is, after applying $\mathcal{I}$, no correct IUT can be considered as incorrect, and no incorrect IUT can be considered as correct.

Once the basic general framework has been presented, `Class I`, `Class III`, `Class IV`, and `Class V` can be easily defined to have those meanings mentioned earlier in the introduction (e.g., a pair (C, E) belongs to `Class I` if there exists a finite complete test suite for C, E; and similarly for classes `Class III`, `Class IV`, and `Class V`). It turns out that each of these classes is strictly included into the next, see [23] for details. As the definition of `Class II` is more relevant to the purposes of this paper (and more tricky to define), in the rest of this section we entirely focus on presenting that class.

Next we elaborate on our idea of finite testability *in the limit* or, more precisely, *unboundedly-approachable* finite testability. When finite testability is not possible, in some cases it may still be possible to test the IUT up to any arbitrarily high *distinguishing rate* with a *finite* test suite. This distinguishing rate notion is the *ratio* between the number of pairs of correct/incorrect functions that are distinguished by the test suite and the total number of correct/incorrect pairs. Note that this ratio can be defined only if the computation formalism C is finite.

Definition 5. Let C be a finite computation formalism for I and O, $E \subseteq C$ be a specification, and $\mathcal{I} \subseteq I$ be a set of inputs. We define the *distinguishing rate* of $\mathcal{I}$ for (C, E), denoted by $\mathtt{d\text{-}rate}\,(\mathcal{I}, C, E)$, as:

$$\frac{|\{(f, f') | f \in E, f' \in C \backslash E, \mathtt{di}\,(f, f', \mathcal{I})\}|}{|E| \cdot |C \backslash E|}$$ □

The assumption that C is finite (otherwise the previous division would be undefined) reduces the applicability of the previous notion, as many interesting computation formalisms are infinite indeed. In order to adapt this idea to infinite computation formalisms, we consider *sequences* of finite computation formalisms. Let us note that, if C is a countable set, then there exists a non-decreasing chain of finite subsets:

$$C^1 \subseteq C^2 \subseteq \cdots \subseteq C^k \subseteq C^{k+1} \subseteq \cdots$$

such that $C = \bigcup_{i \in \mathbb{N}} C^i$. Given a specification $E \subseteq C$, let us consider the sub-specifications $E^i = E \cap C^i$. Since each C^i and E^i are finite sets, we can apply Definition 5 to them indeed. Then, the class of pairs (C, E) which are *unboundedly-approachable by finite testing* (`Class II`) consists of those pairs such that, for some non-decreasing sequence like those described above, any arbitrarily high distinguishing rate less than 1 is reached, by some finite test suite, for *almost all* computation formalisms in the sequence.

Definition 6. We say that (C, E) is *unboundedly-approachable by finite testing* if there is a non-decreasing sequence: $C^1 \subseteq C^2 \subseteq \cdots \subseteq C^k \subseteq C^{k+1} \subseteq \cdots$ with $C = \bigcup_{i \in \mathbb{N}} C^i$ such that, for all $\epsilon < 1$, there exists a finite set of inputs $\mathcal{I} \subseteq I$ and $n \in \mathbb{N}$ such that, for all $l \geq n$, $\mathtt{d\text{-}rate}\,(\mathcal{I}, C^l, E^l) \geq \epsilon$, where $E^l = C^l \cap E$ for all $l \in \mathbb{N}$. We denote by `Class II` the set of all pairs (C, E) that are unboundedly-approachable by finite testing. □

It is straightforward to see that this class is related with the other classes as expected: `Class I` $\subseteq$ `Class II` $\subseteq$ `Class III`. Moreover, both inclusions are proper, and examples within `Class II`$\backslash$`Class I` and within `Class III`$\backslash$`Class II` are given in [23]. Actually, these examples include a very interesting case: a pair $(C, E) \in$ `Class II` and a *smaller* computation formalism $C' \subset C$ such that $(C', E) \notin$ `Class II`. One might think that reducing the set of definitions of the IUT would necessarily make testing easier, but in fact it may make testing a *less productive* activity. In particular, if incorrect functions of any kind are represented in C, but incorrect functions in C' are *only* those that are very difficult to distinguish from the correct ones, then discarding wrong behaviors (i.e. finding faults) might be much harder in (C', E) than in (C, E). This shows that the difficulty of testing does not lie on the sizes of computation formalisms (i.e. how many possible definitions the IUT could have according to our assumptions), but on the *narrowness* of the border between correct and incorrect behaviors (see the examples in [23] for further details).

3 Complexity model

The model presented in the previous section suffices to define the difficulty of testing in qualitative terms (in particular, in terms of whether there exists a finite complete test suite for a pair or not, whether a pair belongs to `Class II`, etc). However, in order to support an (expressive) quantitative measure of testing complexity, more factors should be denoted within the model. Recall that a computation formalism C denotes the set of possible definitions of the IUT according to the tester knowledge, but the relative likelihood of each function f in C (i.e. the likelihood that f is the actual IUT definition, compared to the likelihood of the rest of functions in C) is not quantified. For instance, if the tester assumes that the IUT is unlikely to be non-deterministic (but it is assumed to be possible), then candidate definitions of the IUT including a lot of non-deterministic behavior should be assigned a less relative weight than deterministic or almost deterministic definitions. Let $\mathbb{Q}$ denote the set of rational numbers.

Definition 7. Given a computation formalism C, a *weight function* for C is a function $w : C \to \mathbb{Q}$.

Let us assume the same preliminaries as in Def. 5 (in particular, C must be finite). We define the *distinguishing rate* of $\mathcal{I}$ for (C, E) and w, denoted by $\texttt{d-rate}\,(\mathcal{I}, C, E, w)$, as:

$$\frac{\sum_{f\in E, f'\in C\setminus E, \texttt{di}(f,f',\mathcal{I})} w(f)\cdot w(f')}{\sum_{f\in E, f'\in C\setminus E} w(f)\cdot w(f')} \qquad \square$$

Therefore, pairs of likely functions have a higher impact on the distinguishing rate of a given test suite.

Testers can use different test suites depending on the distinguishing rate they need their test suite to reach. In order to represent all of these choices together, next we introduce the notion of *strategy*.

Definition 8. A *testing strategy* is a function $s : [0, 1] \to 2^I$.
Let $\sigma = [C^1, C^2, \ldots]$ be a sequence of finite computation formalisms such that $C = \bigcup_{i\in\mathbb{N}^+} C^i$ and, for all $i \geq 1$, $C^i \subseteq C^{i+1}$. We say that s is *almost complete* up to $u \in \mathbb{Q}$ through σ if, for all $\alpha \in [0, u)$, there exists some $n \in \mathbb{N}$ such that for all $l \geq n$ we have $\texttt{d-rate}\,(s(\alpha), C^l, E^l, w) \geq \alpha$. $\square$

The *up to* u addition will be omitted when $u = 1$. As mentioned before, the ultimate goal of our testing complexity theory is assessing the *productivity* of testing in terms of how much new information we will get about the possible IUT (in-)correctness by increasing further our *effort* on testing. Hence, a formal measure of that effort is needed. We could consider that the effort of testing just consists in the number of test cases to be applied, i.e. $|\mathcal{I}|$. More sophisticated measures can also be defined, such as considering that each test case has a different cost (for instance, a test case producing more interactions may be considered more expensive). Next we introduce the general notion of test suite cost.

Definition 9. Let $F = \{G \mid G \in 2^I \ \wedge\ G \text{ is finite}\}$. A *test suite cost function* tsc is a function $tsc : F \to \mathbb{N}$. $\square$

For instance, a test suite cost function could be defined just as the addition of the costs of all test cases (inputs) in the test suite, assuming an additional function returning the cost of each individual test case is given. That is, given such a function $\gamma : I \to \mathbb{N}$, we could just define $tsc(\alpha) = \sum_{i\in\alpha} \gamma(i)$. For example, if we are testing FSMs then the cost of each test case could be defined just as the length of the test case (that is, the number of *machine inputs* in the sequence of machine inputs represented by the test case, which is a single element of set I). In this case, the cost of a test suite would be the number of machine inputs produced by all test cases in the test suite.

Next we introduce the cost of a testing strategy. In its basic form, it is a function that, given a target distinguishing rate, returns the cost of the test suite given by the strategy to reach that target rate. An alternative cost function is provided next where, rather than the target distinguishing rate, the input is the inverse of the proportional distance up to the maximum distinguishing rate. For instance, if our strategy is almost complete up to 1, then an input 4 means that our target distinguishing rate is $1 - \frac{1}{4} = 0.75$.[3]

3. This notational approach is used indeed in the definition of other known complexity notions. For instance, in the definition of the (time) complexity class FPTAS, the time complexity is required to be polynomial with the inverse of the distance from the performance ratio to 1 (which is the perfect performance ratio).

Definition 10. Let (C, E), w, tsc, and σ be defined as in previous definitions. We say that the tuple $\mathcal{T} = ((C, E), w, tsc, \sigma)$ is a *testing problem*.
Let s be a testing strategy that is almost complete up to $0 \leq u \leq 1$ through σ. The *basic strategy cost function* of s up to u is a function $\texttt{basic}^u_{s,tsc} : [0, u) \to \mathbb{N}$ defined as $\texttt{basic}^u_{s,tsc}(r) = tsc(s(r))$ for all $0 \leq r < u$.
Besides, the *strategy cost function* of s up to u is a function $\texttt{cost}^u_{s,tsc} : [1, \infty) \to \mathbb{N}$ defined as $\texttt{cost}^u_{s,tsc}(r) = \texttt{basic}^u_{s,tsc}\left(u - \frac{u}{r}\right)$ for all $r \geq 1$.
We say that the testing problem $\mathcal{T}$ can be tested with testing complexity $f : [1, \infty) \to \mathbb{N}$ up to u if there exist a testing strategy s such that $\texttt{cost}^u_{s,tsc} = f$. $\square$

As usual when dealing with complexity measures, the $\mathcal{O}$ notation can be applied to our notions, in particular to both *testing strategies* and *testing problems*, in a similar way as they are applied to measure the time complexity of *algorithms* and *problems*, respectively, in the standard computational complexity theory. For a function $f : [1, \infty) \to \mathbb{N}$, we will use the standard asymptotic notation and write $\mathcal{O}(f)$, $\Omega(f)$ and $\Theta(f)$. If there is a strategy s for a given testing problem $\mathcal{T}$ with $\texttt{cost}^u_{s,tsc} \in \mathcal{O}(f)$, then we say that the testing complexity of $\mathcal{T}$ up to u is within $\mathcal{O}(f)$.

