Front. Comput. Sci., 2022, 0(0): 1–31
https://doi.org/10.1007/sxxxxx-yyy-zzzz-1

RESEARCH ARTICLE

Complexity of adaptive testing in scenarios defined extensionally

Ismael RODRÍGUEZ[1,2], David RUBIO[3], Fernando RUBIO(✉)[1,2]

1 Dpto. Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain.

2 Instituto de Tecnologías del Conocimiento, Universidad Complutense de Madrid, 28040 Madrid, Spain.

3 Instituto de Biomecánica de Valencia. Universitat Politècnica de València, 46022 València, Spain.

© Higher Education Press 2022

Abstract In this paper we consider a testing setting where the set of possible definitions of the Implementation Under Test (IUT), as well as the behavior of each of these definitions in all possible interactions, are extensionally defined, i.e., on an element-by-element and case-by-case basis. Under this setting, the problem of finding the minimum testing strategy such that collected observations will necessarily let us decide whether the IUT is correct or not (i.e., whether it necessarily belongs to the set of possible correct definitions or not) is studied in four possible problem variants: with or without non-determinism; and with or without more than one possible definition in the sets of possible correct and incorrect definitions. The computational complexity of these variants is studied, and properties such as PSPACE-completeness and Log-APX-hardness are identified.

Keywords Formal testing, adaptive testing, Computational Complexity, PSPACE-completeness, approximation hardness.

1 Introduction

The research on Formal Testing Techniques (FTT) has provided lots of models to reason about testing in many possible different settings. Many aspects affecting the testing activity can be considered in formal testing models: The language the implementation under test (IUT) is assumed to be written in can be, for instance, deterministic or non-deterministic finite state machines, i.e., FSMs [1–3], input/output labelled transition systems [4–6], temporal automata [7–10], probabilistic systems [11–13], etc.; the set of divergences the IUT is assumed

Received month dd, yyyy; accepted month dd, yyyy

E-mail: fernando@sip.ucm.es

to be able to expose with respect to the desired behavior can be given by e.g., a fault model [14], a set of hypotheses about the IUT [15–17], etc.; the way we interact with the IUT or collect its responses can be e.g., via delayed interactions, imprecise observations [18, 19], etc.

A particularly interesting aspect of the latter kind is whether the test cases to be applied to the IUT must be decided before the testing activity begins, called *preset* testing, or whether the next inputs to be applied may dynamically depend on the observations (outputs) collected during the application of the previous inputs or test cases, called *adaptive* testing [1, 20–24]. In the latter case, lesser testing effort could provide more information on the (in-)correctness of the IUT, as previous observations can be used to make subsequent interactions guide the IUT into states providing more sensitive information regarding the IUT (in-)correctness.

Efforts have been made to develop strategies of adaptive testing for different frameworks. As usual, these strategies depend on the language the IUT is assumed to be defined in. For instance, the study of adaptive testing of FSMs has been particularly active [1, 20], although recently labelled transition systems have been considered as well [25], and adaptive testing strategies have also been derived from temporal logic specifications [26]. Approaches to adaptive testing also depend on the goal the tester aims at, e.g., assuring conformance testing [23, 24], constructing homing or synchronization sequences [20–22], etc. Despite these differences, the challenge of designing adaptive strategies is essentially the same in all approaches if it is seen with enough abstraction: we need to dynamically choose the next moves of the tester in such a way that the significance of the information extracted by subsequent observations is improved. In fact, typically adaptive testing can be seen as a *game* between the tester and the IUT, where the tester aims at exposing IUT faults and the IUT aims at hiding them. Certainly the IUT does not *aim* at anything, but a strategy discovering the IUT faults *in any case* requires covering also the cases where the IUT (unintentionally) hides its faults particularly well, so reasoning about these worst cases is *as if* the IUT could decide what to show, within its limited freedom, to make things harder to the tester. Could this goal, shared by all adaptive testing settings, be defined and studied in a *general* way, i.e., not being dependent on the particular IUT language, the particular form of its faults, or any other particular difficulties of the testing activity?

During the last years, some pieces of work have provided models to reason about testing from very general and abstract points of view, regardless of the language the IUT is defined in, the assumed fault model or hypotheses, or the particular conformance criterion. They include general models of *testability*, measured in terms of the cardinality required by test suites to be complete [18, 27], in particular whether they are e.g., finite, countable infinite, inexistent, etc., and they also include *testing complexity*, measured in terms of the speed our certainty on the IUT correctness increases with the number of applied test cases [19]. Despite their high generality, these models do not regard testing as a process where subsequent actions depend on previous observations, which is the essence of adaptive testing. In this paper we study the computational complexity of adaptive testing with a simple model designed to capture its essence at its most basic level. In fact, instead of providing a model with enough generality to fully express any previous model on adaptive testing, our way to grasp the essential difficulty of adaptive testing *in gen-*

eral will be quite the opposite, as we will study its most basic, simple, essential, and *less* compact form: the case where the set of possible behaviors of the IUT for all possible interactions is extensionally defined.

This *extensional definition* means we assume the actual definition of the (black box) IUT belongs to a given finite set of possible IUT definitions which are described *one by one*. Thus, this view is opposite to compactly characterizing the set of possible definitions of the IUT as all definitions fulfilling some given rule, such as e.g., "*the IUT belongs to the set of all FSMs defined like some given specification FSM but with at most one output fault*" or "*the IUT is any Java program with up to 1,000 code lines*." Moreover, our extensional definition view consistently applies to the definition of each possible IUT definition itself, as the description of all possible ways of interaction with each possible IUT definition is also given *one by one*. Hence, the set of these interactions is assumed to be finite and, consequently, the sets of possible inputs and outputs to be sent to or received from the IUT, respectively, are finite as well. All input applications to the IUT are assumed to be independent to each other, so it is assumed that the IUT reliably returns to its initial configuration after each input. Thus, inputs can be regarded as test cases, and outputs as their observed results.[1)]

Therefore, we extensionally describe what the IUT can do in all situations: we list all its possible complete behavior definitions, where each of them lists all its possible (generally non-deterministic) responses (i.e., outputs) for all possible inputs. In addition, an extensionally defined subset of the set of all possible definitions of the IUT will denote those considered *correct* according to the requirements of the tester. The goal of the tester will be finding out an adaptive testing strategy determining the (in-)correctness of the IUT —and if it exists, preferably the shortest one. Deciding whether the IUT is correct with an adaptive strategy means applying some input to the IUT such that, for any output responded by the IUT, the tester can react applying some input next –in general, different for each previous output– such that, for each output responded by the IUT, the tester can give some input afterwards, and so on in such a way that, in all cases, the tester can eventually classify the IUT as necessarily correct, i.e., inside the subset mentioned before, or necessarily incorrect (outside).

Despite the apparent excessive specificity of the previous model, it is particularly interesting for two reasons. On the one hand, note that it is much more interesting finding lower bounds of the computational complexity of the most *particular and simple* versions of problems, as proving the hardness of a problem A on some complexity class trivially implies the hardness on that class of any problem B *generalizing* problem A. Thus, the hardness of adaptive testing for any model trivially allowing us to express our extensionally defined model (e.g., the more expressive FSM model can easily be used to denote our extensional model) will be, at least, that found for our extensional model. We will prove the complexity hardness of some variants of adaptive testing for extensionally defined settings in classes such as PSPACE and Log-APX. Let us recall that input applications are assumed to be independent from each other in this model, so these hardness results are actually obtained for

[1)]This means the application of a *sequence* of consecutive machine inputs $i_1 \ldots i_n$ to the IUT must be abstracted in this model by a *single input* σ representing the whole interaction sequence $i_1 \ldots i_n$. Since finite sets of inputs and outputs are assumed, a finite set of possible *interaction sequences* $\sigma_1, \ldots, \sigma_m$ with the IUT would be considered in any case.

a *memory-less* model. This shows something that could be surprising at a first glance: the states inherent to any computation model are *not* needed at all to make adaptive testing a hard task, even a PSPACE-hard one.

On the other hand, note that there are many software engineering situations where the possible test cases the tester could apply to the IUT are extensionally defined: instead of automatically extracting test cases from a given expressive specification model and perhaps a formally defined fault model, which constitutes a kind of ideal testing methodology, testers just manually define a finite set of test cases and the possible responses of the IUT for each one, where each wrong response denotes a possible fault at the IUT. Since IUT responses for interactions not covered by these manually defined test cases will not be tested in any case, and thus will not provide any information to the tester, this model is equivalent to listing, element by element, a finite set of possible faulty definitions of the IUT, where for each one we define the finite set of possible responses for each possible test case. This pragmatic and trivial model or, roughly speaking, lack of model, which is ultimately very used by testing practitioners, turns out to be equivalent to our extensionally defined model. Indeed, for many practitioners the specification is just the set of all possible definitions that would respect all the requirements, and the fault model is just the set of all possible wrong definitions the system could have according to the mistakes they explicitly consider as feasible. In these cases the nature of both is given, by definition, by extensionally defined sets: everything in the lists is possible, and everything outside is not —and any possible analogy between test data is exactly that emerging among the regularities of these definitions.

The computational complexity of adaptive testing under our extensional model will be studied for four different problem variants, resulting from combining two binary choices. On the one hand, we will consider that non-determinism can be either allowed or forbidden. On the other hand, we will consider the case that either the subset of possible correct definitions of the IUT or the subset of incorrect ones are singletons, or the case that both sets have more than one element.[2)] The computational complexity of the four resulting problems will be identified, yielding complexity hardness in different classes: both cases requiring determinism are NP-complete, one being Log-APX-complete and the other one being hard, at least, in Log-APX; both cases dealing with non-determinism are PSPACE-complete. Note that no polynomial-time algorithm can find exact solutions for the former problems unless P = NP. Moreover, given the hardness of these problems in Log-APX, it is known that no polynomial-time approximation algorithm can guarantee any constant *performance ratio* unless P = NP, that is, the ratio between the quality of returned solutions and the quality of optimal solutions can not be guaranteed bo be bounded by a constant. Regarding the latter problems, i.e. those where non-determinism is permitted, their hardness in PSPACE means no polynomial time algorithm can solve them unless P = PSPACE, which is a stronger and even less expected property than P =

[2)] We assume the testing process stops as soon as the observations let us either discard all possible correct IUT definitions or discard all possible wrong IUT definitions — regardless of whether there are still more than one possible definition of the other kind. Note this condition is sufficient to provide a precise incorrectness/correctness diagnosis, respectively. We do not aim at *uniquely* identifying the IUT definition within the set of possibilities, which would require a stronger condition.

NP, given P ⊆ NP ⊆ PSPACE. These complexity hardness results have other practical consequences: our NP-complete problems can be approached by fast sub-optimal metaheuristics such as e.g. Genetic Algorithms, whereas other more expensive solutions such as e.g. minimax algorithms should be considered for our PSPACE-complete problems.

The rest of the paper is organized as follows. The following section introduces a motivation example informally introducing the key aspects of our setting. In Section 3, the model is formally introduced, and the complexity results are presented and proved. Conclusions and future work lines are given in Section 4. The details of the proofs of our theorems are shown in the appendix of the paper.

Besides, although the aim of this paper is theoretical, as we are identifying the computational complexity of the problems under study, some considerations about how to deal with these problems in practice despite their high complexity are given in the appendix of the paper. A much more complex example than that given in the next section is outlined there and used to introduce these practical difficulties, as well as to discuss how to deal with them.

2 Motivational example

In this section we use a simplified version of a robotic system to introduce the main notions of our testing setting. Let us test the *Collision Avoidance System* (CAS) of an autonomous four-wheeled exploration rover walking on potentially unstable and unbalanced surfaces. Three possible cases of interaction are considered in the CAS:

A.- If the collision object is expected to be reached in a time lower than one second, i.e. *near* collision, then the robot fully turns to dodge it. This operation is assumed to have some risk, but it is considered the best choice because the robot is too close to fully brake before the collision.

B.- If the collision object is expected to be reached in a time higher than two seconds, i.e. *far* collision, then the robot brakes to fully stop before the collision. In this case, the alternative choice of turning is not taken because, compared to stopping, it is considered unnecessarily risky.

C.- If the collision object is expected to be reached in a time between one and two seconds, i.e. *intermediate* collision, then the robot is allowed to either fully brake or fully turn, as both choices are considered similarly risky. However, in this case the robot is not allowed to simultaneously brake *and* turn, since fully turning while the tires are blocked due to fully braking could make the robot overturn.

