




Synthesizing adaptive test strategies from temporal logic specifications

Roderick Bloem¹ · Goerschwin Fey^{2,3} · Fabian Greif³ · Robert Könighofer¹ · Ingo Pill¹ · Heinz Riener^{3,4} · Franz Röck¹ 

Published online: 14 October 2019
© The Author(s) 2019

Abstract

Constructing good test cases is difficult and time-consuming, especially if the system under test is still under development and its exact behavior is not yet fixed. We propose a new approach to compute test strategies for reactive systems from a given temporal logic specification using formal methods. The computed strategies are guaranteed to reveal certain simple faults in *every* realization of the specification and for *every* behavior of the uncontrollable part of the system's environment. The proposed approach supports different assumptions on occurrences of faults (ranging from a single transient fault to a persistent fault) and by default aims at unveiling the weakest one. We argue that such tests are also sensitive for more complex bugs. Since the specification may not define the system behavior completely, we use reactive synthesis algorithms with partial information. The computed strategies are *adaptive test strategies* that react to behavior at runtime. We work out the underlying theory of adaptive test strategy synthesis and present experiments for a safety-critical component of a real-world satellite system. We demonstrate that our approach can be applied to industrial

✉ Franz Röck
franz.roeck@iaik.tugraz.at

Roderick Bloem
roderick.bloem@tugraz.at

Goerschwin Fey
goerschwin.fey@tuhh.de

Fabian Greif
fabian.greif@dlr.de

Robert Könighofer
robert.koenighofer@iaik.tugraz.at

Ingo Pill
ipill@ist.tugraz.at

Heinz Riener
heinz.riener@epfl.ch

- ¹ Graz University of Technology, Graz, Austria
- ² Hamburg University of Technology, Hamburg, Germany
- ³ German Aerospace Center, Bremen, Germany
- ⁴ EPFL, Lausanne, Switzerland

specifications and that the synthesized test strategies are capable of detecting bugs that are hard to detect with random testing.

Keywords Automatic test case generation · System testing · Specification testing · Adaptive tests · Synthesis · Reactive systems · Mutation testing

1 Introduction

Model checking [12,48] is an algorithmic approach to prove that a model of a system adheres to its specification. However, model checking cannot always be applied effectively to obtain confidence in the correctness of a system. Possible reasons include scalability issues, third-party IP components for which no code or detailed model is available, or a high effort for building system models that are sufficiently precise. Moreover, model checking cannot verify the final and “live” product but only an (abstracted) model.

Testing is a natural alternative to complement formal methods like model checking, and automatic test case generation helps keeping the effort acceptable. Black-box testing techniques, where tests are derived from a specification rather than the implementation, are particularly attractive: first, tests can be computed before the implementation phase starts, and thus guide the development. Second, the same tests can be reused across different realizations of a given specification. Third, a specification is usually much simpler than its implementation, which gives a scalability advantage. At the same time, the specification focuses on critical functional aspects that require thorough testing. Fault-based techniques [29] are particularly appealing, where the computed tests are guaranteed to reveal all faults in a certain fault class—after all, the foremost goal in testing is to detect bugs.

Methods to derive tests from declarative requirements (see, e.g., [25]) are sparse. One issue in this setting is controllability: the requirements leave plenty of implementation freedom, so they cannot be used to fully predict the system behavior for all given inputs. Consequently, test cases have to be *adaptive*, i.e., able to react to observed behavior at runtime, rather than being fixed input sequences. This is particularly true for *reactive systems* that continuously interact with their environment. Existing methods often work around this complication by requiring a deterministic system model as additional input [24]. Even a probabilistic model fixes the behavior in a way not necessarily required by the specification.

In previous work, we presented a fault-based approach to compute adaptive test strategies for reactive systems [10]. This approach generates tests that enforce certain coverage goals for *every* implementation of a provided specification. The generated tests can be used across realizations of the specification that differ not only in implementation details but also in their observable behavior. This is, e.g., useful for standards and protocols that are implemented by multiple vendors or for systems under development, where the exact behavior is not yet fixed.

Figure 1 outlines the assumed testing setup and shows how the approach for synthesizing adaptive test strategies (illustrated in black) can be integrated in an existing testing flow.