To conclude this section, we address two issues: first, we study how the testing complexity is affected if we change the function tsc, the cost of test suites; second, we will discuss the notion of sequence of finite computation formalism and introduce a notion of natural sequences.

Proposition 1. Let $h : \mathbb{N} \to \mathbb{N}$ be a function and let $\mathcal{T} = ((C, E), w, tsc, \sigma)$ be a testing problem that can be tested up to u with testing complexity f. Then $\mathcal{T}' = ((C, E), w, h \circ tsc, \sigma)$ can be tested up to u with testing complexity $h \circ f$.

Proof: If $\mathcal{T}$ can be tested with f then there is a strategy s that is almost complete up to u in $\mathcal{T}$ such that $\texttt{cost}^u_{s,tsc} = f$. The strategy s is still almost complete up to u in $\mathcal{T}'$ (since this does not depend on the test suite cost). Now, $\texttt{cost}^u_{s,h\,\circ\,tsc}(n) = (h \circ tsc)(s(u - \frac{u}{n})) = h(\texttt{cost}^u_{s,tsc}(n)) = (h \circ f)(n)$, and the thesis follows. $\square$

Let us remark that $f_1 \in \mathcal{O}(f_2)$ does not generally imply $h \circ f_1 \in \mathcal{O}(h \circ f_2)$. However, it is not difficult to see that it is indeed the case if we assume that for every c,

$$h(c \cdot n) \in \mathcal{O}(h) \qquad (1)$$

If h satisfies the previous condition then we can derive a testing complexity in $\mathcal{O}(h \circ f)$ for $\mathcal{T}'$ from a testing complexity in $\mathcal{O}(f)$ for $\mathcal{T}$.

Corollary 1. Let $h : \mathbb{N} \to \mathbb{N}$ be a function satisfying Eq. 1 and let $\mathcal{T} = ((C, E), w, tsc, \sigma)$ be a testing problem that can be tested up to u in $\mathcal{O}(f)$. Then $\mathcal{T}' = ((C, E), w, h \circ tsc, \sigma)$ can be tested up to u in $\mathcal{O}(h \circ f)$.

Given a set C, there are several ways to compose a sequence of finite computation formalisms $\sigma = [C^1, C^2, \ldots]$ such that $C = \bigcup_{i\in\mathbb{N}^+} C^i$ and, for all $i \geq 1$, $C^i \subseteq C^{i+1}$. Note that the chosen way to do so may affect the corresponding complexity. For instance, each C^{i+1} could extend C^i by adding a higher number of easily distinguishable functions than hardly distinguishable functions. If all functions (easy ones and hard ones) were eventually included into some

C^i, then the condition $C = \bigcup_{i\in\mathbb{N}^+} C^i$ would be met. Hence, σ would be a valid sequence for defining a distinguishing rate, and it would make that rate grow particularly slow (so, the corresponding complexity would be particularly low). In order to avoid any bias in the complexity measure caused by these ad-hoc sequences, next we introduce an additional condition for sequences of finite computation formalisms which will be met, in particular, by all examples given in the next section (where systems receiving sequences of machine inputs and replying sequences of machine outputs are tested). Basically it requires that, in each C^i, the number of functions exposing each possible combination of replies for all possible sequences of k machine inputs, with $k \leq i$, is the same. That is, all behaviors involving i or less steps are represented by the same number of functions in C^i. In this case, we will say that the sequence $[C^1, C^2, \ldots]$ is *natural*. Notice that these sets C^i do not bias the number of functions which are easy or hard to distinguish, as all behaviors are necessarily treated in the same way.[4] Next we assume that A^* is the set of all sequences formed by 0 or more symbols from A.

Definition 11. Let $I = \Sigma^*$ and $O = \Gamma^*$ for some finite sets $\Sigma = \{i_1, \ldots, i_n\}$ and $\Gamma = \{o_1, \ldots, o_m\}$. Let C be such that, for all $k \in \mathbb{N}$ and all $f \in C$, if $i \in \Sigma^k$ then $f(i) \in \Gamma^k$. Let $\Sigma^k = \{a^1, \ldots, a^{n^k}\}$, and let $f \xrightarrow{k} (b^1, \ldots, b^{n^k})$ abbreviate $\forall\, 1 \leq h \leq n^k : f(a^h) = b^h$.

We define $\texttt{replies}(C, k) = \{(b^1, \ldots, b^{n^k}) \mid f \in C, f \xrightarrow{k} (b^1, \ldots, b^{n^k})\}$. We say that C is *exhaustive* for I and O if for all $k \in \mathbb{N}$ and all $b = (b^1, \ldots, b^{n^k}) \in \underbrace{\Gamma^k \times \ldots \times \Gamma^k}_{n^k}$ we have $b \in \texttt{replies}(C, k)$.

Let C be exhaustive for I and O, and let $\sigma = [C^1, C^2, \ldots]$ be such that $C = \bigcup_{i\in\mathbb{N}^+} C^i$ and, for all $i \geq 1$, $C^i \subseteq C^{i+1}$. We say that σ is *natural* if, for all C^i, all $k \in \mathbb{N}$ with $k \leq i$, and all tuples $b = (b^1, \ldots, b^{n^k})$ and $b' = (b'^1, \ldots, b'^{n^k})$, we have $|\{f \mid f \in C^i, f \xrightarrow{k} (b^1, \ldots, b^{n^k})\}| = |\{f \mid f \in C^i, f \xrightarrow{k} (b'^1, \ldots, b'^{n^k})\}|$. □

4 Measuring the complexity of testing strategies

In this section we identify the testing complexity of several *testing strategies* for their corresponding problems (the calculation of the testing complexity of the *problems* themselves will be tackled in the Sect. 5). In all cases we assume a common general framework: an *exhaustive* computation formalism C consisting of all functions computed by some class of deterministic reactive machines where each machine input is immediately followed by a machine output. This applies, for instance, to deterministic FSMs or to deterministic Java programs where the interactions from the user and the messages from the program interleave. Hereafter, let the *execution tree* of a system be the infinite tree where each arc is labelled with a machine input and a machine output, and each sequence of consecutive arcs from the root denotes a possible system execution where the machine inputs traversed through its arcs are sequentially given, and the corresponding machine outputs of these arcs are received.

4. It is worth to point out that equally representing *in quantitative terms* all possible behaviors in each C^i does not imply assuming they all are equally probable, and in fact testers can *explicitly* denote that they are not by assigning them different weights. These different probabilities are desirable indeed, as they get the model closer to the tester knowledge (and, as we will see, they will be crucial to determine the testing complexity in each case).

We will consider several possibilities for the weight function, which will result in different complexities. They all assume the *competent programmer hypothesis* [26], [27], that is, it is assumed that the probability of having a mostly correct implementation is higher than the probability of having a very incorrect implementation. Our three examples in this section will present three different interpretations of this assumption. The first one will assume that all faulty machine outputs reduce the feasibility of the corresponding candidate IUT behavior (i.e. function in set C). In the second one, we will assume that the feasibility of a possible IUT behavior is determined by the first step where a faulty machine output occurs. Besides, in order to explore a scenario that is particularly negative to the testing productivity, in the third example we will assume that all possible IUT behaviors expose, at most, one faulty machine output (which is replied after the IUT receives a specific sequence of machine inputs). Note that assigning a 0 weight to some possible IUT behaviors (in particular, to all behaviors exposing more than one faulty machine output) is equivalent to reducing the set C itself. As we will see, this structural difference between this example and the others will result in a quite different asymptotical testing complexity.

More precisely, the sets of testing hypotheses under consideration in these three examples will be combinations of some of the following hypotheses:

(a) **Independency hypothesis (IH)**. For every two edges X, Y of the execution tree of the IUT, we assume that the event that X produces a faulty output is independent from the event that Y produces a faulty output. Note that this is not true in general, as two different edges X and Y of the execution tree could represent executing the *same* logical structure of the IUT (e.g. executing the same C++ instruction line, calling the same function, traversing the same FSM transition, etc), and faults happen precisely because the IUT executes some specific incorrectly defined logical structures. Considering this independence between faults in the execution tree is equivalent to assuming that the dependence between faults is too low to deserve being explicitly represented. The tester could come to this conclusion for several reasons: maybe there are too many possible logical structures (code lines, transitions, etc) to think that explicitly representing these dependencies will significantly affect the calculation of the resulting testing complexity; or the number of available logical structures is low but the rest of the configuration that affects the IUT (variable values, full memory state, etc) influences the IUT decisions so much that repeating a fault is not likely, even if the same logical structure is traversed again; or the tester does not have enough information about the logical structures so a kind of default case (i.e. ignoring these dependencies) has to be adopted instead; or she has that information, but properly representing it into the fault model is too time-consuming, so it is acceptable to adopt a coarse-grained model letting to approximate the actual testing complexity.

(b) **Almost correct hypothesis (ACH)**. The IUT is nearly correct, so its execution tree can contain only up to one edge where the output is incorrect. Again, this assumption is not true in general, as even a single incorrect logical structure can be the source of faults in several different situations (that is, in several edges of the execution tree). Assuming there is up to one faulty edge is equivalent to assuming that the IUT is nearly correct *and* faults are so much context-dependant that the probability of repeating the same fault in two different contexts is too low to significantly affect the calculated testing complexity. This is a paradigmatic bad case for a tester, as faults are extremely difficult to find in this scenario.