The tester fears the programming team may have made some mistakes while developing the robot, so the robot will be deployed in the laboratory and its behavior for different expected collision scenarios will be observed to rule out faults. In order to test the robot, the tester puts an object at some specific distance from the robot and makes the robot move towards it. Thus, the set of possible inputs produced by the tester to stimulate the robot (test cases) are those mentioned in cases (A)-(C), i.e., $I = \{near, far, inter\}$. Besides, the set of possible reactions of the robot (outputs) are defined as $O = \{nothing, turn, brake, both\}$. According to the limited freedom offered by previous conditions (A)-(C), as well as the mistakes the tester assumes the programmers may have made, it is constructed a set of possible correct and incorrect full definitions of the IUT, i.e., definitions for all considered inputs. Each of these full definitions is formally described

$$\begin{array}{llll}
f_1(near) = \{turn\} & f_2(near) = \{turn\} & f_3(near) = \{turn\} & f_4(near) = \{turn\} \\
f_1(far) = \{brake\} & f_2(far) = \{brake\} & f_3(far) = \{brake\} & f_4(far) = \{brake\} \\
f_1(inter) = \{turn, brake\} & f_2(inter) = \{turn\} & f_3(inter) = \{brake\} & f_4(inter) = \{both\}
\end{array}$$

$$\begin{array}{llll}
f_5(near) = \{turn\} & f_6(near) = \{nothing\} & f_7(near) = \{turn\} & f_8(near) = \{brake\} \\
f_5(far) = \{nothing\} & f_6(far) = \{brake\} & f_7(far) = \{turn\} & f_8(far) = \{brake\} \\
f_5(inter) = \{turn\} & f_6(inter) = \{brake\} & f_7(inter) = \{turn\} & f_8(inter) = \{brake\}
\end{array}$$

$$\begin{array}{lll}
f_9(near) = \{nothing\} & f_{10}(near) = \{turn\} & f_{11}(near) = \{nothing\} \\
f_9(far) = \{brake\} & f_{10}(far) = \{nothing\} & f_{11}(far) = \{nothing\} \\
f_9(inter) = \{nothing\} & f_{10}(inter) = \{nothing\} & f_{11}(inter) = \{nothing\}
\end{array}$$

$$\begin{array}{ll}
f_{12}(near) = \{turn\} & f_{13}(near) = \{nothing\} \\
f_{12}(far) = \{nothing\} & f_{13}(far) = \{brake\} \\
f_{12}(inter) = \{turn, nothing\} & f_{13}(inter) = \{nothing, brake\}
\end{array}$$

Figure 1 The possible definitions of the IUT. Definitions f_1-f_3 are considered correct, whereas f_4-f_{13} are considered incorrect.

as a *function* where, for each possible input, the set of possible outputs that could be replied by the IUT after receiving that input is returned. When this set is a singleton, the behavior of that possible IUT definition for that particular input is deterministic else it is non-deterministic.

According to requirements (A)-(C), the possible IUT definitions f_1, f_2, f_3 depicted in Figure 1 are considered correct. Note that definition f_1 is allowed to non-deterministically brake or turn in intermediate-distance collisions (input *inter*), but this is allowed by requirement (C). Also note that definitions f_2 and f_3 are fully deterministic. The tester fears the implementers may have made three possible kinds of mistakes:

1.- In intermediate-distance collisions, programmers could have made the robot *simultaneously* brake and turn.

2.- Programmers may have incorrectly implemented the CAS signal requesting the robot to brake, making it do nothing in this case. Alternatively, this could have happened to the turning signal instead, or even simultaneously to both signals.

3.- Programmers could have totally ignored the case distinction required by the CAS specification, making the robot brake in all situations. Alternatively, this could have happened with the turning operation instead of braking.

The possible incorrect IUT definitions resulting from these mistakes are shown in Figure 1. Mistake (1) will be represented by the possible incorrect IUT definition f_4. Besides, mistake (2) is represented by several incorrect definitions resulting from altering correct definition f_1 (f_{12} and f_{13}), cor-

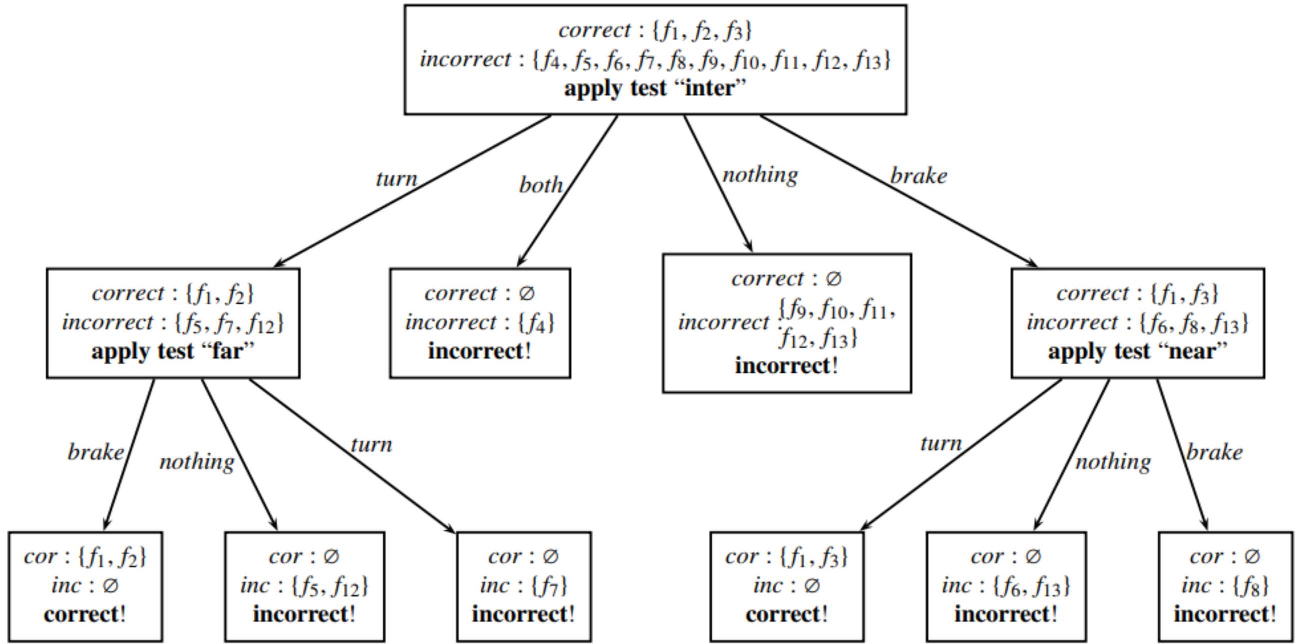


Figure 2 Adaptive testing strategy applying at most *two* test cases in all situations.

rect definition f_2 (f_5 and f_9), correct definition f_3 (f_6 and f_{10}), and any of them (f_{11}). Note that incorrect definitions f_{12} and f_{13} are non-deterministic. Finally, mistake (3) is represented by incorrect functions f_7 and f_8.

The previous tester assumptions about possible programmer mistakes are formalized by the tester hypothesis that the actual IUT definition belongs to set $\{f_1, \ldots, f_{13}\}$, consisting of those definitions fulfilling requirements (A)-(C) (i.e., f_1, f_2, f_3) and those failing them according to mistakes (1)-(3) (i.e., $f_4, \ldots, f_{13}$). The goal of the tester is interacting with the IUT, collecting observations, and using them to decide whether the IUT is necessarily correct (i.e., observations allow us to rule out all possible incorrect definitions) or incorrect (observations rule out all possible correct definitions). Assuming the tester can decide the next test cases to be applied depending on the observations collected by previous tests, i.e., adaptive testing, we tackle the following question: how many tests are needed in the worst case to decide the IUT (in-)correctness?

As shown in Figure 2, we do not need to apply all three available test cases to make that decision, because just two of them are enough in any case if the second one cleverly depends on what is observed after the first one. First, we apply test case *inter*. If the output for this test is *both*, then the IUT is necessarily incorrect, as it must be f_4. If the output is *nothing*, then the IUT is necessarily incorrect too, as it must be some definition in set $\{f_9, \ldots, f_{13}\}$. However, if the output is either *turn* or *brake*, then a second test case must be applied next. On the one hand, if it is *turn* then test case far is applied next, and all three possible subsequent replies of the IUT will let us decide whether the IUT is correct or not. On the other hand, if the output after test case *inter* is *brake*, next we apply test case *near*, and again all three possible replies give enough information to decide whether the IUT is correct or not. Note that making such decisions does not require having enough informa-

tion to uniquely identify which function defines the behavior of the IUT. For instance, in two cases depicted in Figure 2 in the first and fourth boxes at the bottom row, we provide a "correct" diagnosis and finish testing even though we do not know if the IUT is f_1 or f_2, or f_1 or f_3, respectively. Since both pairs of functions are correct and the set of possible incorrect definitions is empty in the respective boxes, distinguishing between both possible correct functions is unnecessary to decide that the IUT must be correct. Analogously, distinctions between *incorrect* definitions are also unnecessary when we have several incorrect definitions and no correct one. This is the case in the second and fifth boxes of the bottom row, as well as in the medium row's box reached after observing output *nothing*. Also note that a definition may simultaneously exist in multiple boxes after applying some test case if the definition is non-deterministic. For instance, f_1, f_{12}, and f_{13} can each be found in multiple boxes of the medium row.

It is worth noting that any alternative adaptive testing strategy starting with test case *near* would need the application of *three* test cases in the worst case, so the order in which test cases are applied in each branch of the tree shown in Figure 2 does matter a lot. After applying *near*, the IUT must be f_6, f_9, f_{11}, or f_{13} if the reply is *nothing* (i.e., incorrect IUT), it is f_8 if the reply is *brake* (incorrect IUT), and otherwise (i.e., *turn* is replied) it can be any of the remaining eight functions: f_1, f_2, f_3, f_4, f_5, f_7, f_{10}, f_{12}. Then, applying test case far would not be enough to provide an (in-)correctness diagnosis yet: observing *nothing* and *turn* would certainly be, but obtaining *brake* would not, as the remaining possible functions would be f_1, f_2, f_3, f_4, where we have three correct functions and an incorrect one. Thus, continuing our strategy with test case far does not allow us to finish testing after two test cases. Let us see that replacing far with *inter* does not work either. Recall that, before the application of the second test, we had functions f_1, f_2, f_3, f_4, f_5, f_7, f_{10}, f_{12}. After applying *inter*, output *both* would mean the IUT is f_4 (incorrect) and output *brake* would imply the IUT is either f_1 or f_3 (correct), but output *turn* would mean the IUT is f_1, f_2, f_5, f_7, or f_{12}, where we have two correct and three incorrect functions.

Symmetric arguments can be given to show that any testing strategy starting with test case far will also need the application of three test cases in the worst case. Indeed, the strategy depicted in Figure 2 is the only one providing (in-)correctness diagnoses in just two test cases in the worst case, so it is optimal.

This strategy would still let us decide whether the IUT is correct or not after introducing some function redefinitions. For instance, let us redefine the behavior of f_7 for test case far so that $f_7(far) = \{turn, nothing\}$, which makes this incorrect function non-deterministic. Then, the same adaptive testing strategy would let us decide whether the IUT is correct or not, and the diagram depicted in Figure 2 would apply to this alternative scenario as it is after adding f_7 to the set of possible incorrect definitions in the second box of the bottom row. Note that f_7 is *also* present in the incorrect set of the *third* box, and this remains unchanged. The incorrectness diagnosis given in that second box is still valid after the addition of f_7, as now we have zero correct functions and three incorrect ones. However, no adaptive or preset testing strategy would allow us decide the (in-)correctness of the IUT if f_7 was made non-deterministic in the following alternative way: $f_7(far) = \{turn, brake\}$. Note that, in this case, the incorrect function f_7

would not be distinguishable from correct functions f_1 and f_2, as f_7 could keep on replying *brake* to test case far no matter how many times it is applied to the IUT.

Restricted versions of our adaptive testing problem will also be studied in this paper, in particular by considering the following situations: assuming all functions must be deterministic, so that this would apply to our example after removing f_1, f_{12}, f_{13} and using the original definition of f_7; assuming that either the set of correct functions or the set of incorrect functions are singletons, so that this would be achieved e.g., by removing f_2 and f_3, or by removing all incorrect definitions but f_7; or by assuming both requirements together, e.g., by removing f_1, f_3, f_{12}, f_{13} and using the original definition of f_7. The complexity of the resulting four problems will be studied in the next section.

In order to introduce the difficulties of adaptive testing in bigger and more realistic scenarios, as well as to discuss how to deal with these difficulties in practice, a much much complex example is outlined in the appendix of the paper.

3 Formal model and complexity results

First, we introduce the notion of collection of definitions, which denotes the set of all possible behavioral definitions the IUT could have, each one denoted by a function. We assume that $\mathcal{P}(A)$ denotes the powerset of set A.

Definition 1. (Collection of definitions) Let I be a finite set of inputs and O be a finite set of outputs. A *collection of definitions* C for I and O is a finite enumeration of functions $f : I \longrightarrow \mathcal{P}(O)$ where for all $i \in I$ we have $f(i) \neq \emptyset$.

If $f(i) = \{o\}$, i.e., $f(i)$ is a singleton, sometimes we will just write $f(i) = o$. If $f(i)$ is a singleton for all $i \in I$, then we will say that f is *deterministic*. If all $f \in C$ are deterministic, then we say that C is *deterministic*.