The user provides a specification φ , which describes the requirements of the system under test (SUT) and additionally a fault model δ , which defines the coverage goal in terms of a class of faults for which the tests shall cause a specification violation. Both the specification and the coverage goal are expressed in Linear Temporal Logic (LTL) [46]. By default, our approach supports the detection of transient and permanent faults and distinguishes four fault occurrence frequencies: faults that occur at least (1) once, (2) repeatedly, (3) from some point

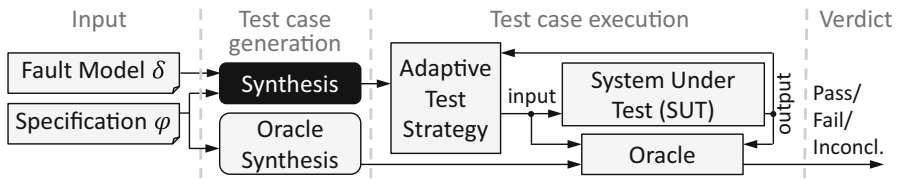


Fig. 1 Testing setup: this paper focuses on test strategy synthesis

on, or (4) permanently. Besides the four default fault occurrence frequencies, a user can also provide a custom frequency using LTL. Our approach then automatically synthesizes a test strategy to reveal a fault for the lowest frequency possible. Such a test strategy guarantees to cause a specification violation if the fault occurs with the defined fault occurrence (and all higher fault occurrence frequencies) and the test is executed long enough. Although test oracles can be synthesized from the specification φ , in this paper, we do not explicitly consider test oracle synthesis, but assume that the oracles are available or manually generated for the test strategies.

Under the hood, reactive synthesis [47] with partial information [33] is used, which provides strong guarantees about all uncertainties: if synthesis is successful and if the computed tests are executed long enough, they reveal all faults from the fault model for every realization of the specification and every behavior of the uncontrollable part of the system's environment. Uncontrollable environment aspects can be seen as part of the system for the purpose of testing. Finally, existing techniques from runtime verification [6] can be used to build an oracle that checks the system behavior against the specification while tests are executed.¹

This paper is an extension of [10]. In summary, this paper presents the following contributions:

- An approach to compute adaptive test strategies for reactive systems from temporal specifications that provide implementation freedom. The tests are guaranteed to reveal certain bugs for *every* realization of the specification.
- The underlying theory is considered in detail, i.e., we show that the approach is sound and complete for many interesting cases and provide additional solutions for other cases that may arise in practice.
- A proof of concept tool, called PARTYStrategy,² that is capable of generating multiple different test strategies, implemented on top of the synthesis tool PARTY [31].
- A post-processing procedure to generalize a test strategy by eliminating input constraints not necessary to guarantee a coverage goal.
- A case study with a safety-critical software component of a real-world satellite system developed in the German Aerospace Center (DLR). We specify the system in LTL, synthesize test strategies, and evaluate the generated adaptive test strategies using code coverage and mutation coverage metrics. Our synthesized test strategies increase both the mutation coverage as well as the code coverage of random test cases by activating behaviors that require complex input sequences that are unlikely to be produced by random testing.

¹ While the semantics of LTL are defined over infinite execution traces, we can only run the tests for a finite amount of time. This can result in inconclusive verdicts [6]. We exclude this issue from the scope of this paper, relying on the user to judge when tests have been executed long enough, and on existing research on interpreting LTL over finite traces [14,15,27,39].

² PARTYStrategy, <https://www.iaik.tugraz.at/content/research/scos/tools/>.

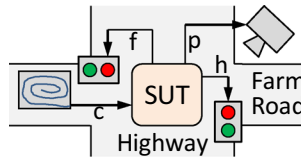


Fig. 2 Traffic light example

The remainder of this paper is organized as follows: Sect. 2 illustrates our approach and presents a motivating example. Section 3 discusses related work. Section 4 gives preliminaries and notation. Our test case generation approach is then worked out in detail in Sect. 5. Section 6 presents the case study and discusses results. Section 7 concludes.

2 Motivating example

Let us develop a traffic light controller for the scenario depicted in Fig. 2. For this highway and farmroad crossing, the controller’s Boolean input signal c describes whether a car is idling at the farmroad. Boolean outputs h and f control the highway and farmroad traffic lights respectively, where a value of true means a green light. Output p controls a camera that takes a picture if a car on the farmroad makes a fast start, i.e., races off immediately when the farmroad light turns green. The controller then should implement the following critical properties:

1. The traffic lights must never be green simultaneously.
2. If a car is waiting at the farmroad, f eventually turns true.
3. If no car is waiting at the farmroad, h eventually becomes true.
4. A picture is taken if a car on the farmroad makes a fast start.