(c) **Mostly initially correct hypothesis (MICH)**. The IUT developer (e.g. a programmer) already applied and passed several informal test cases of random lengths to the IUT. This means that faults at small depths in the tree are less likely than faults at deeper faults, though the probability of faults raises smoothly with the depth of faults, rather than abruptly. Hence, the potential faults of the IUT distribute all along the execution tree and they can simultaneously happen in any combination, though they are less likely at the first execution steps. In our model, we will assume that each fault reduces the weight of the corresponding function, and faults close to the root of the execution tree reduce it even more.

(d) **Initially correct hypothesis (ICH)**. The IUT developer already applied and passed some tests to the IUT although, contrarily to (c), they were performed in a more or less systematic way up to some unknown fix depth. Hence, we expect the faults of the IUT to be abruptly spread beyond some unknown depth. In our model, we will assume that the weight of each function is affected by the *first* level of the execution tree where there is a fault.

(e) **The deeper the better hypothesis (DBH)**. In a nearly correct IUT with at most one fault, we assume that the probability of observing a fault falls as the execution gets deeper into the execution tree. The rationale of this assumption is that, after performing a long execution without observing any fault, the probability of observing a fault that has not been unveiled before is lower, as the amount of sources of potential faults that have not been checked yet is reduced. This means that observing deeper edges of the execution tree is less interesting than observing edges that are closer to the root. In our model, this will be represented by assigning a lower weight to functions whose single fault is deeper into the tree.

Our first example, representing the activity of a typical implementer, will assume **IH** and **MICH**. In our second example, we will suppose that the IUT implementer is equally competent when writing code as in the first example, though he was very organized when checking his work afterwards, so we will assume **IH** and **ICH**. In our third example, we will assume that the implementer was extremely competent, so we will consider **ACH** and **DBH**. These examples illustrate three different paradigmatic testing scenarios which will potentially lead to different difficulty to the tester (as we will see, there will be a difficulty gap between the first two examples and the third). Note that keeping our three examples independent from any particular system definition means they do not particularly represent any specific case, although it increases their applicability to similar cases: since these examples do not depend on the form of the particular system under consideration (as the probabilities of faults in different edges of the execution tree are assumed to be independent from each other), our conclusions will be valid to any other similar testing frameworks regardless of the form of the system —provided that the same premises can be assumed. So, the complexities we will calculate for these three cases can be considered as the complexities *by default* of any other testing frameworks where the same hypotheses can be assumed, before any specific knowledge of the form of the system under consideration (code instructions, transitions, etc) is taken into account.

Let us now formally define a common framework for our three examples. We consider an input alphabet $\{a, b\}$ and an output alphabet $\{0, 1\}$; a specification $E = \{f_0\}$ with only one correct function f_0 whose weight is 1; testing strategies that apply test suites of the form $\{a, b\}^n$, that is, all sequences of machine inputs of a certain length n (different testing strategies will be considered in Sect. 5); and $\sigma = [C^1, C^2, ...]$ defined as follows: C^m is the set of functions that, for each possible combination of answers to $\{a, b\}^m$, contains (i) f_0 if f_0 produces that combination of answers, or (ii) the function computed by the minimum machine represented by C (minimum according to some arbitrary criterion, e.g. minimum number of states in FSMs, minimum number of lines in Java programs, etc.) producing that combination, if not. Clearly, $C^i \subseteq C^{i+1}$ for all $i > 1$, and we also have $\bigcup_{i\in\mathbb{N}^+} C^i = C$ as required.[5] Also note that sequence σ is *natural*.

In [23] it was proved that, when C represents deterministic FSMs in particular, the pair (C, E) is in `Class II`. Regardless of the computation model represented by C, in order to obtain the *testing complexity* of the associated testing problem, we need to define two additional ingredients: the weight function w (assigning a weight to each function) and a test suite cost function tsc (assigning a cost to each test suite). Initially, the cost function will be just the number of inputs (test cases) in the test suite, i.e. $tsc(\mathcal{I}) = |\mathcal{I}|$, so $tsc(\{a, b\}^n) = 2^n$.

Given a strictly increasing function $g : [1, \infty) \to \mathbb{N}$ (so that g^{-1} exists and is also strictly increasing), we assume a testing strategy s_g given by $s_g(u) = \{a, b\}^{\lceil log(g(\frac{1}{1-u}))\rceil}$. In

5. In particular, property $\bigcup_{i\in\mathbb{N}^+} C^i \supseteq C$ holds because, for all $f \in C$, the minimum machine of the considered kind computing f must be the minimum one giving its particular combination of answers to $\{a, b\}^m$ for some m, so $f \in C^m$ for some m (note that, if not, then *all* answers of f would be the same as those of some $f' \in C^r$ for all $\{a, b\}^k$ with $k \in \mathbb{N}^+$, as there is a finite number of machines being smaller than the minimum machine computing f, so $f = f'$ and $f \in C^r$ indeed, a contradiction).

particular, notice that $cost(r) = tsc(s_g(1-\frac{1}{r})) = 2^{log(g(r))} = g(r)$. Therefore, if we prove that the strategy s_g is *almost complete* up to 1 then the problem can be tested up to DR 1 with testing complexity g.

In order to prove that s_g is almost complete up to 1, we need to compute (an under-approximation of) the distinguishing rate of the test suites it returns, and prove that for all $u < 1$ and for all m large enough we have

$$\texttt{d-rate}\,(s_g(u), C^m, E^m, w) \geq u$$

If $k = \frac{1}{1-u}$, this condition is equivalent to

$$\texttt{d-rate}\left(\{a,b\}^{\ulcorner log(g(k))\urcorner}, C^m, E^m, w\right) \geq 1 - \frac{1}{k}$$

and if $n = \ulcorner log(g(k))\urcorner$, this is in turn is implied by

$$\texttt{d-rate}\,(\{a,b\}^n, C^m, E^m, w) \geq 1 - \frac{1}{g^{-1}(2^n)} \qquad (2)$$

as $1-\frac{1}{k} \leq 1-\frac{1}{g^{-1}(2^n)}$ (note that $k \leq g^{-1}(2^n)$). Thus, proving Eq. 2 for all m large enough will suffice to prove that s_g is almost complete up to 1 and that the corresponding problem can be tested up to DR 1 with testing complexity g.

Alternatively, we will also consider that the cost of a test suite is the total number of *machine inputs* applied by all tests in the suite, that is, $tsc'(\mathcal{I}) = \sum_{t\in\mathcal{I}} |t|$, so $tsc'(\{a,b\}^n) = n \cdot 2^n$. Notice that $tsc' = h \circ tsc$, where $h(n) = n \cdot log(n)$. Notice also that h satisfies Eq. 1, that is, for every c, $h(c{\cdot}n) \in \mathcal{O}(h)$. Indeed, for every $k \geq 2c$ and every $n \geq c$ it holds $h(c{\cdot}n) \leq k{\cdot}h(n)$. By Cor. 1, if we obtain a testing complexity in $\mathcal{O}(f(n))$ using tsc then we automatically have a testing complexity in $\mathcal{O}(f(n) \cdot log(f(n)))$ using tsc'.

4.1 First example: a typical implementer who already performed some informal tests

As mentioned before, in this example we will assume hypotheses **IH** and **MICH**. Let us denote by $f^i(t)$ the $i-$th machine output produced by $f \in C$ for input (i.e. test) t, that is, $f(t) = (f^1(t), f^2(t), ..., f^{|t|}(t))$. Let us take any p with $0 \leq p < \frac{1}{2}$ and define

$$w(f) = 1 - \sum_{t\in\{a,b\}^*, f^{|t|}(t)\neq f_0^{|t|}(t)} p^{|t|}$$

That is, from an initial weight of 1, p is subtracted for each wrong machine output replied by f to a machine input given at the first step, p^2 is subtracted for each wrong machine output of f replied to a machine input at the second step, and so on. Thus, this model assumes that machines with many faults are less likely, and that faults at the first steps of the machine behavior are less likely (i.e. they reduce more the weight) than faults at further steps. This amounts to consider **MICH**, that is, a competent implementer who, in particular, is less likely to commit faults at the first steps of the machine behavior, as his own informal tests are more likely to have already checked the machine behavior for short interactions. For instance, $w(f_0) = 1$, whereas a completely faulty machine has weight $1-p-p-p^2-p^2-p^2-p^2-\ldots = 1-2p-4p^2-8p^3-16p^4-\ldots$ Notice also that, when $p = 0$, all functions have the same weight, 1.

Let us recall that, in the three examples considered here, our testing strategy consists in applying test suites of the form $\{a,b\}^n$ (alternative testing strategies will be studied in Sect. 5). Let $p = 1/4$. In the appendix we prove that we can bound the DR in this case as follows:

$$\texttt{d-rate}\left(\{a,b\}^n, C^l, E^l, w\right) \geq 1 - \frac{k_1}{2^{2^n}} \qquad (3)$$

for some constant k_1, which implies that the testing complexity of this strategy is in $\mathcal{O}(log(n))$ if the cost of test suites is the number of inputs (test cases) it contains (tsc). Therefore, if alternatively the cost is the total number of machine inputs in all test cases (tsc'), then we have a testing complexity in $\mathcal{O}(log(n) \cdot log(log(n)))$.

In the appendix, the same asymptotic complexities are found, respectively, for the case where $p = 0$ (i.e. the case where all functions have the same weight).

4.2 Second example: a typical implementer who already performed some systematic tests

Let us assume hypotheses **IH** and **ICH**. Consequently, let us consider the following alternative weight function:

$$w(f) = 1 - \frac{1}{2^{err(f)-1}}$$

where $err(f) = min\{|t| \mid t \in \{a,b\}^*, f^{|t|}(t) \neq f_0^{|t|}(t)\}$. Intuitively, $err(f)$ denotes the first step where f produces a wrong machine output. Thus, now we are assuming **ICH**, so that the weight loss of each function is determined by the first step where a fault occurs, and subsequent faults do not contribute to further reduce the feasability of the behavior denoted by the corresponding function. For instance, in case there is a fault at the first level (first step), we have $w(f) = 1 - \frac{1}{2^{1-1}} = 0$. Analogously, if the first fault appears in the second level, we have $err(f) = 2$ and $w(f) = 0.5$, while $w(f) = 0.75$ in case the first fault of f appears in the third level.