We will assume that each function f is extensionally defined. That is, for each input $i \in I$ we explicitly define the value $f(i)$. □

Each function in C is the formal representation of some possible behavior of the IUT for all inputs: given a function $f \in C$, $f(i)$ represents the set of defined 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. For instance, in our motivation example, $f_1(inter) = \{turn, break\}$, so f_1 is non-deterministic, although f_2 is deterministic: $f_2(near) = \{turn\}$, $f_2(far) = \{break\}$ and $f_2(inter) = \{turn\}$. Note that $C = \{f_1, f_2, f_3, \ldots, f_{13}\}$ in this example. That is, we define the set C extensionally by enumerating all its (finite) elements.

Collections of definitions are used to represent the set of possible implementations we are considering in a given testing scenario. Implicitly, a collection of definitions C represents the *hypotheses* about the IUT the tester is assuming, see e.g., [15]. For instance, if the tester assumes the IUT is deterministic then C should only include deterministic functions; if the IUT assumes the IUT may produce wrong outputs for at most one input, then all functions in C should be defined this way; etc.

Collections of definitions 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 one after each other so that the observed results $o_1 \in f(i_1), o_2 \in f(i_2), \ldots$ allow us to provide a verdict.

Definition 2. (Specification and testing scenario) Let C be a collection of definitions. A *specification* of C is a set $E \subseteq C$. A *testing scenario* is a tuple (C, E). □

If $f \in E$ then f denotes a *correct* behavior, while $f \in C\backslash E$ denotes that f is incorrect. Thus, a specification E implicitly denotes a correctness criterion, and the set $C\backslash E$ implicitly defines the *fault model* assumed by the tester, as it consists of all incorrect behaviors the IUT may have. In any case, the correctness of a function in our model consists, by definition, in belonging to E, where E is extensionally defined function by function. In our motivation example, $E = \{f_1, f_2, f_3\}$, and this set reflects the specification implicitly defined by the requirements (A)-(C) mentioned in Section 2. Besides, $C\backslash E = \{f_4, \ldots, f_{13}\}$, and this set reflects the fault model implicitly given by the set of considered mistakes (1)-(3) in that section. Alternatively, let a specification require that a program, receiving and producing natural numbers, computes the square of the input value, and let our fault model assume that each possible wrong program P_i, with $i \in [-10, 10]$, will produce outputs being i units above the actual square. Then, $E = \{f\}$ with $f(x) = x^2$, and $C = \{g_i \mid i \in [-10, 10]\}$ where $g_i(x) = f(x) + i$.

When we deal with adaptive testing, we will not use a set $\mathcal{I} \subseteq I$ to denote a possible test suite, because now we are free to select our second input test after observing the output obtained after applying the first input, to select our third input test after obtaining the second output, and so on. Thus, we will rather speak about testing *strategies*, which are neither sets nor sequences. A testing strategy can be seen as a *tree* where each node represents the application of a test, the number of children of each node is the number of possible outputs that can be obtained for such input, and verdicts of necessary (in-)correctness are given in the leaves of the tree, as depicted in Figure 2. Thus, the sequence of nodes of each branch from the root to a leaf of the tree represents a possible sequence of tests to be applied to a IUT (similarly to a test suite in preset testing), but the concrete branch to be applied to each IUT depends on the sequence of outputs produced by the IUT during the testing process.

The aim of our problem is to decide whether it is possible or not to find a testing tree such that its depth, i.e., the number of input tests of its largest branch, is at most k, being k a given natural number. Let us start defining the decision problem in case we are only dealing with deterministic functions, that is, for all $f \in C$ and $i \in I$, $f(i)$ is a singleton.

Definition 3. (Deterministic Adaptive Test Decision Problem) Let C be a deterministic collection of definitions for I and O, $E \subseteq C$ be a specification, and k be a natural number. We say that (C, E, k) satisfies `D-ATDP` if:

$$\begin{gathered}
\exists i_1 \in I \ \forall o_1 \in \{f(i_1) : f \in C\} \\
\exists i_2 \in I \ \forall o_2 \in \{f(i_2) : f \in C,\ o_1 = f(i_1)\} \\
\vdots \\
\exists i_l \in I \ \forall o_l \in \{f(i_l) : f \in C,\ o_m = f(i_m)\ \forall m < l\} \\
\vdots \\
\exists i_k \in I \ \forall o_k \in \{f(i_k) : f \in C,\ o_m = f(i_m)\ \forall m < k\} \\
\{f \in C : o_m = f(i_m)\ \ \forall m \leq k\} \subseteq E \ \vee \\
\{f \in C : o_m = f(i_m)\ \ \forall m \leq k\} \subseteq C\backslash E
\end{gathered}$$

Given (C, E, k), the *Deterministic Adaptive Test Decision Problem* consists in finding out whether (C, E, k) satisfies `D-ATDP`. In a notation abuse, the

problem itself will also be denoted by D-ATDP. In D-ATDP instances, we will assume k is given in binary form.

When either E is a singleton or $C\backslash E$ is a singleton, that is, there is only one correct or there is only one incorrect function, the resulting particular-case problem will be denoted by D-ATDP$_1$. □

In the previous definition, satisfying D-ATDP actually consists in finding an adaptive testing strategy applying no more than k inputs.[3)] Each new level of possible input tests depends on the previous outputs produced by the IUT. Moreover, for each branch of inputs of the strategy from the root to a leaf, the functions $f \in C$ that are consistent with the outputs observed through the branch (that is, $\{f \in C : o_m = f(i_m) \ \forall m \leq k\}$) either belong all to the set of correct functions E or belong all to the set of incorrect functions $C\backslash E$. Thus, each branch of inputs classifies the corresponding IUT as either correct or incorrect. In addition, as the branches consider all possible outputs produced by the IUT, the whole tree of test cases is actually *complete* in the task of determining whether the IUT is correct or not.

Once the decision problem has been defined, it is trivial to define the corresponding optimization problem, where we are interested in finding the minimum k such that (C, E, k) satisfies D-ATDP.

Definition 4. (Deterministic Adaptive Test Optimization Problem) Let C be a deterministic collection of definitions for I and O, and $E \subseteq C$ be a specification. Given (C, E), the *Deterministic Adaptive Test Optimization Problem* (D-ATOP) consists in finding $\min\{k : (C, E, k)$ satisfies D-ATDP$\}$.

When E is a singleton or $C\backslash E$ is a singleton, we will use D-ATOP$_1$ to refer to D-ATOP. □

[3)] Although the definition requires exactly k inputs, using less inputs is clearly allowed, as filler inputs could be trivially added to satisfy the definition, e.g., repeated inputs.

Once we have defined the decision and optimization problems in the deterministic scenario, we can easily modify them to deal with the non-deterministic case. In order to do that, we only need to take into account that the output obtained by each function is not unique. Thus, we have to consider the union of all possible outputs produced by all possible functions at each of the levels of the corresponding tree, as shown in the next definition.

Definition 5. (Non-deterministic Adaptive Test Decision Problem) Let C be a collection of definitions for I and O, $E \subseteq C$ be a specification, and k be a natural number. We say that (C, E, k) satisfies N-ATDP if:

$$\begin{array}{l}
\exists i_1 \in I \ \ \forall o_1 \in \bigcup_{f \in C} f(i_1) \\
\exists i_2 \in I \ \ \forall o_2 \in \bigcup_{f \in \{f \in C:\, o_1 \in f(i_1)\}} f(i_2) \\
\quad \vdots \\
\exists i_l \in I \ \ \forall o_l \in \bigcup_{f \in \{f \in C:\, o_m \in f(i_m)\, \forall m < l\}} f(i_l) \\
\quad \vdots \\
\exists i_k \in I \ \ \forall o_k \in \bigcup_{f \in \{f \in C:\, o_m \in f(i_m)\, \forall m < k\}} f(i_k) \\
\{f \in C : o_m \in f(i_m) \ \ \forall m \leq k\} \subseteq E \ \vee \\
\{f \in C : o_m \in f(i_m) \ \ \forall m \leq k\} \subseteq C\backslash E
\end{array}$$

Given (C, E, k), the *Non-deterministic Adaptive Test Decision Problem* consists in finding out whether (C, E, k) satisfies N-ATDP. The problem itself will also be denoted by N-ATDP. In N-ATDP instances, k will be defined in binary form.

When either E or $C\backslash E$ is a singleton, the resulting problem will be denoted by N-ATDP$_1$. □

Again, it is trivial to define the corresponding optimization problem, as shown in the next definition.

Definition 6. (Non-Deterministic Adaptive Test Optimization Problem) Let C be a collection of

definitions for I and O, and $E \subseteq C$ be a specification. Given (C, E), the *Non-deterministic Adaptive Test Optimization Problem* (N-ATOP) consists in finding $\min\{k : (C, E, k)$ satisfies N-ATDP$\}$.

When E is a singleton or $C \backslash E$ is a singleton, we will use N-ATOP$_1$ to refer to N-ATOP. □

Given the obvious equal treatment of E and $C \backslash E$ in the definitions of the satisfaction conditions of D-ATDP and N-ATDP (see definitions 3 and 5), hereafter only the case where E is a singleton will be regarded when dealing with all problems D-ATDP$_1$, D-ATOP$_1$, N-ATDP$_1$, and N-ATOP$_1$.

Once we have defined our problems, we present the following properties for the deterministic scenario. The inclusion of D-ATDP in class NP might look strange at a first glance, given the apparent exponential size of D-ATDP solutions, as they are trees, and, moreover, the apparent exponential *depth* of these trees with respect to the size representation of D-ATDP instances, as k is defined in *binary* in them.

Theorem 1. We have:

(a) D-ATDP is NP-complete.

(b) D-ATDP$_1$ is NP-complete.

(c) D-ATOP is Log-APX-hard.

(d) D-ATOP$_1$ is Log-APX-complete.

□

Proof. Let us start proving (c). In order to do it, we have to provide a Log-APX-hardness preserving polynomial reduction from a Log-APX-hard problem into D-ATOP. We consider an S-reduction[4)] from *Minimum Set Cover*, MSC [29]. Given a collection C of sets $C = \{S_1, \ldots, S_m\}$ with $\bigcup_{S \in C} S = \{e_1, \ldots, e_p\}$, MSC consists in picking a subset C' of C such that $\bigcup_{S \in C'} S = \{e_1, \ldots, e_p\}$ in such a way that $|C'|$ (i.e., the number of sets in C') is minimized. The construction of an S-reduction from MSC into D-ATOP can be done as follows. From an MSC instance C defined as before, we define a D-ATOP instance in the same terms as in Definition 4 where we will have a different input for each set, a single correct function (which always returns output 0), and an incorrect function for each possible element of the subsets:

- $I = \{1, 2, \ldots, m\}$
- $O = \{0, 1\}$
- $E = \{g\}$, where $\forall i \in I \; g(i) = 0$
- $C = E \cup \{f_e : e \in \{e_1, \ldots, e_p\}\}$ where

$$f_e(l) = \begin{cases} 1 & \texttt{if } e \in S_l \\ 0 & \texttt{otherwise} \end{cases}$$

Let us remark that each input test represents a set of the original MSC problem. Moreover, there is an incorrect function for each possible element of the original sets, and the application of this function to an input (that is, to a *set*) returns 1 iff the element belongs to the corresponding set.

We can see that there exists a solution to this D-ATOP instance with cost $q \in \mathbb{N}$ (say solution $\{S_{l_1}, \ldots, S_{l_q}\}$) iff there exists a solution to the original MSC instance with the same cost q, and we can easily map the former into the latter.

Taking into account our reduction, $\bigcup_{j=1}^{q} S_{l_j}$ is a set cover of C iff $\forall e \in \bigcup C \; \exists r \in \{1 \ldots q\}$ such that $e \in S_{l_r}$. Thus, $f_e(l_r) = 1$. Hence, by using input l_r in D-ATOP we *distinguish* function f_e from the only correct function g, which always returns 0, i.e., the observed output will always let us discard one of them. Therefore, the set of inputs $\{l_j : j \in \{1, \ldots, q\}\}$ allows to distinguish g from all

4) In an S-reduction, the solutions of the second problem can be mapped into solutions of the first problem having exactly the same cost, and the optima of both problems have the same cost as well. See e.g., [28].

functions in $C\backslash\{g\}$. Hence, by applying these q inputs in any order, observed outputs will always let us know either the IUT is correct or incorrect.

Analogously, if some set of q inputs $\{l_j : j \in \{1,\ldots,q\}\}$ allows to distinguish g from all incorrect functions, then for each incorrect function f_e we have an input l_r such that $f_e(l_r) = 1$, because the correct function always returns 0. Thus, in the corresponding MSC problem we will also have that for each element $e \in \bigcup C$ we have a set S_{l_r} such that $e \in S_{l_r}$. Thus, $\{S_{l_j} : j \in \{1,\ldots,q\}\}$ is a set cover of C with cost q.