We model the four properties in Linear Temporal Logic (LTL) [46] as

$$\varphi_1 = G(\neg f \vee \neg h) \quad (1)$$

$$\varphi_2 = G(c \rightarrow F f) \quad (2)$$

$$\varphi_3 = G(\neg c \rightarrow F h) \quad (3)$$

$$\varphi_4 = G((\neg f \wedge X(c \wedge f \wedge X \neg c)) \leftrightarrow X X p) \quad (4)$$

where the operator G denotes *always*, F denotes *eventually*, and X denotes *in the nextstep*.

The resulting specification is then:

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$$

To compute a test strategy (only from the specification) that enforces a specification violation by the system under the existence of a certain fault (or class of faults), we have some requirements for our approach.

Enforcing test objectives To mitigate scalability issues, we compute test cases directly from the specification φ . Note that φ focuses on the desired properties only, and allows for plenty of implementation freedom. Our goal is to compute tests that *enforce* certain coverage objectives *independent* of this implementation freedom. Some uncertainties about the SUT behavior may actually be rooted in uncontrollable environment aspects (such as weather conditions) rather than implementation freedom inside the system. But for our testing approach, this

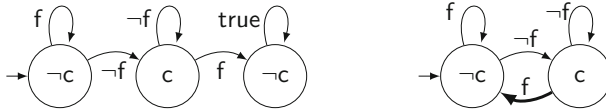


Fig. 3 Two adaptive test strategies for the traffic light controller: on the left, \mathcal{T}_1 that enforces $p = \text{true}$ once. On the right, \mathcal{T}_2 that enforces $p = \text{true}$ infinitely often

makes no difference. We can force the farmroad’s traffic light to turn green ($f = \text{true}$) by relying on a correct implementation of Property 2 and setting $c = \text{true}$. Depending on how the system is implemented, $f = \text{true}$ might also be achieved by setting $c = \text{false}$ all the time, but this is not guaranteed.

Adaptive test strategies Certain test goals may not be enforceable with a static input sequence. For our example, for p to be **true**, a car must do a fast start. Yet, the specification does not prescribe the exact point in time when the traffic light turns to green. We thus synthesize *adaptive* test strategies that guide the controller’s inputs based on the previous inputs and outputs and, therefore, can take advantage of situational possibilities by exploiting previous system behavior.

Figure 3 shows a test strategy \mathcal{T}_1 (on the left) to reach $p = \text{true}$, illustrated as a state machine. States are labeled by the value of controller *input* c (which is an *output of the test strategy* \mathcal{T}_1). Edges represent transitions and are labeled with conditions on observed output values (since the SUT’s outputs are inputs for the test strategy). First, c is set to false to provoke $h = \text{true}$ via Property 3, implying $f = \text{false}$ via Property 1. As soon as this happens, the strategy traverses to the middle state, setting $c = \text{true}$ in order to have $f = \text{true}$ eventually (Property 2). As soon as f switches from false to true, \mathcal{T}_1 sets $c = \text{false}$ in the rightmost state to trigger a picture (Property 4). A system with a permanent stuck-at-0 fault at signal p is unable to satisfy the specification and the resulting violation can be detected by a runtime verification technique.

Coverage objectives We follow a fault-centered approach to define the test objectives to enforce. The user defines a class of (potentially transient) faults. Our approach then computes adaptive test strategies (in form of state machines) that detect these faults. For a permanent stuck-at-0 fault at signal p , our approach could produce the test strategy \mathcal{T}_1 from the previous paragraph: for any correct implementation of φ , the strategy enforces p becoming true at least once. Thus, a faulty version where p is always false necessarily violates the specification, which can be detected [6] during test strategy execution. The test strategy \mathcal{T}_2 , as shown on the right of Fig. 3, is even more powerful since it also reveals stuck-at-0 faults for p that occur not always but only from some point in time onwards. The difference to \mathcal{T}_1 is mainly in the bold transition, which makes \mathcal{T}_2 enforce $p = \text{true}$ infinitely often rather than only once. Our approach distinguishes four fault occurrence frequencies (a fault occurs at least once, infinitely often, from some point on, or always) and synthesizes test strategies for the lowest one for which this is possible.