We also prove in the appendix that, in this case, the DR can be bound as follows:

$$\texttt{d-rate}\left(\{a,b\}^n, C^l, E^l, w\right) \geq 1 - \frac{k_1}{2^{2^n}} \qquad (4)$$

for some constant k_1. Thus, the testing complexity of the considered strategy is again in $\mathcal{O}(log(n))$ when the cost of each test suite consists in the number of test cases. Consequently, it is also in $\mathcal{O}(log(n) \cdot log(log(n)))$ if the cost is the total number of applied machine inputs.

4.3 Third example: a very competent implementer (so further testing is finding a needle in a haystack)

Let us now go one step further into the competent programmer hypothesis, and let us assume **ACH**, so that each machine contains at most one fault (i.e. at most one machine output can be wrong in its behavior tree).

Moreover, let us also assume **DBH**, so that functions exposing their (single) fault at a further step are considered *less* likely. The intuition behind this alternative view is that, if the single fault does not appear after some long interactions, then it is less likely to appear after even longer interactions. In fact, if faults do not appear after very long interactions, it is more likely that the implementation is completely correct.

Formally, we define

$$w(f) = \begin{cases} 1 & \text{if } f = f_0 \text{ (f is correct)} \\ p^{err(f)} & \text{if } f \neq f_0 \text{ (f is incorrect)} \end{cases}$$

Therefore, a function denoting an incorrect machine has weight p^i assuming that its single fault appears in level i.

Now the distinguishing rate is significantly smaller than in the previous examples. Indeed, if we take p of the form $p_r \triangleq \frac{1}{2^{r+1}}$ for $r > 0$, we prove (see details in the appendix) that

$$\texttt{d-rate}\left(\{a,b\}^n, C^l, E^l, w\right) \geq 1 - \frac{k_1}{2^{n\cdot r}} \qquad (5)$$

for some constant k_1, which implies that the testing complexity of the strategy is in $\mathcal{O}(n^{1/r})$ when the cost function is tsc (i.e. number of test cases). In particular, for $r = 1$ (hence $p_r = 1/4$), we obtain a linear complexity, and for $r = 2$ (hence $p_r = 1/8$), we obtain a complexity in $\mathcal{O}(\sqrt{n})$. Alternatively, if the cost function is tsc' (total number of applied machine inputs), then the testing complexity is in $\mathcal{O}(n^{1/r} \cdot log(n^{1/r}))$.

5 Measuring the complexity of testing problems

For decades, researchers have been calculating the complexity of algorithms (i.e. the complexity of specific solutions to specific problems) and problems (the complexity of *any* solution to a specific problem). Note that the relation between strategies and problems in our testing framework is similar to the relation between algorithms and problems in computation: in both cases, the former are specific ways to tackle the latter. After measuring the complexity of some strategies for their corresponding testing problems in Sect. 4, next we will show how to identify the complexity of the testing problems themselves.

We already found that the complexity of some strategies is under some thresholds. By computing some underestimation of the DR of each strategy, we overestimated the costs of the test suites returned by that strategy to reach each required DR. For instance, we know that the complexity of the strategy given in Section 4.2 is at most logarithmic when the cost is the number of test cases, though it could be lower. This proves that the testing *problem* faced by that strategy can be solved with logarithmic complexity, but this is not enough to prove that it cannot be solved better: even if the complexity of that strategy were not overestimated at all, other strategies could provide a complexity being better than logarithmic. As it happens with algorithms and problems, reasoning about the complexity of a testing problem will generally be harder than reasoning about the complexity of a specific strategy to tackle the problem. Here we will do it by reasoning about the *optimal* testing strategy (i.e. the strategy that, given any target DR to be reached, returns the cheapest test suite reaching that DR), as the complexity of the corresponding problem cannot be better.

Definition 12. Let s be a testing strategy that is almost complete up to u. The strategy s is *optimal* for u if, for any other strategy s' that is almost complete up to u, we have $tsc(s(\alpha)) \leq tsc(s'(\alpha))$.

A testing problem is in $\Theta(f)$ for u if it has an optimal testing strategy for u with testing complexity in $\Theta(f)$. □

As usual, u will be omitted when $u = 1$.

5.1 Example revisited: typical implementer who performed some systematic tests

Let us reconsider our previous example in Sect. 4.2, where the weight function was defined as $w(f) = 1 - \frac{1}{2^{err(f)-1}}$. Next we will find out the complexity of this *problem*, rather than the complexity of a specific strategy to this problem as we did before. In order to find the problem complexity, we will directly tackle the optimal strategy for the problem under consideration. As we will see, this strategy will consist in applying a *single* long test case. Here, we will concentrate on measuring the cost of test suites in terms of the total number of machine inputs applied by all tests in the suite, that is, we will consider cost function tsc' (note that measuring the testing complexity in terms of number of test cases, i.e. with function tsc, is trivial and not very interesting if test suites consist of a single test case).

In our previous approach to this problem we considered a strategy applying all tests in $\{a,b\}^n$, i.e. the strategy was balanced and exhaustive up to some length n. Note that, due to the nature of the problem, the weight of each function in C is higher if its first error (i.e. the first point of its execution tree where the output is wrong) appears later (i.e. farther from the root of that tree), as errors at the first execution steps are considered less likely. In addition, note that any two test cases sharing some non-null prefix are redundant to some extent. For instance, after applying a test aaa, the test aab can detect a new fault only at its third step, as the first two steps observe a part of the behavior of the IUT that was already observed before (recall that systems are assumed to be deterministic in these examples). Hence, if we wish to spend e.g. 6 machine inputs in a test suite, then the test suite $\{aaa, aab\}$ is less interesting that the suite $\{aaaaaa\}$ or the test suite $\{aaa, bbb\}$. Given the facts that (a) applying machine inputs farther allows to distinguish functions with more weight, and that (b) shared prefixes between tests do not contribute to provide more DR, we conclude that, given some number of machine inputs to be applied, the optimal testing strategy for this problem is to apply a single very large test (instead of a balanced test suite like the one mentioned in our previous approach). That is, instead of using 2^k tests of length k for some k, the best strategy is using a single test of length $k \cdot 2^k$ (note that the cost of both test suites is the same): by doing so, the highest DR will be reached.

Let us consider the testing strategy containing a single test a^n. In this case, we prove (see the appendix) that

$$\texttt{d-rate}\left(\{a^n\}, C^l, E^l, w\right) \geq 1 - \frac{k_1}{2^{2^n}} \qquad (6)$$

for some constant k_1. Interestingly, the minimum distinguishing rate of the test suite $\{a^n\}$ identified by the previous expression is the same as the minimum DR identified for the test suite $\{a,b\}^n$ in Eq. 4 (up to the different values of local constants k_1). Hence, if we consider that the cost of a test suite is the total number of machine inputs applied by all test cases (n in test suite $\{a^n\}$), then the complexity of the optimal testing strategy for this problem, and thus the complexity of the problem itself, is in $\mathcal{O}(log(n))$. Notice that this is slightly better than the complexity $\mathcal{O}(log(n) \cdot log(log(n)))$ achieved by using the test suite $\{a,b\}^n$, which was previously calculated in Sect. 4.2.

5.2 Example revisited: a very competent implementer

Next we will find the complexity of a testing problem by *indirectly* reasoning about its optimal strategy, instead of actually constructing it (which was the approach we followed in the previous problem). As we will see, this will be enough to compute the complexity of the testing problem.

Let us revisit the testing setting given in Section 4.3, that is, each incorrect function is faulty because it produces a wrong machine output in a *single* edge of its tree of possible executions, and the weight of each incorrect function is $(p_r)^l$ if that single fault is at depth l of that tree (recall from Sect. 4.3 that $p_r = \frac{1}{2^{r+1}}$ for some given $r > 0$). Let us also remind that the testing strategy proposed in Section 4.3 returned test suites of the form $\{a,b\}^k$, i.e. all inputs of some length k were tested. Next we will assume that the cost of test suites consists in the number of machine inputs tested by all test cases in the test suite (instead of just the number of test cases). That is, we will consider cost function tsc'.

Let us fix $r = 1$, so $p_r = p_1 = \frac{1}{4}$. Let us see that, for any n being the inverse of the maximum allowed distance between the reached DR and completeness (e.g. if $n = 8$ then the target DR is $1 - \frac{1}{8} = \frac{7}{8}$), the smallest test suite of the form $\{a,b\}^k$ reaching that DR is $\{a,b\}^{\ulcorner log\ n \urcorner}$.

Lemma 1. The least k such that $\texttt{d-rate}\left(\{a,b\}^k, C^l, E^l, w\right) \geq 1 - \frac{1}{n}$ is $k = \ulcorner log\ n \urcorner$.