We conclude that there is an S-reduction from MSC into D-ATOP, and this reduction proves that D-ATOP is Log-APX-hard. Moreover, note that the mapping proposed in our S-reduction also polynomially reduces MSC to D-ATDP, which proves that D-ATDP is NP-hard. In order to conclude (a), we also need to show D-ATDP is NP.

Let us recall our view of testing strategies as trees, as illustrated in Figure 2. Since all functions are deterministic in D-ATDP (contrarily to the case shown in Figure 2, actually dealing with N-ATDP), the sets of all (correct and incorrect) functions being possible at the children nodes of a given node of a testing strategy must be pairwise disjoint. By a trivial inductive reasoning, this implies that the sets of functions being possible at all leaves of the strategy must be pairwise disjoint as well, and in fact they must constitute a partition of the collection of definitions C. Let $n = |C|$ denote the number of functions in C and $m = |I|$ denote the number of inputs. We infer that the number of leaves of a testing strategy for D-ATDP cannot be higher than n. Let us consider a testing strategy where all applied inputs are different from all other inputs appearing in the *same* branch, i.e., no branch of the strategy contains repeated inputs. Clearly, the depth of this strategy cannot be higher than m. Note that deciding D-ATDP does not require exploring any testing strategy where some branch contains repeated inputs, because repeated inputs are useless. This is obvious given the deterministic nature of all functions in D-ATDP. Hence, regardless of the value of k given in a D-ATDP instance (C, E, k), we can always decide D-ATDP by considering trees with no more than n leaves and no more than m depth. We infer that the total number of nodes of any candidate tree we might need to consider to decide D-ATDP cannot be higher than $n \cdot m$. Moreover, as the representation size of C in a D-ATDP instance cannot be lower than n and cannot be lower than m, we infer that the sizes of candidate solutions of D-ATDP, i.e., testing strategies, are polynomial with the sizes of the corresponding problem instances. Given this limitation, it is easy to see that checking the validity of any candidate solution takes polynomial time, so D-ATDP is NP.

Let us note that, in our previous S-reduction, we produced D-ATOP instances which are also, in fact, D-ATOP$_1$ instances, and the cost functions to be minimized in both optimization problems coincide. Thus, that S-reduction also proves that D-ATOP$_1$ is Log-APX-hard and D-ATDP$_1$ is NP-hard.

As we have proven D-ATOP$_1$ is Log-APX-hard, in order to prove (d) we only need to prove D-ATOP$_1$ is Log-APX. We can do it by providing an S-reduction from D-ATOP$_1$ into MSC. As we know MSC is Log-APX, this will imply that D-ATOP$_1$ is also in Log-APX.

The construction of an S-reduction from D-ATOP$_1$ into MSC can be done as follows. From a D-ATOP$_1$ instance $(C, \{g\})$, we define an MSC instance where we create a set S_i for each input $i \in I$. The elements of the sets will be the incorrect functions of the original D-ATOP$_1$ problem. Moreover, each set S_i will contain those functions that produce an

incorrect output for input i. That is:

$$S_i = \{f : f \in C \backslash \{g\}, f(i) \neq g(i)\} \ \ \forall i \in I$$

Then, a set $\{S_{l_i} : i \in \{1, \ldots, q\}\}$ is a set cover iff each function $f \in C\backslash\{g\}$ is included in at least a set S_{l_i}. This property holds iff for each function $f \in C\backslash\{g\}$ there exists at least one input l_i such that $f(l_i) \neq g(l_i)$, meaning that applying all inputs in the set $\{l_i : i \in \{1, \ldots, q\}\}$ in any order is a valid testing strategy for the original testing problem.

Let us remark that the cost of the solution is the same in both problems, q, also when the solution of MSC is optimal. So, we have an S-reduction from D-ATOP$_1$ to MSC. Thus, D-ATOP$_1$ is Log-APX. As we have already proved D-ATOP$_1$ is Log-APX-hard, we conclude that D-ATOP$_1$ is Log-APX-complete.

Moreover, by taking this S-reduction as a polynomial reduction, we also prove D-ATDP$_1$ is NP, as MSC is NP. Since we proved earlier that D-ATDP$_1$ is NP-hard, we conclude D-ATDP$_1$ is NP-complete, as stated in (b). □

Once we have analyzed the deterministic scenario, next we deal with the non-deterministic case.

Theorem 2. We have that N-ATDP and N-ATDP$_1$ are PSPACE-complete. □

Proof Sketch: The complete proof can be seen in the appendix, here we describe the main ideas. First, we prove that N-ATDP is PSPACE and, as a consequence, we will also have that its particular case N-ATDP$_1$ is also in PSPACE. Then, we prove that N-ATDP$_1$ is PSPACE-hard and, as a consequence, we will also have that its generalization N-ATDP is also PSPACE-hard.

N-ATDP is PSPACE

Let us show an algorithm that solves N-ATDP by using a polynomial amount of memory with respect to the size of the input. We assume that there is a total order for the elements of I and also a total order for the elements of O, and that we can compare any pair of elements using a polynomial amount of memory with respect to the size of the sets I and O. This can be trivially done, as we can explicitly define all the possible pairs by using $|I|^2$ and $|O|^2$ memory.

Our algorithm will use two vectors i and o, as well as a natural variable l. Vector i (respectively, o) will contain the input (resp. output) variables we are currently considering at the candidate testing strategy branch under study, while the natural variable l will contain the depth we are located at in the decision tree. Assuming the testing strategy is allowed to apply up to k inputs in our N-ATDP instance, apparently k should be the size of vectors i and o, as well as the maximum possible value of l. However, note that the amount represented by k is exponential with respect to the length of the binary code representing it in the N-ATDP instance. Thus, the size of vectors i and o need to be smaller than k to guarantee a polynomial use of memory. Their actual size will be explained later.

The algorithm, whose pseudocode is given in Appendix 4, tries all possible combinations of inputs and outputs in a minimax fashion. In case a combination does not allow to either discard all correct functions or discard all incorrect functions, then this combination is not useful. Thus, we modify the last input whose possible values have not been completely tested yet, that is, we try the next input possibility. In case there are not more input possibilities, it means that the answer to the decision problem is not.

In case a given combination allows to discard all correct functions or all incorrect functions, then we change the last output whose possible values have not been completely tested, that is, we go on

to check whether the new output does not modify the feasibility of this strategy. In case there are not more output possibilities, it means that we have been able to discard all correct or all incorrect functions in all the possible situations. That is, the answer to the decision problem is yes.

Despite the exponential size of the tree being explored, during the execution of the algorithm the program only needs to remember where it is along the tree exploration, and vectors i and o and natural number l define that place. Note that, regardless of the actual value of k given in the N-ATDP instance (C, E, k), no testing strategy constructed for deciding N-ATDP needs to have a depth higher than h, where $h = |I|$ is the number of available inputs, because repeating inputs in the same branch is useless. Notice that the non-determinism of functions in N-ATDP allows the IUT to reply the same output to the same input no matter how many times it is applied. Hence, if some node of the testing strategy applies a repeated input, i.e., an input already applied in some previous node of the same branch, then, for some child of this node, the set of non-discarded functions will be exactly the same as in its parent node. Therefore, the problem of discarding either all correct or all incorrect functions available at the former node will be the same as that problem for the latter node —although after applying one additional input, which makes this repeated input useless. We conclude that the depth of the exploration performed by the algorithm will never need to be higher than h, so the actual size of vectors i and o, as well as the maximum possible value of l, can be defined as $min(k, h)$. Since the number of available inputs h cannot be higher than the representation size of C in any N-ATDP instance (C, E, k), the depth of the tree being explored is polynomial, and vectors i and o can be represented with polynomial memory.

In addition, each specific action of the algorithm requires a constant number of auxiliary variables. Note that we only need to obtain the minimum or maximum of a given set, to obtain the next element of a set, to check whether $o[m] \in f(i[m])$ for all $1 \le m \le min(k, h)$, and to check whether $f \in E$ or $f \in C \backslash E$. We conclude that the algorithm runs with polynomial space.

$\texttt{N-ATDP}_1$ is PSPACE-hard

In this case we have to provide a polynomial reduction from a PSPACE-hard problem into our problem $\texttt{N-ATDP}_1$. In particular, we will reduce the *True Qualified Boolean Formula* problem (TQBF [30, 31]) into $\texttt{N-ATDP}_1$. In this problem, given a propositional logic formula and a quantification of all its variables by universal and existential quantifiers, we decide if the resulting first-order logic formula is true, i.e., equivalent to $\top$. There are several possible presentations of this problem, and in particular we assume that the propositional formula is given in *conjunctive normal form* (CNF) and that existential and universal quantifiers alternate. That is, formulas will always have the following form:

$$\exists x_1 \forall y_1 \exists x_2 \forall y_2 \dots \exists x_k \forall y_k \ \phi$$

where ϕ is a propositional logic formula defined over variables $x_1, y_1, \dots, x_k, y_k$ which is given in CNF, i.e., $\phi = c_1 \wedge \dots \wedge c_n$ for some disjunctive clauses $c_1, \dots, c_n$.

Our polynomial reduction from TQBF into $\texttt{N-ATDP}_1$ will be easier to introduce if we view both problems as games: in TQBF a "player" tries to iteratively set the existential variables in such a way that the formula will be true no matter how the corresponding universal variables are set by an "opponent," and in $\texttt{N-ATDP}_1$ the tester iteratively selects the inputs to be applied to the IUT in such

a way that either all correct or all incorrect functions are eventually discarded no matter how the corresponding outputs are chosen by the "opponent IUT." Our reduction will map TQBF existential variables into N-ATDP$_1$ inputs (chosen by the tester), and TQBF universal variables into the corresponding N-ATDP$_1$ outputs produced afterwards (chosen by the IUT). Hence, when the tester introduces some input implicitly setting the value of existential variable x_j, the IUT will reply the output implicitly setting the value of universal variable y_j. As we will see, formula ϕ will be true under some variable values iff either all correct or all incorrect functions are discarded under the corresponding inputs and outputs. In particular, each disjunctive clause c_i of ϕ in TQBF will be represented by an *incorrect* function f_{c_i} in N-ATDP$_1$ (actually, satisfying c_i will be equivalent to discarding f_{c_i}), and the N-ATDP$_1$ instance will contain a single correct function g. Since the tester will immediately *win* if the only correct function is discarded, the hard and most interesting part of classifying the IUT as correct or incorrect *in all cases* will be when IUT outputs are always consistent with that single correct function.

More technically, from a TQBF instance of the form $\exists x_1 \forall y_1 \ldots \exists x_k \forall y_k \phi$, we define a N-ATDP$_1$ instance where we will have: two different inputs for each variable existentially quantified, one representing that x_j is set to $\top$ and another one representing it is set to $\bot$; a single correct function g, which can return 0 or 1 for all inputs; three possible outputs $0, 1, -1$ representing the two possible values of variables y_j and a new extra value (-1) to distinguish any function producing it from the correct function, which cannot produce it; and an incorrect function (f_{c_i}) for each disjunctive clause (c_i) of the logic formula ϕ. Formally,

- The allowed depth of the testing strategy, i.e., third value of our N-ATDP$_1$ instance, is set to k.
- $I = \{x_1, x_2, \ldots, x_k\} \cup \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k}\}$,
- $O = \{0, 1, -1\}$,
- $E = \{g\}$, where $\forall i \in I \; g(i) = \{0, 1\}$,
- $C = E \cup \{f_{c_i} : c_i \in \phi\}$ where

$$f_{c_i}(x_j) = \begin{cases} -1 & \texttt{if}\; x_j \in c_i \\ -1 & \texttt{if}\; y_j, \overline{y_j} \in c_i \\ 0 & \texttt{if}\; y_j \in c_i \;\wedge\; x_j, \overline{y_j} \notin c_i \\ 1 & \texttt{if}\; \overline{y_j} \in c_i \;\wedge\; x_j, y_j \notin c_i \\ \{0,1\} & \texttt{otherwise} \end{cases}$$

$$f_{c_i}(\overline{x_j}) = \begin{cases} -1 & \texttt{if}\; \overline{x_j} \in c_i \\ -1 & \texttt{if}\; y_j, \overline{y_j} \in c_i \\ 0 & \texttt{if}\; y_j \in c_j \;\wedge\; \overline{x_j}, \overline{y_j} \notin c_i \\ 1 & \texttt{if}\; \overline{y_j} \in c_i \;\wedge\; \overline{x_j}, y_j \notin c_i \\ \{0,1\} & \texttt{otherwise} \end{cases}$$

Each function f_{c_i} will be discarded with some inputs and outputs in the N-ATDP$_1$ instance iff clause c_i is satisfied in the original TQBF instance with the corresponding variable values. If literal x_j (analogous for $\overline{x_j}$) appears in disjunctive clause c_i, then c_i is obviously satisfied when x_j holds. Then, input x_j (respectively, $\overline{x_j}$) distinguishes f_{c_i} from the correct function g, because f_{c_i} returns -1 to that input and g returns either 0 or 1. Note that if the IUT returns -1 then the tester immediately *wins* by discarding the only correct function g, and if it returns 0 or 1 then the tester will discard f_{c_i}. In addition, if both y_j and $\overline{y_j}$ belong to c_i then f_{c_i} will return -1 for both inputs x_j and $\overline{x_j}$, as c_i will be trivially satisfied in both cases. Otherwise, if literal y_j (analogous for $\overline{y_j}$) appears in clause c_i, then f_{c_i} is defined to return the *opposite* value to that fulfilling that literal. For instance, if literal $\overline{y_j}$ appears in clause c_i and the IUT replies 0 to x_j or $\overline{x_j}$, i.e., the IUT is setting y_j

to $\perp$, then f_{c_i} is *discarded*, i.e., clause c_i is satisfied, because f_{c_i} is defined to return 1 in that case.