3 Background and related work

Fault-based testing Fault-based test case generation methods that use the concept of mutation testing [29] seed simple faults into a system implementation (or model) and compute

tests that uncover these faults. Two hypotheses support the value of such tests. The Competent Programmer Hypothesis [1,16] states that implementations are mostly close to correct. The Coupling Effect [16,41] states that tests that detect simple faults are also sensitive to more complex faults. Our approach also relies on these hypotheses. However, in contrast to most existing work that considers permanent faults and deterministic system descriptions that define behavior unambiguously, our approach can deal with transient faults and focuses on uncovering faults in *every* implementation of a given LTL [46] specification (and all behaviors of the uncontrollable part of the system's environment).

Adaptive tests If the behavior of the system or the uncontrollable part of the environment is not fully specified, tests may have to react to observed behavior at runtime to achieve their goals. Many testing theories and test case generation algorithms from specifications of labelled transition systems have been developed. Tretmans [49], for instance, proposed a testing theory analogous to the theory of testing equivalence and preorder for labelled transition systems under the assumption that an implementation communicates with its environment via inputs and outputs. Adaptive tests have been studied by Hierons [28] from a theoretical perspective, relying on fairness assumptions (every non-deterministic behavior is exhibited when trying often enough) or probabilities. Petrenko et al. compute adaptive tests for trace inclusion [43–45] or equivalence [35,42,44] from a specification given as non-deterministic finite state machine, also relying on fairness assumptions. Our work makes no such assumptions but considers the SUT to be fully antagonistic. Aichernig et al. [2] present a method to compute adaptive tests from (non-deterministic) UML state machines. Starting from an initial state, a trace to a goal state, the state that shall be covered by the resulting test case, is searched for every possible system behavior, issuing inconclusive verdicts only if the goal state is not reachable any more. Our approach uses reactive synthesis to enforce reaching the testing goal for all implementations if this is possible.

Testing as a game Yannakakis [52] points out that testing reactive systems can be seen as a game between two players: the tester providing inputs and trying to reveal faults, and the SUT providing outputs and trying to hide faults. The tester can only observe outputs and has thus partial information about the SUT. The goal is to find a strategy for the tester that wins against every SUT. The underlying complexities are studied by Alur et al. [3] in detail. Our work builds upon reactive synthesis [47] (with partial information [33]), which can also be seen as a game. However, we go far beyond the basic idea. We combine the game concept with user-defined fault models, work out the underlying theory, optimize the faults sensitivity in the temporal domain, and present a realization and experiments for LTL [46]. Nachmanson et al. [40] synthesize game strategies as tests for non-deterministic software models, but their approach is not fault-based and focuses on simple reachability goals. A variant of their approach considers the SUT to behave probabilistically with known probabilities [40]. The same model is also used in [8]. Test strategies for reachability goals are also considered by David et al. [13] for timed automata.

Vacuity detection Several approaches [5,7,34] aim at finding cases where a temporal specification is trivially satisfied (e.g., because the left side of an implication is false). Good tests avoid such vacuities to challenge the SUT. The method by Beer et al. [7] can produce witnesses that satisfy the specification non-vacuously, which can serve as tests. Our approach avoids vacuities by requiring that certain faulty SUTs violate the specification.

Testing with a model checker Model checkers can be utilized to compute tests from temporal specifications [25]. The method by Fraser and Ammann [22] ensures that properties are not vacuously satisfied and that faults propagate to observable property violations (using finite-trace semantics for LTL). Tan et al. [50] also define and apply a coverage metric based on vacuity for LTL. Ammann et al. [4] create tests from CTL [12] specifications using model mutations. All these methods assume that a deterministic system model is available in addition to the specification. Fraser and Wotawa [23] also consider non-deterministic models, but issue inconclusive verdicts if the system deviates from the behavior foreseen in the test case. In contrast, we search for test strategies that achieve their goal for *every* realization of the specification. Boroday et al. [11] aim for a similar guarantee (calling it *strong test cases*) using a model checker, but do not consider adaptive test cases, and use a finite state machine as a specification.