Proof Sketch: Let us illustrate the general argument for all n by firstly reasoning for some particular case, say $n = 8$. In this case, the test suite $\{a,b\}^3$ distinguishes all incorrect functions failing only at the first step (two functions, one failing for a and the other for b, whose weight amounts $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$), all functions failing at the second step (four functions whose total weight is $4 \cdot \frac{1}{4^2} = \frac{1}{4}$), and all failing at the third ($8 \cdot \frac{1}{4^3} = \frac{1}{8}$), so the total weight of all pairs of functions (formed by the only correct function and some incorrect one) distinguished by $\{a,b\}^3$ is $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 1 - \frac{1}{8}$. Note that the sum of weights of *all* possible pairs of correct and incorrect functions within each C^m tends to 1 as m tends to ∞ ($\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$). Hence, the smallest test suite of the form $\{a,b\}^k$ making the inverse of the distance to completeness be at least 8 for *all* C^m is $\{a,b\}^3$ —and trivially by induction over k, the smallest test suite of the form $\{a,b\}^k$ making that measure be at least n is $\{a,b\}^{\ulcorner log\ n \urcorner}$. □

Note that the cost of the test suite $\{a,b\}^{\ulcorner log\ n \urcorner}$ is $2^{\ulcorner log\ n \urcorner} \in \mathcal{O}(n)$ if costs consist in the number of applied test cases, and $2^{\ulcorner log\ n \urcorner} \cdot \ulcorner log\ n \urcorner \in \mathcal{O}(n \cdot log\ n)$ if costs count the number of machine inputs in the test suite. Thus, the testing complexities $\mathcal{O}(n)$ and $\mathcal{O}(n \cdot log\ n)$, previously calculated in Sect. 4.3 for this testing strategy, are exactly $2^{\ulcorner log\ n \urcorner}$ and $2^{\ulcorner log\ n \urcorner} \cdot \ulcorner log\ n \urcorner$, respectively.

Let us introduce some notation for test suites. It will be based on viewing each test suite as the part of the execution tree that is covered by the test cases in the suite. For instance, the test suite $\{a,b\}^k$ is seen as a tree of depth k where edges leading to the left child are labelled by a and edges leading to the right child are labelled by b.

Definition 13.

- A suite is *complete up to* depth l if all branches reach level l.
- A suite is *balanced* (or *complete*) if all branches reach the same depth.
- A suite is *linear* it it has a single branch.
- An edge of a suite is a *bifurcation* if the node it departs from has two children.
- An edge is an *extension* if the node it departs from has one children.
- A node of a suite is *bifurcated* if it has two children and is *e*xtended if it has one. It is a *leaf* if it has no children.
- We say that a tree (i.e. a suite) is *optimal* if, for the DR it reaches, there exists no other tree reaching at least that DR with lower cost.

□

We will consider that the root node is at depth 0, and each d-depth edge connects a $(d-1)$-depth node to a d-depth node. Also, for any tree T we will denote by $DR(T)$ and $Cost(T)$ the DR and the cost of the test suite represented by T, respectively.

Let us reason about the optimal strategy for our problem, i.e. a function that, given n, returns the cheapest test suite (i.e. optimal tree) reaching a DR $1 - \frac{1}{n}$. Despite the fact that the setting of our example treats equally machine inputs a and b (that is, left and right edges in the execution tree), optimal trees are *not* balanced in general, as shown in the next example.

Example 1. Let $A = \{a,b\}^5$ and $B = (\{a,b\}^5 \backslash \{aaaaa, aaaab, aaaba, aaaabb, aabaa, aabab\}) \cup \{aaaaaa, aaaaba, aaabaa, aaabba, aabaaa\}$. Both A and B have the same number of machine inputs ($32 \cdot 5 = 32 \cdot 5 - 6 \cdot 5 + 5 \cdot 6 = 160$). By following a similar reasoning as we did in the proof of the previous lemma, the total weight of all incorrect functions distinguished by A is $1 - \frac{1}{32} = \frac{31}{32}$, whereas the total weight of those distinguished by B is $\frac{31}{32}$ minus $\frac{1}{1024}$ (since $aabab$ is removed and we do not add any test case extending it) plus $5 \cdot \frac{1}{4096}$ (as we extend the other five test cases with one more machine input), which amounts more than $\frac{31}{32}$. Hence, the unbalanced test suite B provides more DR than the balanced test suite A with the same cost. On the other hand, optimal trees are not linear in general either: the test suite $\{a,b\}$ reaches a $\frac{1}{4} + \frac{1}{4}$ DR, whereas the test suite $\{aa\}$, also with 2 machine inputs, reaches $\frac{1}{4} + \frac{1}{16}$ DR, which is lower.

Hence, contrarily to what one could think at first glance, in general optimal test suites for this testing problem are neither balanced nor linear. In the next results we characterize the cost of the optimal test suites without actually constructing them. We start with the following lemma.

Lemma 2. If an optimal tree is complete up to depth d but not complete up to depth d' for any $d' > d$, then it cannot contain any bifurcation from depth $d+2$ on.

Proof: Let us assume it contains such a bifurcation at depth $d+2$ or higher. In order to obtain a contradiction we build from it a cheaper tree with a higher DR by cutting the subtree pending from such bifurcation and pasting it from the shallowest available non-bifurcated node (i.e. extended or leaf), which must be at level $d+1$. The cost of the resulting tree cannot be higher than the cost of the original tree (actually it can be lower, as reaching all moved edges from the root requires now traversing less previous nodes, which reduces the number of edges traversed several times by the test suite). However, the DR of the resulting tree is higher, as all moved edges are shallower than before, and shallower edges provide more DR (note that an edge at level l distinguishes the correct function from an incorrect function whose weight is ${p_r}^l$ with $p_r < 1$, so the weight of the distinguished function decreases with l). Thus, the tree with some bifurcation at level $d+2$ or beyond was not optimal, which is a contradiction. □

Definition 14. Given $d > 0$, we denote by D_d and $Cost_d$ the DR and the cost, respectively, of the tree that is complete up to depth d and contains no more edges. □

Proposition 2. Let T be an optimal tree that is complete up to depth d but not from depth $d+1$ on. Then:

1) $D_d \leq DR(T) < D_{d+2}$, and
2) $Cost(T) \geq Cost_d$.

Proof: Since T is complete up to depth d, we know that $DR(T) \geq D_d$, and $Cost(T) \geq Cost_d$. Let us now see that $DR(T) < D_{d+2}$. Note that the DR contributed by any sequence of consecutive extensions departing from level l is lower than the DR contributed by any single edge at any depth $d < l$: the former is at most $\sum_{i=l}^{\infty}(\frac{1}{2^{r+1}})^i$, which is lower than $\frac{1}{2^l}$ due to $r > 0$ (recall that we are considering in particular $r = 1$), but the latter is $\frac{1}{2^d}$.

By Lemma 2 we know that tree T does not contain any bifurcation from depth $d+2$ on. Let us find an upper bound of the DR of this tree. For all sequences of consecutive extensions of this tree from depth $d+3$ on, let us remove the sequence and add a new bifurcation departing from a $(d+1)$-depth node, in particular from the *parent* of the node the removed sequence departed from (so this $(d+1)$-depth parent node becomes a bifurcation node after receiving its new $(d+2)$-depth bifurcation). By the property in the previous paragraph, each of these changes does not reduce the DR of the tree. After performing these changes, the resulting new tree is a subset of the tree that is complete up to depth $d+2$ and contains no more edges. Actually, it is a *strict* subset, as the $(d+1)^{th}$ level of *edges* was not complete before the modification and this has not changed. Thus, the DR of the resulting tree is less than D_{d+2}. □

For any $d \in \mathbb{N}$, let us reason about the *optimal* tree that reaches DR D_d.

Proposition 3. Let T be an optimal tree for DR D_d. Then

$$Cost_{d-2} \leq Cost(T) \leq Cost_d$$

Proof: Property $Cost(T) \leq Cost_d$ trivially holds because $DR(T) \geq D_d$ and T is optimal. Let us see that $Cost_{d-2} \leq Cost(T)$. By Prop. 2, we know that any optimal tree T' that is complete only up to depth $d-2$ cannot reach DR D_d. We also know that the cost of T' is at least $Cost_{d-2}$. We conclude that the cost of T (which is the cheapest tree reaching DR D_d) cannot be lower than $Cost_{d-2}$: if it were, then it would provide more DR than T' with a lower cost, so T' would not be an optimal tree, hence obtaining a contradiction. □

For the particular case of $p_r = \frac{1}{4}$ $(r = 1)$, we saw before that the strategy consisting in applying test suites of the form $\{a, b\}^k$ (instead of the optimal ones) returns, for each target n, the test suite $\{a, b\}^{\ulcorner log\ n \urcorner}$, and the cost of this test suite is $2^{\ulcorner log\ n \urcorner} \cdot \ulcorner log\ n \urcorner$. Let $v = \ulcorner log\ n \urcorner$. Since $\{a, b\}^{\ulcorner log\ n \urcorner}$ is the test suite that is complete up to depth $\ulcorner log\ n \urcorner = v$, its DR and cost are actually D_v and $Cost_v$, respectively, so we have $Cost_v = 2^v \cdot v$. On the other hand we know that, in order to reach a D_v DR, the cost of the *optimal* strategy is *at least* $Cost_{v-2} = 2^{v-2} \cdot (v-2)$.

For each DR D_v, the proportion between the cost of the optimal tree reaching that DR and the cost of the complete tree reaching it is at least $\frac{2^{v-2} \cdot (v-2)}{2^v \cdot v} = \frac{1}{4} \cdot \frac{v-2}{v} = \frac{1}{4} \cdot \frac{\ulcorner log\ n \urcorner - 2}{\ulcorner log\ n \urcorner}$. Note that for any $v \geq 10$ (or $n > 512$) this proportion will be higher than or equal to $\frac{1}{5}$. We conclude that, for any DR D_v with a high enough v, the complexity of the optimal strategy is at most a constant factor times lower than the complexity of the strategy returning complete trees (which is $2^{\ulcorner log\ n \urcorner} \cdot \ulcorner log\ \ n \urcorner$). Even though we have not reasoned about the complexity of the optimal strategy for all other DR (i.e. those which are not equal to D_v for some v), for all $v \in \mathbb{N}$ and DR α with $D_{v-1} \leq \alpha < D_v$ the cost of the cheapest tree reaching DR α cannot be lower than the cost of the cheapest tree reaching D_{v-1}, as the complexity function cannot be decreasing. We conclude that, for all high enough n, the complexity of optimal strategy is at least $2^{\llcorner log\ n \lrcorner} \cdot \llcorner log\ \ n \lrcorner$ (the cost of the complete tree reaching D_{v-1}) divided by some constant factor, which in turn proves that the complexity of the optimal strategy is in $\Omega(n \cdot log\ n)$. Since there is a strategy for this problem which is in $\mathcal{O}(n \cdot log\ n)$ indeed (that is, using complete trees), we conclude that the complexity of the optimal strategy for this testing problem is in $\Theta(n \cdot log\ n)$.