The previous construction is not correct yet as it, because we must still solve two problems. On the one hand, although only k inputs are allowed in the constructed $\mathtt{N\text{-}ATDP}_1$ instance, the tester could apply k inputs and still represent an impossible logical valuation. For instance, for $k = 3$, the tester could apply x_1, $\overline{x_1}$, and x_3, thus giving both possible values to x_1 and leaving x_2 undefined. On the other hand, we must guarantee that the order used to select the inputs (i.e., tests) is the same as the order in which variables appear in the corresponding TQBF formula. Note that inputs (i.e., tests) can be applied in any order in $\mathtt{N\text{-}ATDP}_1$, but variables are introduced in a mandatory order in TQBF, and this order does matter. For instance, it is easy to see that $\exists x_1 \forall y_1 \exists x_2 \forall y_2 ((x_2 \wedge y_1) \vee (\overline{x_2} \wedge \overline{y_1}))$ holds but $\exists x_2 \forall y_2 \exists x_1 \forall y_1 ((x_2 \wedge y_1) \vee (\overline{x_2} \wedge \overline{y_1}))$ does not.

We will add new incorrect functions to be discarded to our $\mathtt{N\text{-}ATDP}_1$ instance, and we will make sure that it is impossible to discard all of them *in all cases* (more precisely, in all cases where the tester does not trivially win by discarding the only correct function g) *unless* exactly one valuation is given by the tester to each existential variable (the first problem mentioned above) and all of them are applied in the order required by the TQBF instance (the second problem). An additional incorrect function f_0, as well as new incorrect functions f_j and f'_j for all $1 \leq j \leq k-1$, will be added to set C. By their design, discarding all of them will force the tester to apply at least one input representing each existential variable, it does not matter if positively or negatively. Given the limit of k inputs to be applied in our $\mathtt{N\text{-}ATDP}_1$ instance, this will solve the first of our previous problems.

In order to solve the second problem, i.e. the order, the new incorrect functions will be defined as follows. For any input representing a value for existential variable x_j (again, it does not matter if positive or negative), f_j and f'_j will reply 0 and 1, respectively. Given this definition, the IUT will discard f_j if it replies 1 and f'_j if it replies 0. Alternatively, in any branch where the IUT replies -1, g will be discarded and the tester will trivially win, so next we focus on all branches where this does not happen. Note that the choice between discarding f_j or discarding f'_j will be out of the control of the tester, as this will be decided by the IUT. We will give the tester the power to control which one of either f_j or f'_j is discarded when the input corresponding to variable x_{j+1} is applied. In order to allow this, we duplicate all the inputs representing existential variables into *prime* and *non-prime* versions, so that for each j we will have versions x_j, $\overline{x_j}$, x'_j, and $\overline{x_j}'$. When a version of the input corresponding to x_{j+1} is applied to the IUT, function f'_j will be necessarily discarded if a non-prime version is used, i.e., x_{j+1} or $\overline{x_{j+1}}$, as f'_j will return -1 to any of them, and function f_j will be discarded if a prime version is used (i.e., x'_{j+1} or $\overline{x_{j+1}}'$), for which f_j will return -1. The choice of the version of x_{j+1} to be applied will let the tester control which one of f_j or f'_j is discarded but, again, any version of x_{j+1} will discard either f_{j+1} or f'_{j+1} *without* any tester control. Therefore, the tester will be forced to discard the non-discarded function with a suitable version of x_{j+2}, and so on.

We can see that discarding *all* functions f_0, f_1, f'_1, ..., f_{k-1}, f'_{k-1} will force the tester to apply the chosen versions of variables $x_1, \ldots, x_k$ in that *exact* order, while f_0, the anchor function of this process, will be discarded by any version of x_1. If some version of variable x_{j+1} is applied to the IUT *before* the

corresponding version of x_j is applied, then when we apply the former we will not know which one of either f_j or f'_j will not be discarded *afterwards* by the latter, so the decision of which function needs to be *rescued* by x_{j+1} will be blind. Hence, there will exist a branch in the testing strategy where the application of x_j discards, between f_j and f'_j, the function which was *already* discarded before, letting the other one non-discarded and making the tester lose.

By selecting some input x_j, $\overline{x_j}$, x'_j, or $\overline{x_j}'$ for each j, the tester will simultaneously make a decision in the negation vs non-negation axis to discard all f_{c_i} functions, and in the prime vs non-prime axis to discard all f_j and f'_j functions. Note that the tester will *not* win if there exists a branch in the testing strategy where g and either some f_{c_i} or some f_l or f'_l is not discarded. Thus, satisfying all clauses *and* keeping the variables order with a consistent valuation of variables are both mandatory conditions to win in the $\texttt{N-ATDP}_1$ instance.

Given the previous considerations, the $\texttt{N-ATDP}_1$ instance its now defined as follows. The correctness of the resulting polynomial reduction is proved in Appendix B.

- $I = \{x_1, x_2, \ldots, x_k\} \cup \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k}\} \cup \{x'_1, x'_2, \ldots, x'_k\} \cup \{\overline{x_1}', \overline{x_2}', \ldots, \overline{x_k}'\}$,
- k, O, and E are not modified. Again, $g(i) = \{0, 1\}$ for all inputs $i \in I$, including the new ones,
- $C = E \cup \{f_{c_i} : c_i \in \phi\} \cup \{f_0\} \cup \{f_l, f'_l : \forall l \in \{1, \ldots, k-1\}\}$ where f_{c_i} is defined as before for all x_j or $\overline{x_j}$, and the values it returns for the new inputs x'_j (analogously $\overline{x_j}'$) are the same as those returned by x_j (analogously, $\overline{x_j}$), that is,

$$f_{c_i}(x'_j) = f_{c_i}(x_j) \qquad f_{c_i}(\overline{x_j}') = f_{c_i}(\overline{x_j})$$

Regarding the new functions f_l, we use $2k-1$ of them: two for each existentially quantified variable x_j (f_j and f'_j) but the first one, for which we add only one function (f_0). These functions are defined in the same way for each variable and its negation, that is:

$$f_l(x_j) = f_l(\overline{x_j}) \qquad f_l(x'_j) = f_l(\overline{x_j}')$$
$$f'_l(x_j) = f'_l(\overline{x_j}) \qquad f'_l(x'_j) = f'_l(\overline{x_j}')$$

The concrete definitions of these functions are as follows for $1 \le l < k$:

$$\begin{array}{ll} f_l(x_l) = \{0\} & f_l(x'_l) = \{0\} \\ f'_l(x_l) = \{1\} & f'_l(x'_l) = \{1\} \\ f_l(x_{l+1}) = \{0,1\} & f_l(x'_{l+1}) = \{-1\} \\ f'_l(x_{l+1}) = \{-1\} & f'_l(x'_{l+1}) = \{0,1\} \\ f_l(x_j) = f'_l(x'_j) = \{0,1\} & \forall j \notin \{l, l+1\} \end{array}$$

Finally, we have an additional function f_0 to deal with the anchor case:

$$f_0(x_1) = f_0(\overline{x_1}) = f_0(x'_1) = f_0(\overline{x_1}') = \{-1\}$$
$$f_0(x_j) = f_0(\overline{x_j}) = f_0(x'_j) = f_0(\overline{x_j}') = \{0,1\}$$
$$\text{for } 1 < j \le k$$

Clearly, this transformation can be done in polynomial time with respect to the size of the $\texttt{TQBF}$ instance: being n the number of clauses in ϕ, there are $2k + n$ functions in the $\texttt{N-ATDP}_1$ instance, $4k$ inputs, and 3 outputs. We prove that the transformation actually reduces $\texttt{TQBF}$ into $\texttt{N-ATDP}_1$ in the appendix of the paper.

4 Conclusions and future work

In this paper we have studied the computational complexity of adaptive testing in a setting where

the set of possible definitions of the IUT and the behavior of each of them for each possible interaction are extensionally defined case by case. Problem variants where non-determinism is allowed or forbidden, and either the set of possible correct IUT definitions or the set of incorrect ones must be a singleton or not, have been studied. Due to the propagation of complexity hardness by generalization, our complexity hardness results apply not only to our simple model, but also to any other more expressive adaptive testing models generalizing ours. In particular, the PSPACE-completeness of our two non-deterministic problem variants shows that states are not needed at all to make adaptive testing a PSPACE-hard problem, as a fully memory-less setting is assumed in them.

Regarding practical application, let us remark that in many practical scenarios it is very common to list the errors that could occur. Although this list can be relatively long when all of them are listed one by one, it is also very common that for practical purposes it can be described in a reasonably compact way. A good example of this situation is the one that can be seen in the example in Appendix A, where the repertoire of possible errors of the implementation under test is relatively short for a tester to express. However, this specification is easily translated into an enormously long list of possible implementations. Thus, although the tester does not need to name them one by one, it is easy to obtain such a detailed enumeration.

Our main line of future work, already ongoing work, consists in heuristically solving random and designed instances of our adaptive testing problems by applying heuristic, non-optimal methods. We are developing a minimax implementation to handle the PSPACE-complete variants, i.e., the non-deterministic ones, and an implementation based on Genetic Algorithms to handle the deterministic ones. Preliminary observations suggest that these methods could provide reasonable sub-optimal testing strategies in reasonable execution times.

Acknowledgements This work has been partially supported by project PID2019-108528RB-C22, and by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union. The authors would like to thank the anonymous reviewers for valuable suggestions on a previous version of this paper.

Appendixes

Appendix A. Second motivational example: how to deal with larger scenarios

In the example shown in Section 2, the number of possible IUT definitions (functions) to be considered was quite reduced, so the set of them was easy to enumerate manually. In more complex and realistic testing scenarios, a big number of possible IUT definitions could emerge by combining, in any way deemed possible by the tester, all considered possible correct choices and all considered possible mistakes. Note that, given the lack of mandatory specification model —e.g. FSMs, Timed Automata, C++ programs, etc.— in our extensional setting, the tester has full freedom to consider in each scenario which mistakes can happen simultaneously and which ones cannot. Let us point out that the existence of a huge set of possible definitions of the IUT does not imply a lengthy definition *work* by the tester. In practice, those huge sets should usually be the result of explosively combining a much more manageable number of factors describing, in a compact fashion, all possible allowed and forbidden behaviors. In these cases, a simple program can be written to enumerate all the resulting functions, i.e. possible IUT definitions, by just exhaustively combining all factors in all ways considered possible. This way, the tester is released

from the enumeration task. Next we present an example involving a big number of functions.

Let us assume that we test an already assembled automated teller machine (ATM). Note that, in this case, the definition of each test does not only consist in a way to interact with the ATM's keyboard, because the initial configuration of the ATM also affects the outcome of the test. This configuration involves the availability of notes of each type in the ATM and the state, in the bank database, of all user accounts connecting to ATM during the test. Thus, the application of each test case of the ATM involves putting some exact amount of notes of each kind (10€, 20€, 50€, and 100€) into the cashier's banknote deposit, changing the database to create fake users with the desired balance amounts, interacting with the ATM via its keyboard, letting the money out, and resetting the system again. The whole process may take several minutes, so it is not feasible to apply a huge number of test cases. Hence, our testing strategy must be designed so as to provide as much information about the IUT (in-)correctness as possible —with a limited total interaction with it.

In this setting, a full test case, i.e. *input* in our setting, is defined in terms of the following data:

- A Number of bills of each amount in the ATM before starting.
- B For each user ID involved in the withdrawals to be made during the test case, money in the account of that user before starting the test.
- C For each consecutive money withdrawal made during the same test, we have the following information: ID of the user making the withdrawal; amount of money being requested; and confirmation "yes / no" that will be answered in case the system requests it.

Information (A) can be denoted by a 4-tuple, using one element for each note type, (B) by a set of pairs (ID and money in account), and (C) by an ordered list of tuples. Hence, a test case can be denoted by a triple containing all three. For instance,

$$
\begin{aligned}
(\quad & (24, 13, 14, 23), \\
& \{(ID_1, 100), (ID_2, 140)\}, \\
& [(ID_1, 80, yes), (ID_2, 200, no), (ID_1, 40, yes)] \ )
\end{aligned}
$$

denotes a test where we initially provide the ATM with 24 10€ bills, 13 20€ bills, 14 50€ bills, and 23 100€ bills, next we create fake user accounts ID_1 and ID_2 with 100€ and 140€, respectively, and finally we interact with the ATM making three consecutive withdrawal attempts involving these two users: ID_1 tries to withdraw 80€, next ID_2 tries to take 200€, and finally ID_1 comes back to try to withdraw 40€.

Regarding the output of each test, it includes the following information for each withdrawal being requested: money given to the user, in particular numbers of notes given of each kind; information on the screen of the resulting balance of that user, which could be different to the actual balance being registered internally, though the latter is unknown to the user; and whether a confirmation "yes / no" is requested to the user or not. Moreover, we assume we can detect every time the ATM tries to give a bill that must exist in its notes deposit according to its internal data but in fact does not, so these errors are also included in the information associated with each withdrawal. Consequently, the output of each test will be represented by a list, where each element denotes all the previous output information for a withdrawal.

Regarding the specification, i.e. the set of possible IUT definitions being considered correct, some freedom is given to the implementer on several issues. We consider the following options:

- (2 options) Return every 100€ to be delivered using a single 100€ bill; or two bills of 50€ each. If one of these types of bill runs out, then the rest of the money is provided by using the other type of bill.
- (2 options) Return the remainder of dividing the amount to be given by 100 by using the maximum possible number of 20€ bills available at the ATM and then complete with 10€ bills; or with the maximum possible number of 10€ bills and then complete with 20€ bills if we run out of 10€ bills.
- (2 options) If there is not enough physical money deposited in the ATM, then the implementation can give everything that is left in the ATM; or not give anything.
- (3 options) If the client does not have enough money in his/her account, then the implementation can give him/her exactly what is left in the account; give nothing; or give on credit. In the latter case, a request for confirmation will be required.
- (2 options) Request confirmation in all cases concerned in the previous two items where some money could be given; or not.
- (2 options) Before giving the requested money or displaying that the operation cannot be done, show an ad of the bank's investment funds; or show an ad of bank's loans.

For each of the previous six choices, each correct implementation will be allowed to deterministically follow one of the available options. Moreover, in the last choice, it will also be allowed to follow an additional option: to non-deterministically choose any of both options in each case. Thus, this choice unleashes 3 possibilities instead of 2. By combining the resulting options in any possible way, we have $2*2*2*3*2*3 = 144$ possible correct functions, i.e. correct possible IUT definitions, in our specification set.

Regarding the set of possible incorrect functions, we consider the following possible errors:

a In the second and all subsequent money withdrawals during the same test, assume the data of the first user ID regardless of whether the test tries to use different users.

b Not updating the amount of bills that are available in the ATM after each withdrawal.

c Not updating the amount of money available on the user's account after each withdrawal.

d (2 options) Give one more or one less bill of the biggest type of bill which should be delivered.

e (2 options) Give the part of the money amount being multiple of 100 in alternative bills of 100€ and 50€ until availability, ignoring the two allowed criteria; or totally ignoring that part of the money, i.e. not giving it.

f (2 options) Give the remaining part of the money amount, i.e. the remains of 100, in alternative bills of 20€ and 10€ until availability, ignoring the two allowed criteria; or totally ignore that part of the money, i.e. not giving it.

g (6 options) Believe that bills of any type $x \in \{10, 20, 50, 100\}$ are interchangeable with bills of type $y \in \{10, 20, 50, 100\}$ with $x \neq y$, both for delivering them and for accounting the number of each available in the ATM. Since this creates an equality relation between bills of type x and those of type y with $x \neq y$, and the order between x and y is irrelevant, 6 options arise.

h Giving all requested money regardless of the amount of money of the user in the bank.

i Never showing any ad.

j (2 options) In confirmation requests, internally swap yes and no answers; or consider that all answers are the same as that given in the first withdrawal.

This gives us $1+1+1+2+2+2+6+1+1+1+2 = 19$ types of possible errors. For each of the 144 correct specifications mentioned above, we consider multiple incorrect versions that include one or more of the above errors. The tester assumes the probability that the development team made more than 5 mistakes is negligible, so there are $\binom{19}{5} + \binom{19}{4} + \binom{19}{3} + \binom{19}{2} + \binom{19}{1} + 1 = 11,628 + 3,876 + 969 + 171 + 19 + 1 = 16,664$ possible ways to introducing any number of mistakes from 0 to 5 in any of these 144 possible correct IUT definitions. Thus, there are $144 * 16,664 = 2,399,616$ possible definitions of the IUT, of which 144 are correct.

Since the execution of each test case will take several minutes, we are interested in minimizing the number of tests that we need to apply, and we can reduce that number if we use adaptive testing instead of preset testing. Note that, even if the first tests do not exhibit any behavior which is faulty by its own, i.e. before checking the consistency of their outputs with *other* test results, they can show us which other tests should be applied next to chase potential faults more efficiently. For instance,

- if some test shows that the part of the money being multiple of 100 is given with 100€ notes, then checking whether 50€ notes are dealt with correctly requires putting an insufficient amount of 100€ notes in subsequent tests, as otherwise they will not be used by the ATM. We have the opposite situation if that part of the money is given with 50€ notes;
- we have a similar situation for the remainder of 100 and 20€–10€ notes;
- if some test detects that the ATM chooses to deliver *all* money available in the ATM when the user tries to withdraw more money than available in the ATM (instead of rejecting the withdrawal), then checking whether the accounting of notes in the ATM works as expected requires less interactions: in some tests, the ATM will tell us exactly how many notes it *thinks* it has;
- if some test detects that confirmations are not requested, then there is no need to execute tests aimed at detecting faults in the confirmation process.

This illustrates that the order in which test cases are applied generally affects the number of test cases we need to apply to collect useful information about the IUT correctness/incorrectness: if the first test cases focus on deciding which other test cases would be useful afterwards, depending on the outputs collected by these first tests, then reaching correctness/incorrectness diagnoses will require less interactions in general. Hence, the order of test cases in a testing strategy, and in particular in each of its branches, is crucial.

As we mentioned before, a simple program can enumerate all possible correct and incorrect definitions of the IUT by combining all correct and incorrect alternatives in all ways allowed by the tester. In general, this method gives the tester full flexibility to reflect any knowledge she has on the way she knows or suspects the IUT was developed. For instance, it could be the case that faults X and Y are not expected to happen together but X and Z are; that fault X is not expected if the developers choose legitimate choice W; etc.

In the resulting enumeration, each possible IUT definition will be a function associating inputs, i.e. test cases, and outputs. Actually, the behavior of

functions needs to be defined only for some finite set of test cases considered significant and representative by the tester, out of which a suitable testing strategy will be designed. This finite set of test cases, having in general more elements than the amount of test cases we can afford to apply to the IUT, will be extracted from the set of all possible test cases, and each test case in the set will have the possibility to be included or not in the testing strategy.

In general, each of these *a priori* significant test cases will compose values from several finite, discrete, and/or continuous domains of choices depending on the nature of the IUT. Examples of these possibilities include, respectively, simple choices such as 'yes'/'no', amounts of euros to be withdrawn such as 70€ or 450€, or time lapses between consecutive keystrokes such as 3.47 seconds or 0.1242 seconds.[5] Note that, for choices involving continuous or infinite discrete domains, some representative elements have to be selected, out of infinite possibilities, to be included in the test cases under consideration. Both extreme and standard values should be taken, where the former consist in very small and very big values, and the latter can be extracted by picking several values with the same expected distribution as the one the IUT is expected to face in real life usage. For instance, if some real value is expected to be distributed according to an exponential variable with $\lambda = 3$, then several samples of that variable could be used in the composition of the test cases under consideration, as well as some very small and very big values for the sake of degenerate behavior observation.

Notice that we do not need to store a *program* to represent each function: a simple vector of legitimate and faulty choices can denote each function, and a single generic program receiving these choices will tell us the behavior of the corresponding function denoted by that vector.

Outputs do not denote continuous values in our example because no system requirement depends on any continuous domain in this case. Alternatively, let us consider some scenario where they do, so we need to deal with outputs such as e.g. o = "*signal Ok is shown after 3.448 seconds*." In this case, some error margin should be allowed when matching observed IUT outputs during testing. Thus, output o should be redefined as e.g. "*signal Ok is shown after 3.448 ± 0.05 seconds*". Let us note that these intervals could overlap and produce some non-determinism at the *observation* level of the testing process. For example, let $f \in E$ and $f' \in C\backslash E$ with $f(a) = \{1.4\}$ and $f'(a) = \{1.5\}$. That is, according to the possible correct definition of the IUT denoted by f, the numeric output 1.4 is deterministically answered to input a, and according to possible incorrect definition f', output 1.5 is deterministically replied to a. Let the observation precision margin of the tester be ± 0.2. Let us suppose the actual definition of the IUT is f. Note that, upon the reception of input a, the 1.4 reply of the IUT will be seen by the tester as any value within $[1.2, 1.6]$. On the other hand, if the actual IUT definition is f', then the 1.5 answer will be seen as any value within $[1.3, 1.7]$. Note that the sub-interval $[1.3, 1.6]$ is common to both functions, whereas any value within intervals $[1.2, 1.3)$ and $(1.6, 1.7]$ would uniquely identify the function producing that answer as either f or f', respectively. In order to integrate this observability limitation,

[5] These time lapses are not considered in test cases in our example because no functionality depends on them. If a functionality such as e.g. logging out when a timeout is reached were added, then explicitly references to time lapses between consecutive interactions in the ATM would be needed in test cases.

the behavior of f and f' for input a is redefined as follows: $f(a) = \{o_1, o_2\}$ and $f'(a) = \{o_2, o_3\}$, where these three new outputs o_1, o_2, o_3 abstractly represent the disjoint numeric intervals [1.2, 1.3), [1.3, 1.6], and (1.6, 1.7], respectively.

Thus, although the original definitions of f and f' produce deterministic responses 1.4 and 1.5 to input a, respectively, after the redefinition their response is non-deterministic to reflect our limited capability to distinguish some observations from actual outputs.

Although the goal of this paper is just showing the computational complexity of, given the previous data, finding testing strategies, next we will briefly outline how these problems could be heuristically faced in practice. Testing strategies could be dynamically constructed similarly as strategies for other PSPACE-complete games are computationally found in run time as long as the game develops. After the result of each test case is collected, and thus some possible correct and incorrect IUT definitions are discarded, a minimax algorithm should be applied up to some limited depth —due to the exponential size of the search space— and the test case returned by the algorithm as the best first move to be made should be applied next to the IUT. Most interesting branches could be explored more deeply, and some heuristic should be applied in general at nodes at that maximum depth. For instance, the ratio between possible correct definitions and possible incorrect definitions could be used as heuristic in these nodes.

Actually, in complex testing scenarios, the realistic goal should not be guaranteeing that, in any situation, *all* possible correct or *all* possible incorrect IUT definitions will be discarded. On the contrary, it should be reaching configurations where the ratio between possible correct definitions and possible incorrect definitions is extreme in either way —that is, even though the minority side is not fully discarded, its relative probability is very low. Thus, the actual goal would not be reaching the exhaustiveness and the precise correctness/incorrectness diagnoses it allows to give, but just reaching some kind of pseudo-exhaustiveness.

Given the non-negligible time cost of executing a minimax algorithm to decide each new test case to be applied to the IUT (depending on the depth it aims to reach), it is essential to adjust that depth in such a way its execution takes a relatively short time —in particular, in comparison with the time needed to *apply* each test case to the IUT. Otherwise, the testing process would be halted for a non-negligible time, waiting for the decision of which test case should be applied next, and a significant amount of time would be wasted. Note that, in this case, rather than executing the algorithm and waiting for it, that time could actually be used to apply a big amount of *any* test cases: even if these test cases are chosen without any algorithm-driven criterion, e.g. randomly, applying *many more* test cases could be better than applying many fewer thoroughly-designed test cases. Let us point out that the execution of the algorithm and the application of additional tests to the IUT can be trivially done *in parallel*, so the best of both choices can be trivially combined: we can apply both those clever test cases selected by the algorithm *and also* other randomly designed test cases, in such a way that the testing process *never* has to halt until the next (good) test case to be applied is decided by the minimax algorithm. Moreover, note that the execution of the minimax algorithm could give us not only the best test case to be applied next, but also other interesting test cases to be applied next –other moves reaching nodes with good maximin

values–, so at least part of these other test cases to be applied could not be that random after all.

This combination of best test cases (best first moves) and other tests is unnecessary when the application of each test case is expected to take several minutes, as the example developed before. In this case, the execution of the minimax algorithm up to a decent depth would take a significantly shorter time than the application of any test case to the IUT, so there would be no "idle testing time" to be filled with those test cases commented above: *all* tests applied to the IUT would be exactly the best moves discovered by the minimax algorithm.