Corollary 2. Let $p_r = \frac{1}{4}$ and let us consider cost function tsc'. The testing problem in Sect. 4.3 has a testing complexity $\Theta(n \cdot log\ n)$.

Therefore, even though the strategy that applies complete test suites is not optimal, it is not a bad strategy in asymptotic terms, as it belongs to the same complexity class as the optimal strategy. It is worth to point out that discovering this did not require finding out the actual form of optimal trees (note that our previous lemmas and propositions just state some properties necessarily fulfilled by optimal trees, but never identify these trees).

6 Conclusions and future work

In this paper we have introduced the notion of complexity to formal testing. Inspired by how designers asymptotically measure the complexity of an algorithm in terms of the time or the memory it uses (or the complexity of *any* algorithm solving a problem), the proposed general testing framework lets us measure how fast our partial certainty on the IUT correctness increases as long as the IUT passes the test cases we apply to it.

We have studied the testing complexity of several examples differing in their testing assumptions. These assumptions concern how the likelihood of each possible definition of the IUT depends on the number of faults its execution can expose, or how long we would have to interact with the IUT in order to find them. The problems in the first and second examples assume that the IUT may expose any number of faults, and both show a logarithmic testing complexity for testing strategies involving balanced test suites: in this case, the number of test cases to be applied by the test suite increases logarithmically with the inverse of the gap up to a full certainty on the IUT correctness. On the other hand, in the example where the behavior of the IUT exposes a single faulty machine output (the third example), the resulting testing complexity of the same strategy may be linear, proportional to the square root, or proportional to any other subsequent root, depending on how the likelihood of functions grows with the number of steps before the single fault. Intuitively, in this case faults are much harder to detect, so testing is a quite less productive activity (in particular, many more test cases are required to reach the same certainty levels on the IUT correctness). These testing complexities are measured in terms of the number of applied test cases. Alternatively, if we count the total number of machine inputs applied by all test cases in the test suite, then the previously mentioned complexities are multiplied by the logarithm of themselves.

Following the dichotomy between the complexity of an algorithm and the complexity of the problem solved by that algorithm, the previous testing properties are on the "algorithm" side (in our case, on the *testing strategy* side): they show that there exist testing strategies providing these complexities —but this does not imply that better strategies cannot be found. In order to illustrate how the testing complexity of *problems* can be analyzed, we have identified the testing complexity of two of our previous testing problems, in particular for the case where the cost of a test suite is the total number of machine inputs applied by all test cases in the suite. In one case, the actual optimal strategy for the problem revealed a slightly better complexity than that achieved by our previously considered strategy. In the other case, an indirect reasoning about the properties of any optimal strategy (instead of the direct identification of the optimal strategy itself) allowed us to identify the Θ complexity of the problem, and showed us that the $\mathcal{O}$ complexity of our previous (non-optimal) strategy belongs to the same complexity class, which makes it a reasonable strategy despite its known lack of optimality.

Hopefully, the identification of the testing complexity of *problems* could lead us to characterize *hard* problems for some testing complexity classes, i.e. problems such that, if they can be solved with some testing complexity, then a whole class of testing problems can (similarly as it happens for NP-hard problems in time complexity). The testing reduction notion defined in [23] already lets reduce a testing problem into another. Unfortunately, it just translates complete test cases from a problem to another, so it fully ignores any testing complexity notion. Clearly, this reduction notion should be adapted to properly handle and preserve the testing complexity after the translation (rather than just preserving the completeness of the corresponding test suites), in the same way as some special reductions partially preserve, in Complexity theory, the approximability class between optimization problems (e.g. AP-reductions).

APPENDIX
PROOFS OF EQUATIONS IN SECTIONS 4 AND 5

In this appendix we present the proofs of all equations mentioned and not proved in previous sections.

Recall that, given a function $g : [1, \infty) \to \mathbb{N}$, we assume a testing strategy s_g given by $s_g(u) = \{a, b\}^{\lceil log(g(\frac{1}{1-u}))\rceil}$, so that $cost(r) = g(r)$. Therefore, if we prove that the strategy s_g is *almost complete* up to 1 then the problem can be tested up to DR 1 with testing complexity g.

In order to prove that s_g is almost complete up to 1, we prove that Eq. 2 is satisfied, that is,

$$\texttt{d-rate}\,(\{a, b\}^n, C^m, E^m, w) \geq 1 - \frac{1}{g^{-1}(2^n)}$$

To further simplify the previous condition, given a weight function w, let us denote

$$T_m = \sum_{f \neq f_0} w(f) \text{ and } Ind_{n,m} = \sum_{\neg di(f, f_0, \{a,b\}^n)} w(f)$$

Hence, T_m is the accumulated weight of all incorrect functions in C^m and $Ind_{n,m}$ is the accumulated weight of all the functions in C^m that cannot be distinguished from f_0 using $\{a, b\}^n$ as test suite. With this notation, we can write

$$\texttt{d-rate}\,(\{a, b\}^n, C^m, E^m, w) = \frac{T_m - Ind_{n,m}}{T_m} = 1 - \frac{Ind_{n,m}}{T_m}$$

Note that there is only one correct function f_0 in E, so all pairs of correct and incorrect functions in the numerator and the denominator of the fraction denoting the DR (see Def. 7) are multiplied by the same removable factor. We conclude that $\texttt{d-rate}\,(\{a, b\}^n, C^m, E^m, w) \geq 1 - \frac{1}{g^{-1}(2^n)}$ if and only if

$$D_{n,m} \triangleq \frac{Ind_{n,m}}{T_m} \leq \frac{1}{g^{-1}(2^n)} \qquad (7)$$

Thus, we will use Eq. 7 to prove that each testing problem has complexity g.

Proof of Equation 3

For a fixed p with $0 \leq p < \frac{1}{2}$ we have

$$w(f) = 1 - \sum_{t \in \{a,b\}^*, f^{|t|}(t) \neq f_0^{|t|}(t)} p^{|t|}$$

By Eq. 7, in order to prove Eq. 3 (therefore obtaining a logarithmic complexity), it is enough to prove that

$$\frac{Ind_{n,m}}{T_m} \leq \frac{k_1}{2^{2^n}}$$

for some constant k_1, and define g as a logarithmic function (in particular, $g(x) = log(x) + log(k_1) = log(x) + k_2$ for some constant k_2, so $g(x) \in \mathcal{O}(log(x))$).

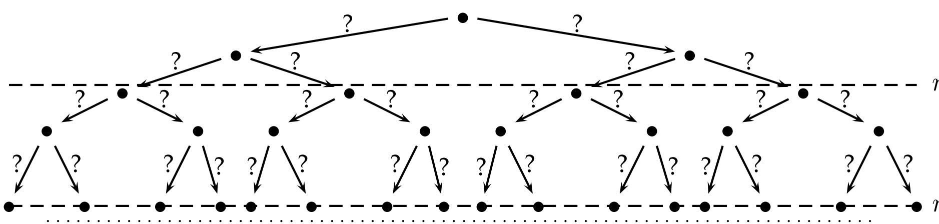

Figure 1. Computing $D_{n,m}$: when testing an IUT in C^m with test suite $\{a,b\}^n$, only behaviors up to step n are checked. Question marks denote possible machine outputs in response to machine inputs a (left branches) and b (right branches).

Let us first compute and bound T_m. As it can be seen in Fig. 1, in the tree denoting the behavior of a function up to the $m-$th step, the number of question marks (i.e. places where the function behavior can expose different machine outputs) is $2^{m+1}-2$, being m the depth of the tree of combinations. For each question mark, we have two possibilities, so the number of different combinations is $M_m \triangleq |C^m| = 2^{2^{m+1}-2}$.

We could think that, by adding together the weights of all the functions in C^m, and taking $q=2p$, we obtain

$$M_m - \frac{M_m}{2}p - \frac{M_m}{2}p - \frac{M_m}{2}p^2 - \frac{M_m}{2}p^2 - \frac{M_m}{2}p^2 - \frac{M_m}{2}p^2 - \ldots$$

$$= M_m - \frac{M_m}{2}(2p+4p^2+8p^3+\ldots) = M_m - \frac{M_m}{2}\sum_{i=1}^{\infty} q^i$$

However, the previous formula only adds the appropriate terms until $i=m$ (depth m). From that point on, we can not guess how many wrong machine outputs will be produced by each function, and how much weight must be subtracted for that reason. By the construction of C^m, for all functions in C^m we know that all faults at all steps up to the m-th step are produced by the same number of functions, as we include in C^m the "smallest" functions producing each possible behavior up to m steps (for some arbitrary size criterion). However, we do not know if these functions, in particular, are biased towards some combinations of behaviors over others for steps beyond the m-th. Given our lack of information regarding the weights lost due to faults beyond step m, we consider an *underestimation* of T_m where all behaviors beyond level m are faulty. Hence, we can bound T_m as follows:

$$T_m \geq M_m - \frac{M_m}{2}\sum_{i=1}^{m} q^i - M_m \sum_{i=m+1}^{\infty} q^i \qquad (8)$$

Since the third term in the addition is not divided by 2, it is assumed that everything is wrong from step $m+1$ on (thus, we are *underestimating* T_m). Let us now bound $Ind_{n,m}$. The number of question marks in the lower part of Fig. 1 (i.e. from step $n+1$ to step m) is $2^{m+1}-2-(2^{n+1}-2) = 2^{m+1}-2^{n+1}$. Thus, the number of functions that are indistinguishable from f_0 in C^m by $\{a,b\}^n$ is $N_{n,m} = 2^{2^{m+1}-2^{n+1}}$.

If we assume that each f indistinguishable from f_0 has no errors below level n (*overestimation* of $Ind_{n,m}$), we can bound $Ind_{n,m}$ as:

$$Ind_{n,m} \leq N_{n,m} - \frac{N_{n,m}}{2}\sum_{i=n+1}^{m} q^i \qquad (9)$$

Putting together Eq. 8 and Eq. 9 we can conclude that

$$\begin{aligned} D_{n,m} &= \frac{Ind_{n,m}}{T_m} \\ &\leq \frac{N_{n,m} - \frac{N_{n,m}}{2}\sum_{i=n+1}^{m} q^i}{M_m - \frac{M_m}{2}\sum_{i=1}^{m} q^i - M_m\sum_{i=m+1}^{\infty} q^i} \end{aligned} \qquad (10)$$

Case of functions with equal weights:

If $p=0$ then all functions have weight 1 and $q=0$. In this case, Eq. 10 boils down to $D_{n,m} = \frac{N_{n,m}}{M_m} = 2^{-2^{n+1}+2} = 2^2 \cdot 2^{-2^{n+1}} \leq k_1 \cdot 2^{-2^n}$ for some constant k_1 and we are done, as this proves the $\mathcal{O}(log(n))$ complexity if the cost function is tsc (if it is tsc', then the complexity is in $\mathcal{O}(log(n) \cdot log(log(n)))$).

General case:

For $p < \frac{1}{2}$ (and hence $q<1$), let us recall that:

$$\sum_{i=n+1}^{m} q^i = \frac{q^{n+1}-q^{m+1}}{1-q} \qquad \sum_{i=1}^{m} q^i = \frac{q-q^{m+1}}{1-q}$$

$$\sum_{i=m+1}^{\infty} q^i = \frac{q^{m+1}}{1-q}$$

For $q=\frac{1}{2}$ (hence $p=\frac{1}{4}$, as required in Eq. 3), these expressions yield

$$\sum_{i=n+1}^{m} q^i = \frac{1}{2^n} - \frac{1}{2^m} \qquad \sum_{i=1}^{m} q^i = 1 - \frac{1}{2^m} \qquad \sum_{i=m+1}^{\infty} q^i = \frac{1}{2^m}$$

In this case, the expression in the right handside of Eq. 10 can be calculated as:

$$\frac{2^{2^{m+1}-2^{n+1}} - 2^{2^{m+1}-2^{n+1}-1-m}}{2^{2^{m-1}-2} - 2^{2^{m-1}-3} + 2^{2^{m+1}-3-m} - 2^{2^{m-1}-2-m}} = \begin{bmatrix}\text{factoring out} \\ 2^{2^{m-1}}\end{bmatrix}$$

$$\frac{2^{-2^{n+1}} - 2^{-2^{n+1}-m-1}}{2^{-2}-2^{-3}+2^{-m-3}-2^{-m-2}} = \qquad [2^{-m-2} - 2^{-m-3} = 2^{-m-3}]$$

$$\frac{2^{-2^{n+1}} - 2^{-2^{n+1}-m-1}}{2^{-3}-2^{-m-3}} < k_1 \cdot 2^{-2^{n+1}} < k_1 \cdot 2^{-2^n}$$

for some k_1 for a large enough m, and the thesis follows. Summing up, we obtain a logarithmic complexity if the cost of a test suite equals the number of inputs (test cases) in the test suite, and a complexity in $\mathcal{O}(log(n) \cdot log(log(n)))$ if the cost of a test suite is the total number of machine inputs applied by all test cases in the test suite.

Proof of Equation 4

Now we are considering the following weight function:
$w(f) = 1 - \frac{1}{2^{err(f)-1}}$
where $err(f) = min\{|t| \mid t \in \{a,b\}^*, f^{|t|}(t) \neq f_0^{|t|}(t)\}$, that is, $err(f)$ denotes the first step where f produces a wrong machine output.

Since $1 - \frac{1}{2^n} = \sum_{i=1}^{n} \frac{1}{2^i}$, we can calculate $w(f)$ by adding $\frac{1}{2^i}$ for each correct level i (rather than calculating 1 minus the penalty due to the first faulty level).

The number of functions in C^m that are correct up to level i is given by $2^{2^{m+1}-2^{i+1}}$: Indeed, there are $2^{m+1}-2$ question marks up to level m (i.e. output machines which can be either 0 or 1), and $2^{i+1}-2$ up to level i.

Thus, and assuming that all correct functions are incorrect at level $m+1$ (underestimation of T_m), we have:

$$\begin{aligned} T_m &= \sum_{f \in C^m} w(f) \geq \sum_{i=1}^{m} \left(2^{2^{m+1}-2^{i+1}} \cdot \frac{1}{2^i} \right) \\ &= 2^{2^{m+1}} \sum_{i=1}^{m} 2^{-2^{i+1}-i} \end{aligned}$$

In order to compute and bound $Ind_{n,m}$, we first observe (as we did above) that the number of functions in C^m that are indistinguishable from f_0 up to level n (i.e. by $\{a,b\}^n$) is $2^{2^{m+1}-2^{n+1}}$. Then,

$$Ind_{n,m} \leq \left(1 - \frac{1}{2^n}\right) 2^{2^{m+1}-2^{n+1}} + \sum_{i=n+1}^{m} \left(2^{2^{m+1}-2^{i+1}} \cdot \frac{1}{2^i} \right)$$

where the first summand is the sum of weight due to the fact that all functions accounted here are correct up to level n. Also, notice that we are assuming that all functions are correct up to level m (overestimation of $Ind_{n,m}$). Putting together the previous two equations, we can compute and bound $D_{n,m}$ as:

$$\frac{(1-\frac{1}{2^n})2^{2^{m+1}-2^{n+1}} + \sum_{i=n+1}^{m}\left(2^{2^{m+1}-2^{i+1}} \cdot \frac{1}{2^i}\right)}{2^{2^{m+1}} \sum_{i=1}^{m} 2^{-2^{i+1}-i}} = \qquad \left[\begin{matrix}\text{factoring out} \\ 2^{2^{m-1}}\end{matrix}\right]$$

$$\frac{(1-\frac{1}{2^n})2^{-2^{n+1}} + \sum_{i=n+1}^{m}\left(2^{-2^{i+1}} \cdot \frac{1}{2^i}\right)}{\sum_{i=1}^{m} 2^{-2^{i+1}-i}} < \qquad \left[\begin{matrix}\sum_{i=n+1}^{m} 2^{-2^{i+1}-i} < \\ < 2^{-2^n+1}\end{matrix}\right]$$

$$\frac{2^{-2^{n+1}} - 2^{-2^{n+1}-n} + 2^{-2^n+1}}{\sum_{i=1}^{m} 2^{-2^{i+1}-i}} < \qquad [k' < \textstyle\sum_{i=1}^{m} 2^{-2^{i+1}-i}]$$

$$\frac{2^{-2^{n+1}} - 2^{-2^{n+1}-n} + 2^{-2^n+1}}{k'} < k_1 \cdot 2^{-2^n}$$

for some k_1, and we are done, as the last equation proves the logarithmic testing complexity (provided that it is measured in terms of the number of tests in the test suite).

As in the previous example, if we rather consider that the cost is the total number of machine inputs applied by all test cases in the suite, then the resulting complexity is again in $\mathcal{O}(log(n) \cdot log(log(n)))$.

Proof of Equation 5

Now we consider $w(f) = 1$ if f is correct (it is f_0) and $w(f) = p^{err(f)}$, where $err(f)$ is defined as above, otherwise. Therefore, a function denoting an incorrect machine has weight p^i assuming that its fault appears in level i.

Under these assumptions, and taking $q = 2p$, we have

$$T_m = \sum_{i=0}^{m} q^i = 1 + \frac{q - q^{m+1}}{1-q}$$

$$Ind_{n,m} = \sum_{i=n+1}^{m} q^i = \frac{q^{n+1} - q^{m+1}}{1-q}$$

In particular (later we will see the general case), for $p = \frac{1}{4}$ (hence $q = \frac{1}{2}$), we obtain $T_m = 1 + \frac{\frac{1}{2} - \frac{1}{2^{m+1}}}{\frac{1}{2}} = 2 - 2^{-m} > 1$ for all m. On the other hand, $Ind_{n,m} = \frac{\frac{1}{2^{n+1}} - \frac{1}{2^{m+1}}}{\frac{1}{2}} = 2^{-n} - 2^{-m} < 2^{-n}$ for all m. Therefore, $D_{n,m} < 2^{-n}$ and we are done. Indeed, $D_{n,m} < \frac{1}{f^{-1}(2^n)}$ where f is just the identity function, so we conclude that the testing complexity of this problem for $p = \frac{1}{4}$ is *linear*.

In general, if we consider an arbitrary $p < \frac{1}{2}$ then we can repeat the calculations above to obtain

$$D_{n,m} = \frac{\frac{q^{n+1}-q^{m+1}}{1-q}}{1 + \frac{q-q^{m+1}}{1-q}} < \frac{q^{n+1} - q^{m+1}}{1 - q^{m+1}} < k_1 \cdot q^{n+1}$$

for some k_1, and for all m.

Let us take $p = p_r$ of the form $p_r = \frac{1}{2^{r+1}}$ for some $r > 0$, so that $q = q_r = \frac{1}{2^r}$. Then we have $D_{n,m} < k_1 \cdot \frac{1}{(2^r)^{n+1}} = \frac{1}{g^{-1}(2^n)}$ for $g(x) = k' \cdot x^{\frac{1}{r}}$ for some constant k'. In other words, we have proved that the testing problem for p_r has a testing complexity in $\mathcal{O}(n^{1/r})$ when the cost function is tsc (consequently, for tsc' we have a $\mathcal{O}(n^{1/r} \cdot log(n^{1/r}))$ complexity).