Appendix B. Proof of Theorem 2

B.1 Algorithm solving `N-ATDP` in polynomial space

The algorithm solving `N-ATDP` in polynomial space we informally described earlier is the following:

$l = 1$; $h = |I|$; $k = min(k, h)$

[Step 0]

$i[l] = \min(I)$

[Step 1]

$o[l] = \min(\bigcup_{f \in \{f \in C:\, o[m] \in f(i[m])\ \forall m < l\}} f(i[l]))$

[Step 2]

- if $l < k$:
  - $l = l + 1$; go to [Step 0].
- if $l = k$:
  - if $\{f \in C : o[m] \in f(i[m])\ \forall m \leq k\}$ is a subset of either $C \backslash E$ or E, that is, in case there does not exist $f \in E$ such that $o[m] \in f(i[m])$ for all $1 \leq m \leq k$ or there does not exist $f \in C \backslash E$ such that $o[m] \in f(i[m])$ for all $1 \leq m \leq k$:

    while ($l > 0$ and

    $o[l] = \max \bigcup_{f \in \{f \in C:\, o[m] \in f(i[m])\ \forall m < l\}} f(i[l])$)

    $l = l - 1$

    if $l = 0$: **return YES**.

    else $o[l]$ is assigned to the next output and we go to [Step 2].
  - if $\{f \in C : o[m] \in f(i[m])\ \forall m \leq k\}$ is neither a subset of $C \backslash E$ nor a subset of E:

    while ($l > 0$ and

    $i[l]$ is the maximum input $a \in I$ with

    $a \neq i[m]\ \forall\ 1 \leq m \leq l - 1$)

    $l = l - 1$.

    if $l = 0$: **return NO**.

    else $i[l]$ is assigned to the next input $a \in I$ with $a \neq i[m]\ \forall\ 1 \leq m \leq l-1$ and we go to [Step 1].

Obviously, in order to solve the optimization problem rather than the decision problem, we could iteratively try $k = 1$, then $k = 2$, etc., until we find the first value returning YES.

The efficiency of the previous algorithm could be improved. For instance, the algorithm checks whether all functions belong to E or all belong to $C \backslash E$ only at level $min(k, h)$, but this could also hold earlier, and extending the current branch after it holds is unnecessary. Still, the algorithm runs within polynomial memory as it is. In particular, as we noted earlier, i, o, and l define the exploration point through the tree, and they can be defined with polynomial memory.

B.2 $\texttt{N-ATDP}_1 \in \texttt{PSPACE-hard}$

Given the reduction presented in the proof sketch of Theorem 2, we have to prove that any instance of `TQBF` is satisfied iff the corresponding instance of $\texttt{N-ATDP}_1$ is satisfied.

- B.2.1 $\texttt{TQBF} \Rightarrow \texttt{N-ATDP}_1$

We start proving that in case the `TQBF` formula is satisfied, then the corresponding $\texttt{N-ATDP}_1$ instance is also satisfied.

First, we prove that either g or all the f_l (and f_l') functions will be discarded if the inputs corresponding to the existential variables are appropriately selected in the correct order, in particular as follows. The first input can be any value $i_1 \in \{x_1, \overline{x_1}, x_1', \overline{x_1}'\}$. Then, for the rest of inputs, we have to appropriately select them from the set $\{x_l, \overline{x_l}\}$ or from the set $\{x_l', \overline{x_l}'\}$ to discard f_l or f_l', depending on the last output that was obtained. Formally, for each $1 \le l < k$:

- if $o_l = -1$ then the correct function g is discarded and we know for sure that the IUT is incorrect. Thus, no more inputs are needed at this strategy branch;
- if $o_l = 0$ (which does not discard f_l) then we use $i_{l+1} \in \{x_{l+1}', \overline{x_{l+1}}'\}$ (which does);
- if $o_l = 1$ (which does no discard f_l') then we use $i_{l+1} \in \{x_{l+1}, \overline{x_{l+1}}\}$ (which does).

Thus, in case any output is -1, then all the functions that are consistent with the inputs and outputs are incorrect, because the only correct function g never returns -1, i.e., g is discarded. Otherwise, all the outputs belong to the set $\{0, 1\}$, but all the functions f_l and f_l' are discarded. Note that for all $1 \le l < k$ we have two cases for o_l:

$$o_l = 0 \Rightarrow \begin{cases} o_l \notin f_l'(i_l) = \{1\} \\ i_{l+1} \in \{x_{l+1}', \overline{x_{l+1}}'\} \Rightarrow \\ \qquad 0 \le o_{l+1} \notin f_l(i_{l+1}) = \{-1\} \end{cases}$$

$$o_l = 1 \Rightarrow \begin{cases} o_l \notin f_l(i_l) = \{0\} \\ i_{l+1} \in \{x_{l+1}, \overline{x_{l+1}}\} \Rightarrow \\ \qquad 0 \le o_{l+1} \notin f_l'(i_{l+1}) = \{-1\} \end{cases}$$

Hence, in any case we will discard both f_l and f_l' for at least one input, which can be either i_l or i_{l+1}. Finally, we have to discard also function f_0. However, taking into account that $i_1 \in \{x_1, \overline{x_1}, x_1', \overline{x_1}'\}$ then we know that $0 \le o_1 \notin f_0(i_1) = \{-1\}$, which discards f_0.

Thus, we conclude that for all $1 \le l < k$ we have $f_l, f_l', f_0 \notin \{f \in C : o_m \in f(i_m), m \le k\}$.

Let us now prove that in case the qualified boolean formula is satisfied then either some output -1 is observed –so g is discarded and the IUT is incorrect–, or all functions f_{c_i} will also be discarded. In case variable x_1 is set to true in the valuation strategy applied to the TQBF instance, then we use $i_1 \in \{x_1, x_1'\}$. Otherwise, we use $i_1 \in \{\overline{x_1}, \overline{x_1}'\}$. Then, for the rest of inputs, that is, for all $1 \le l < k$ we have two options: if $o_l < 0$ then no more inputs are needed, because the correct function g is discarded by o_l, as g never returns -1; otherwise we take $y_l = o_l$ and have two cases again for the next input:

$$\begin{cases} i_{l+1} \in \{x_{l+1}, x_{l+1}'\} & \text{if } x_{l+1} \text{ is true in the TQBF valuation} \\ i_{l+1} \in \{\overline{x_{l+1}}, \overline{x_{l+1}}'\} & \text{if } x_{l+1} \text{ is false in the TQBF valuation} \end{cases}$$

That is, either all outputs are positive, or an output is negative and then we discard the correct function, so that all the functions that are consistent with the inputs and outputs are incorrect. Let us see that, in the latter case, all functions of the form f_{c_i} are discarded.

Let us suppose that no observed output is negative. Then, for all disjunctive clauses c_i of the logic formula ϕ and for all $1 \le j \le k$,

- if x_j is true then $i_j \in \{x_j, x_j'\}$. If $x_j \in c_i$ then $0 \le o_j \notin f_{c_i}(i_j) = \{-1\}$.
- if x_j is false then $i_j \in \{\overline{x_j}, \overline{x_j}'\}$. If $\overline{x_j} \in c_i$ then $0 \le o_j \notin f_{c_i}(i_j) = \{-1\}$.
- if $y_j, \overline{y_j} \in c_i$ then $i_j \in \{x_j, \overline{x_j}, x_j', \overline{x_j}'\}$. Thus, $0 \le o_j \notin f_{c_i}(i_j) = \{-1\}$.

- if $y_j \in c_i$ but $\overline{y_j} \notin c_i$ and y_j is true, then $f_{c_i}(i_j)$ is either $\{0\}$ or $\{-1\}$ for $i_j \in \{x_j, \overline{x_j}, x_j', \overline{x_j}'\}$ and $1 = y_j = o_j \notin f_{c_i}(i_j) \subset \{-1, 0\}$.
- if $\overline{y_j} \in c_i$ but $y_j \notin c_i$ and y_j is false, then $f_{c_i}(i_j)$ is either $\{1\}$ or $\{-1\}$ for $i_j \in \{x_j, \overline{x_j}, x_j', \overline{x_j}'\}$ and $0 = y_j = o_j \notin f_{c_i}(i_l) \subset \{-1, 1\}$.

All in all, if c_i is satisfied, i.e., it has a literal whose value is true, then $f_{c_i} \notin \{f \in C : o_m \in f(i_m),\ m \leq k\}$. Hence, if ϕ is satisfied then all its clauses are satisfied and none of the f_{c_i} belongs to $\{f \in C : o_m \in f(i_m),\ m \leq k\}$.

Let us remark that the election of inputs taken to discard functions f_{c_i} is compatible with the election of inputs used to discard functions f_l, f_l', and f_0. In the second case, we choose between the inputs with and without primes, while in the first case we choose between the positive and negative inputs. By considering both factors together, a single input will be available in each situation for all inputs but the first one, where prime and non-prime versions are available. Therefore, by combining both strategies, in case we introduce the variables in the proper order and ϕ is satisfied, then we will discard either g or all the other functions, i.e., those with forms f_{c_i}, f_l, f_0, and then we solve the $\mathtt{N\text{-}ATDP}_1$ problem.

- B.2.2 $\mathtt{N\text{-}ATDP}_1 \Rightarrow \mathtt{TQBF}$

Now we prove that the qualified boolean formula holds in case the corresponding instance of $\mathtt{N\text{-}ATDP}_1$ is satisfied.

First, we prove that if (C, E, k) satisfies $\mathtt{N\text{-}ATDP}_1$ then, at each branch of the strategy, either there is some output -1 (and g is discarded), or for all $1 \leq j \leq k$ we have at least one input from the set $\{x_j, \overline{x_j}, x_j', \overline{x_j}'\}$. Thus, as we can only use k inputs at any branch, we have exactly one possible valuation for each x_j in the latter kind of branches, i.e., those without -1.

Let us suppose (C, E, k) satisfies $\mathtt{N\text{-}ATDP}_1$ and let us assume the inputs and outputs at the same branch are $\{i_m\}_{m=1}^k$ and $\{o_m\}_{m=1}^k$. In case there exists $o_m = -1$ then we have the property. Thus, we have to look for contradictions in the other case. That is, let us now consider that the branch is such that, for all $1 \leq m \leq k$, $o_m \in \{0, 1\}$. Then, $g \in \{f \in C : o_m \in f(i_m), i \leq k\}$. Moreover, as (C, E, k) satisfies $\mathtt{N\text{-}ATDP}_1$ then we have that $E = \{g\} = \{f \in C : o_m \in f(i_m), i \leq k\}$, because otherwise the tests would not allow to distinguish incorrect functions from the correct one as required to satisfy $\mathtt{N\text{-}ATDP}_1$. Let us prove that this property cannot hold if $\exists j : x_j, \overline{x_j}, x_j', \overline{x_j}' \notin \{i_m\}_{m=1}^k$:

- If $j = 1$ then f_0 could not be distinguished from g with any other input, as f_0 is equal to g for all inputs but those corresponding to $j = 1$. Thus, $f_0 \in \{f \in C : o_m \in f(i_m), i \leq k\}$, and this is a contradiction, because the incorrect function f_0 would not be distinguished from the correct function g.
- If $j > 1$ and $\nexists i_M \in \{x_{j-1}, \overline{x_{j-1}}, x_{j-1}', \overline{x_{j-1}}'\}$ then we have two consecutive variables which are not covered (those with subscripts j and $j-1$), but functions f_l only differ from g for variables l and $l+1$. That is, we will have that for all m: $f_{j-1}(i_m) = f_{j-1}'(i_m) = g(i_m) = \{0, 1\}$. Thus $f_{j-1}, f_{j-1}' \in \{f \in C : o_m \in f(i_m), i \leq k\}$ and that is again a contradiction.
- If $j > 1$ and $\exists! i_M \in \{x_{j-1}, \overline{x_{j-1}}, x_{j-1}', \overline{x_{j-1}}'\}$ then o_M can be either 0 or 1. In the first case, $f_{j-1} \in \{f \in C : o_m \in f(i_m), i \leq k\}$, because o_M does not discard f_{j-1} and this function can only be discarded with inputs of the set $\{x_{j-1}, \overline{x_{j-1}}, x_{j-1}', \overline{x_{j-1}}', x_j, \overline{x_j}, x_j', \overline{x_j}'\}$, and we have not more inputs of that type. Analogously, in case o_M is 1 we have the same situation with f_{j-1}'. In any case, we have a

contradiction, as either f_{j-1} or f'_{j-1} is not distinguished from g.

- If $j > 1$ and we have that $\exists i_{M_1}, i_{M_2}, \ldots, i_{M_p} \in \{x_{j-1}, \overline{x_{j-1}}, x'_{j-1}, \overline{x_{j-1}}'\}$ then we are applying several inputs corresponding to variable x_{j-1}. However, for f_{j-1} (respectively f'_{j-1}), all of them will return the value 0 (respectively 1), so we will be in the same situation as in the previous item. That is, either f_{j-1} or f'_{j-1} will not be discarded.