Proof of Equation 6

Now we are considering a single test case a^n. In this case, the value of T_m is exactly the same as before, because it does not depend on the testing strategy. That is, it is underestimated by

$$\begin{aligned} T_m &= \sum_{f \in C^m} w(f) \\ &\geq \sum_{i=1}^{m} \left(2^{2^{m+1}-2^{i+1}} \cdot \frac{1}{2^i} \right) = 2^{2^{m+1}} \sum_{i=1}^{m} 2^{-2^{i+1}-i} \end{aligned}$$

Hence, we only have to compute a new value for $Ind_{n,m}$. In the new testing scenario, assuming the length of the test is n, the behaviour from level $n+1$ to level m is counted in exactly the same way as before in Sect. 4.2, because tests do not detect faults at those levels. The difference is that now there are functions with errors appearing at levels 1 to n that are not detected by our test a^n, whereas the former exhaustive strategy testing suite $\{a,b\}^n$ detected all of them —at the cost of using a dramatically higher number of machine inputs for each n (or using a dramatically lower value of n, if we wish to compare both suites for the same total cost). Thus, for each level i, we have to add the weight of the functions whose first error appears in level i but that are not detected by our large test. This extra weight is represented by the last summand of the following equation:

$$\frac{\left(1-\frac{1}{2^n}\right)2^{2^{m+1}-2^{n+1}}+\sum_{i=n+1}^{m}\left(2^{2^{m+1}-2^{i+1}}\cdot\frac{1}{2^i}\right)+\sum_{i=1}^{n}\left(2^{2^{m+1}-2^{i+1}-(n-i)}\cdot\frac{1}{2^i}\right)}{2^{2^{m+1}}\sum_{i=1}^{m}2^{-2^{i+1}-i}} = \qquad [\text{factoring out } 2^{2^{m+1}}]$$

$$\frac{\left(1-\frac{1}{2^n}\right)2^{-2^{n+1}}+\sum_{i=n+1}^{m}\left(2^{-2^{i+1}}\cdot\frac{1}{2^i}\right)+\sum_{i=1}^{n}\left(2^{-2^{i+1}-(n-i)}\cdot\frac{1}{2^i}\right)}{\sum_{i=1}^{m}2^{-2^{i+1}-i}} < \qquad [\textstyle\sum_{i=n+1}^{m}2^{-2^{i+1}-i} < 2^{-2^n+1}]$$

$$\frac{2^{-2^{n+1}}-2^{-2^{n+1}-n}+2^{-2^n+1}+\sum_{i=1}^{n}2^{-2^{i+1}-n}}{\sum_{i=1}^{m}2^{-2^{i+1}-i}} < \qquad [k' < \textstyle\sum_{i=1}^{m}2^{-2^{i+1}-i}]$$

$$\frac{2^{-2^{n+1}}-2^{-2^{n+1}-n}+2^{-2^n+1}+2^{-2-n}}{k'} < k_1\cdot 2^{-2-n}$$

Figure 2. In order to prove Eq. 6, for some constant k_1, we can bind $D_{n,m}$ as shown in this figure.

$$Ind_{n,m} \leq \left(1-\frac{1}{2^n}\right)2^{2^{m+1}-2^{n+1}} + \sum_{i=n+1}^{m}\left(2^{2^{m+1}-2^{i+1}}\cdot\frac{1}{2^i}\right) + \sum_{i=1}^{n}\left(2^{2^{m+1}-2^{i+1}-(n-i)}\cdot\frac{1}{2^i}\right)$$

After repeating the calculation of $D_{n,m}$, following the same steps as we did before with the previous testing strategy in Sect. 4.2, we can find some constant k_1 to bind $D_{n,m}$ as shown in Fig. 2. Hence, if we consider that the cost of the testing strategy is the total number of machine inputs applied by all test cases in the suite (in our case, n machine inputs applied by a single test case), then the complexity of the optimal testing strategy is in $\mathcal{O}(log(n))$, which is slightly better than the complexity in $\mathcal{O}(log(n)\cdot log(log(n)))$ obtained when using the test suite $\{a,b\}^n$ in Sect. 4.2.

References

[1] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines: A survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[2] A. Petrenko, "Fault model-driven test derivation from finite state models: Annotated bibliography," in *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*. Springer, 2001, pp. 196–205.

[3] E. Brinksma and J. Tretmans, "Testing transition systems: An annotated bibliography," in *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*. Springer, 2001, pp. 187–195.

[4] I. Rodríguez, M. Merayo, and M. Núñez, "$\mathcal{HOTL}$: Hypotheses and observations testing logic," *Journal of Logic and Algebraic Programming*, vol. 74, no. 2, pp. 57–93, 2008.

[5] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, and N. Yevtushenko, "FSM-based conformance testing methods: A survey annotated with experimental evaluation," *Information & Software Technology*, vol. 52, no. 12, pp. 1286–1297, 2010.

[6] J. Tretmans, "Conformance testing with labelled transition systems: Implementation relations and test generation," *Computer Networks and ISDN Systems*, vol. 29, pp. 49–79, 1996.

[7] ——, "Testing concurrent systems: A formal approach," in *10th Int. Conf. on Concurrency Theory, CONCUR'99, LNCS 1664*. Springer, 1999, pp. 46–65.

[8] C. Gaston, P. L. Gall, N. Rapin, and A. Touil, "Symbolic execution techniques for test purpose definition," in *18th IFIP TC6/WG6.1 International Conference, TestCom 2006, LNCS 3964*. Springer, 2006, pp. 1–18.

[9] J. Springintveld, F. Vaandrager, and P. D'Argenio, "Testing timed automata," *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 225–257, 2001, previously appeared as Technical Report CTIT-97-17, University of Twente, 1997.

[10] M. Krichen and S. Tripakis, "Black-box conformance testing for real-time systems," in *11th Int. SPIN Workshop on Model Checking of Software, SPIN'04, LNCS 2989*. Springer, 2004, pp. 109–126.

[11] I. Berrada, R. Castanet, P. Félix, and A. Salah, "Test case minimization for real-time systems using timed bound traces," in *18th IFIP TC6/WG6.1 International Conference, TestCom 2006, LNCS 3964*. Springer, 2006, pp. 289–305.

[12] M. Merayo, M. Núñez, and I. Rodríguez, "Extending EFSMs to specify and test timed systems with action durations and timeouts," *IEEE Transactions on Computers*, vol. 57, no. 6, pp. 835–844, 2008.

[13] M. Stoelinga and F. Vaandrager, "A testing scenario for probabilistic automata," in *30th Int. Colloquium on Automata, Languages and Programming, ICALP'03, LNCS 2719*. Springer, 2003, pp. 464–477.

[14] N. López, M. Núñez, and I. Rodríguez, "Specification, testing and implementation relations for symbolic-probabilistic systems," *Theoretical Computer Science*, vol. 353, no. 1–3, pp. 228–248, 2006.

[15] Y. Cheon and G. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *16th European Conference on Object-oriented programming, ECOOP 2002, LNCS 2374*. Springer, 2002, pp. 231–255.

[16] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering*, vol. 11, no. 1, pp. 33–70, 2006.

[17] R. Hierons, "Comparing test sets and criteria in the presence of test hypotheses and fault domains," *ACM Trans. on Software Engineering and Methodology*, vol. 11, no. 4, pp. 427–448, 2002.

[18] ——, "Verdict functions in testing with a fault domain or test hypotheses," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 4, 2009.

[19] G. Bernot, M.-C. Gaudel, and B. Marre, "Software testing based on formal specification: a theory and a tool," *Software Engineering Journal*, vol. 6, pp. 387–405, 1991.

[20] M.-C. Gaudel, "Testing can be formal, too," in *6th CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*. Springer, 1995, pp. 82–96.

[21] J. Cherniavsky and C. Smith, "A recursion theoretic approach to program testing," *IEEE Transactions on Software Engineering*, vol. 13, pp. 777–784, 1987.

[22] J. Cherniavsky and R. Statman, "Testing: An abstract approach," in *Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis*. IEEE Computer Society Press, 1988, pp. 38–44.

[23] I. Rodríguez, L. Llana, and P. Rabanal, "A general testability theory: classes, properties, complexity, and testing reductions," *IEEE Transactions on Software Engineering*, vol. 40, pp. 862–894, 2014.

[24] I. Rodríguez, "A general testability theory," in *CONCUR 2009 - Concurrency Theory, 20th International Conference, LNCS 5710*. Springer, 2009, pp. 572–586.

[25] M. Hennessy, *Algebraic Theory of Processes*. MIT Press, 1988.

[26] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis," School of Information and Computer Science, Georgia Inst. of Technology, Tech. Rep. GIT-ICS-79/08, sept 1979.

[27] R. Gopinath, C. Jensen, and A. Groce, "Mutant census: An empirical examination of the competent programmer hypothesis," School of Electrical Engineering and Computer Science, Oregon State University, Tech. Rep., 2014.

Ismael Rodríguez is an Associate Professor in the Computer Systems and Computation Department, Complutense University of Madrid (Spain). He obtained his MS degree in Computer Science in 2001 and his PhD in the same subject in 2004. Dr. Rodríguez received the Best Thesis Award of his faculty in 2004. Dr. Rodríguez has published more than 100 papers in international refereed conferences and journals. His research interests cover formal methods, testing techniques, swarm and evolutionary optimization methods, and functional programming

Fernando Rosa-Velardo is an Associate Professor in the Computer Systems and Computation Department, Complutense University of Madrid (Spain). He obtained his MS degree in Computer Science in 2004 and his PhD in the same subject in 2007. Dr. Rosa-Velardo has published over 30 papers in international refereed conferences and journals. His research interests include formal methods, testing techniques, infinite-state systems, and Petri nets.

Fernando Rubio is an Associate Professor in the Computer Systems and Computation Department, Complutense University of Madrid (Spain). He obtained his MS degree in Computer Science in 1997, and he was awarded by the Spanish Ministry of Education with "Primer Premio Nacional Fin de Carrera". He finished his PhD in the same subject four years later. Dr. Rubio received the Best Thesis Award of his faculty in 2001. Dr. Rubio has published more than 80 papers in international refereed conferences and journals. His research interests cover formal methods, swarm and evolutionary optimization methods, parallel computing, and functional programming.