Let us now prove that inputs are necessarily introduced in the same order as in the definition of the TQBF instance. That is, we have that if (C, E, k) satisfies $\texttt{N-ATDP}_1$ then $\forall j : i_j \in \{x_j, \overline{x_j}, x'_j, \overline{x_j}'\}$.

Let us assume that (C, E, k) satisfies $\texttt{N-ATDP}_1$ and let us consider a strategy branch with set of inputs $\{i_m\}_{m=1}^k$ and the corresponding outputs $\{o_m\}_{m=1}^k$. Since (C, E, k) satisfies $\texttt{N-ATDP}_1$, the functions of C which are consistent with these inputs and outputs are either all correct or all incorrect.

In case there exists an output whose value is -1, then we obviously have $\{f \in C : o_m \in f(i_m), i \leq k\} \subseteq C \backslash E$, because $E = \{g\}$ and g only returns either 0 or 1. In the other case (that is, $o_m \in \{0, 1\}$ for all $1 \leq m \leq k$), we have $g \in \{f \in C : o_m \in f(i_m), i \leq k\}$. Thus, as (C, E, k) satisfies $\texttt{N-ATDP}_1$, we must have $E = \{g\} = \{f \in C : o_m \in f(i_m), i \leq k\}$. That is, we discard the rest of functions (all incorrect ones). Let us see that there is a contradiction if the inputs are introduced out of order.

Consider the minimum value j such that $i_j \notin \{x_j, \overline{x_j}, x'_j, \overline{x_j}'\}$. Then, we have that $i_j \in \{x_n, \overline{x_n}, x'_n, \overline{x_n}'\}$ for some $n > j$.

Let us denote by i_l the only input that belongs to $\{x_{n-1}, \overline{x_{n-1}}, x'_{n-1}, \overline{x_{n-1}}'\}$ in $\{i_m\}_{m=1}^k$. Then we have $l > j$. If $i_j \in \{x_n, \overline{x_n}\}$ (respectively $i_j \in \{x'_n, \overline{x_n}'\}$), in case $o_j \in \{0, 1\}$ we will have f_{n-1} (respectively f'_{n-1}) $\in \{f \in C : o_m \in f(i_m), i \leq \boldsymbol{j}\}$, and in the branch of the strategy where $o_l = 0$ (respectively $o_l = 1$) we will have f_{n-1} (resp. f'_{n-1}) $\in \{f \in C : o_m \in f(i_m), i \leq \boldsymbol{k}\}$, as the only way to distinguish these functions from function g is by using variables with subscripts n or $n - 1$. This makes the contradiction.

Now that we know that there is a single input for each variable and that the inputs are necessarily in the same order as the variables, that is, every input satisfies $i_j \in \{x_j, \overline{x_j}, x'_j, \overline{x_j}'\}$, we prove that, whenever (C, E, k) satisfies $\texttt{N-ATDP}_1$, the corresponding formula $\exists x_1 \forall y_1 \ldots \exists x_k \forall y_k\ \phi$ is true, i.e., it satisfies TQBF. Let us show that, by following a TQBF strategy derived from the $\texttt{N-ATDP}_1$ strategy used for instance (C, E, k), every clause $c_i \in \phi$ is satisfied. In order to satisfy ϕ, in each TQBF strategy branch we set x_j to true if $i_j \in \{x_j, x'_j\}$ in the corresponding $\texttt{N-ATDP}_1$ strategy branch, whereas we set it to false if $i_j \in \{\overline{x_j}, \overline{x_j}'\}$.

Let us remind that the values of each i_j (for $j > 1$) depend on the previous output o_{j-1}. In our case, we use $o_{j-1} = y_{j-1}$. Note that $\texttt{N-ATDP}_1$ strategy branches with some -1 output trivially discard the correct function and make the testing strategy win in branches *not* existing in the TQBF strategy counterpart, where variables can only be true or false. As the TQBF strategy cannot include branches like this, next we focus on the other branches. For all branches of the $\texttt{N-ATDP}_1$ strategy where no o_m is -1, we know that for all $1 \leq m \leq k$ we have $o_m \in \{0, 1\}$. Hence, $g \in \{f \in C : o_m \in f(i_m), i \leq k\}$. Therefore, as we know (C, E, k) satisfies $\texttt{N-ATDP}_1$, we have that $E = \{g\} = \{f \in C : o_m \in f(i_m), i \leq k\}$ in those branches.

Under the previous conditions, let us show that if a function f_{c_i} is discarded, that is, we have that $f_{c_i} \notin \{f \in C : o_m \in f(i_m), i \leq k\}$, then the corresponding clause c_i is satisfied. In order to discard f_{c_i}, there must exist an input i_M such that $o_M \notin f_{c_i}(i_M)$. Ob-

viously, if $f_{c_i}(i_M) = \{-1\}$ then the function is discarded, as we are considering branches where outputs -1 are not observed. By construction, function f_{c_i} can be defined like this for input i_M in three situations:

- $x_M \in c_i$ and $i_M \in \{x_M, x'_M\}$, so x_M is true.
- $\overline{x_M} \in c_i$ and $i_M \in \{\overline{x_M}, \overline{x_M}'\}$, so x_M is false.
- $y_M, \overline{y_M} \in c_i$.

Obviously, in the three cases we have that clause c_i is satisfied.

The rest of cases where f_{c_i} is discarded are the following:

- if $y_M \in c_i$ but $x_M, \overline{y_M} \notin c_i$ then $f_{c_i}(i_M) = \{0\}$ for $i_M \in \{x_M, x'_M\}$. Thus, if f_{c_i} is discarded then $o_M = 1$.
- if $y_M \in c_i$ but $\overline{x_M}, \overline{y_M} \notin c_i$ then $f_{c_i}(i_M) = \{0\}$ for $i_M \in \{\overline{x_M}, \overline{x_M}'\}$. Thus, if f_{c_i} is discarded then $o_M = 1$.
- if $\overline{y_M} \in c_i$ but $x_M, y_M \notin c_i$ then $f_{c_i}(i_M) = \{1\}$ for $i_M \in \{x_M, x'_M\}$. Thus, if f_{c_i} is discarded then $o_M = 0$.
- if $\overline{y_M} \in c_i$ but $\overline{x_M}, y_M \notin c_i$ then $f_{c_i}(i_M) = \{1\}$ for $i_M \in \{\overline{x_M}, \overline{x_M}'\}$. Thus, if f_{c_i} is discarded then $o_M = 0$.

As it can be seen, if f_{c_i} is discarded then either $y_M \in c_i$ and $o_M = y_M = 1$ or $\overline{y_M} \in c_i$ and $o_M = y_M = 0$. In both cases, c_i is satisfied.

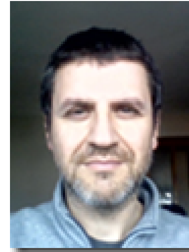
In any other case, f_{c_i} is not discarded, because in any other case $f_{c_i}(i_M) = \{0, 1\}$.

As we have seen that every c_i is satisfied, we also have that ϕ is satisfied.

References

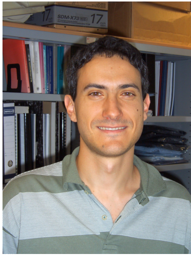
1. Lee D, Yannakakis M. Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE, 1996, 84(8): 1090–1123
2. Petrenko A. Fault model-driven test derivation from finite state models: Annotated bibliography. In: 4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067. 2001, 196–205
3. Dorofeeva R, El-Fakih K, Maag S, Cavalli A, Yevtushenko N. FSM-based conformance testing methods: A survey annotated with experimental evaluation. Information & Software Technology, 2010, 52(12): 1286–1297
4. Tretmans J. Conformance testing with labelled transition systems: Implementation relations and test generation. Computer Networks and ISDN Systems, 1996, 29: 49–79
5. Tretmans J. Testing concurrent systems: A formal approach. In: 10th Int. Conf. on Concurrency Theory, CONCUR'99, LNCS 1664. 1999, 46–65
6. Brinksma E, Tretmans J. Testing transition systems: An annotated bibliography. In: 4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067. 2001, 187–195
7. Springintveld J, Vaandrager F, D'Argenio P. Testing timed automata. Theoretical Computer Science, 2001, 254(1-2): 225–257. Previously appeared as Technical Report CTIT-97-17, University of Twente, 1997
8. Krichen M, Tripakis S. Black-box conformance testing for real-time systems. In: 11th Int. SPIN Workshop on Model Checking of Software, SPIN'04, LNCS 2989. 2004, 109–126
9. Berrada I, Castanet R, Félix P, Salah A. Test case minimization for real-time systems using timed bound traces. In: 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, LNCS 3964. 2006, 289–305
10. Merayo M, Núñez M, Rodríguez I. Extending EFSMs to specify and test timed systems with action durations and timeouts. IEEE Transactions on Computers, 2008, 57(6): 835–844
11. Stoelinga M, Vaandrager F. A testing scenario for probabilistic automata. In: 30th Int. Colloquium on Automata, Languages and Programming, ICALP'03, LNCS 2719. 2003, 464–477
12. López N, Núñez M, Rodríguez I. Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science, 2006, 353(1–3): 228–248
13. Efatmaneshnik M, Shoval S, Joiner K. System test architecture evaluation: A probabilistic modeling approach. IEEE Systems Journal, 2019, 1–12
14. Morell L J. A theory of fault-based testing. IEEE Transactions on Software Engineering, 1990, 16(8): 844–857
15. Hierons R. Comparing test sets and criteria in the presence of test hypotheses and fault domains. ACM Trans. on Software Engineering and Methodology, 2002, 11(4): 427–448
16. Hierons R. Verdict functions in testing with a fault domain or test hypotheses. ACM Transactions on Soft-

ware Engineering and Methodology, 2009, 18(4)
17. Rodríguez I, Merayo M, Núñez M. $\mathcal{HOTL}$: Hypotheses and observations testing logic. Journal of Logic and Algebraic Programming, 2008, 74(2): 57–93
18. Rodríguez I, Llana L, Rabanal P. A general testability theory: classes, properties, complexity, and testing reductions. IEEE Transactions on Software Engineering, 2014, 40: 862–894
19. Rodríguez I, Rosa F, Rubio F. Introducing complexity to formal testing. Journal of Logical and Algebraic Methods in Programming, 2020, 111
20. Kushik N, Yevtushenko N. Adaptive homing is in P. In: Tenth Workshop on Model-Based Testing (MBT 2015), EPTCS 180. 04 2015, 73–78
21. Yenigün H, Yevtushenko N, Kushik N. The complexity of checking the existence and derivation of adaptive synchronizing experiments for deterministic FSMs. Information Processing Letters, 2017, 127: 49–53
22. Kushik N, Yevtushenko N, Yenigün H. Reducing the complexity of checking the existence and derivation of adaptive synchronizing experiments for nondeterministic FSMs. In: International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2016. 2016, 83–90
23. Petrenko A, Yevtushenko N. Adaptive testing of deterministic implementations specified by nondeterministic FSMs. In: Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011. 2011, 162–178
24. Petrenko A, Yevtushenko N. Adaptive testing of nondeterministic systems with FSM. In: 15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014. 2014, 224–228
25. Bos v. d P, Vaandrager F W. State identification for labeled transition systems with inputs and outputs. CoRR, 2019, abs/1907.11034
26. Bloem R, Fey G, Greif F, Könighofer R, Pill I, Riener H, Röck F. Synthesizing adaptive test strategies from temporal logic specifications. Formal Methods in System Design, 2019, 55: 103–135
27. Rodríguez I. A general testability theory. In: CONCUR 2009 - Concurrency Theory, 20th International Conference, LNCS 5710. 2009, 572–586
28. Crescenzi P. A short guide to approximation preserving reductions. In: Proceedings of Computational Complexity. Twelfth Annual IEEE Conference. 1997, 262–273
29. Feige U. A threshold of ln n for approximating set cover. Journal of the ACM (JACM), 1998, 45(4): 634–652
30. Sipser M. Introduction to the Theory of Computation. Cengage Learning, 2012
31. Stockmeyer L. Classifying the computational complexity of problems. The journal of symbolic logic, 1987, 52(1): 1–43



Ismael Rodríguez is an Associate Proffessor 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 testing techniques, swarm and evolutionary optimization algorithms, computational complexity, formal methods, and functional programming.

David Rubio obtained a bachelor degree in Computer Science and another bachelor degree in Mathematics from Complutense University of Madrid (Spain) in the year 2019. He has also studied a master on Artificial Intelligence at Universitat Politècnica de València (Spain), where he is currently a researcher at the Biomecanics Institute. His research interests cover image recognition, artificial vision, artificial intelligence, and formal testing techniques.



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 100 papers in international refereed conferences and journals. His research interests cover formal methods, swarm and evolutionary optimization methods, parallel computing, and functional programming.