Synthesis of test strategies Bounded synthesis [21] aims for finding a system implementation of minimal size in the number of states. Symbolic procedures based on binary decision diagrams [18] and satisfiability solving [31] exist. In our setting, we do not synthesize an implementation of the system, but an adaptive test strategy, i.e., a controller that mimics the system’s environment to enforce a certain test goal. In contrast to a complete implementation of the controller, we strive for finding a partial implementation that assigns values only to those signals that necessarily contribute to reach the test goal. Other signals can be kept non-deterministic and either chosen during execution of the test strategy or randomized. We use a post-processing procedure that eliminates assignments from the test strategy and invokes a model-checker to verify that the test goal is still enforced. This post-processing step is conceptually similar to procedures that aim for counterexample simplification [30] and don’t care identification in test patterns [38]. Jin et al. [30] separate a counterexample trace into forced segments that unavoidably progress towards the specification violation and free segments that, if avoided, may have prevented the specification violation. Our post-processing step is similar, but instead of counterexamples, adaptive test strategies are post-processed. Miyase and Kajihara [38] present an approach to identify don’t cares in test patterns of combinational circuits. In contrast to combinational circuits, we deal with reactive systems. Instead of post-processing a complete test strategy, a partial test strategy can be directly synthesized by modifying a synthesis procedure to compute minimum satisfying assignments [17]. Although feasible, modifying a synthesis procedure requires a lot of work. Our post-processing procedure uses the synthesis procedure in a plug-and-play fashion and does not require manual changes in the synthesis procedure.

4 Preliminaries and notation

Traces We want to test reactive systems that have a finite set $I = \{i_1, \dots, i_m\}$ of Boolean inputs and a finite set $O = \{o_1, \dots, o_n\}$ of Boolean outputs. The input alphabet is $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and $\Sigma = 2^{I \cup O}$. An infinite word $\bar{\sigma}$ over Σ is an (*execution*) *trace* and the set Σ^ω is the set of all infinite words over Σ .

Linear Temporal Logic We use *Linear Temporal Logic (LTL)* [46] as a specification language for reactive systems. The syntax is defined as follows: every input or output $p \in I \cup O$ is an LTL formula; and if φ_1 and φ_2 are LTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $X\varphi_1$ and $\varphi_1 \cup \varphi_2$. We write $\bar{\sigma} \models \varphi$ to denote that a trace $\bar{\sigma} = \sigma_0\sigma_1 \dots \in \Sigma^\omega$ satisfies LTL formula φ . This is defined inductively as follows:

- $\sigma_0\sigma_1\sigma_2 \dots \models p$ iff $p \in \sigma_0$,
- $\bar{\sigma} \models \neg\varphi$ iff $\bar{\sigma} \not\models \varphi$,
- $\bar{\sigma} \models \varphi_1 \vee \varphi_2$ iff $\bar{\sigma} \models \varphi_1$ or $\bar{\sigma} \models \varphi_2$,
- $\sigma_0\sigma_1\sigma_2 \dots \models X\varphi$ iff $\sigma_1\sigma_2 \dots \models \varphi$, and
- $\sigma_0\sigma_1 \dots \models \varphi_1 \cup \varphi_2$ iff $\exists j \geq 0. \sigma_j\sigma_{j+1} \dots \models \varphi_2 \wedge \forall 0 \leq k < j. \sigma_k\sigma_{k+1} \dots \models \varphi_1$.

That is, $X\varphi$ requires φ to hold in the *next* step, and $\varphi_1 \cup \varphi_2$ means that φ_1 must hold *until* φ_2 holds (and φ_2 must hold eventually). We also use the usual abbreviations $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$, $F\varphi = \text{true} \cup \varphi$ (meaning that φ must hold *eventually*), and $G\varphi = \neg F\neg\varphi$ (φ must hold *always*). By $\varphi[x \leftarrow y]$ we denote the LTL formula φ where all occurrences of x have been textually replaced by y .

Mealy machines We use Mealy machines to model the reactive system under test. A *Mealy machine* is a tuple $\mathcal{S} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \rightarrow Q$ is a total transition function, and $\lambda : Q \times \Sigma_I \rightarrow \Sigma_O$ is a total output function. Given the input trace $\bar{\sigma}_I = x_0x_1 \dots \in \Sigma_I^\omega$, \mathcal{S} produces the output trace $\bar{\sigma}_O = \mathcal{S}(\bar{\sigma}_I) = \lambda(q_0, x_0)\lambda(q_1, x_1) \dots \in \Sigma_O^\omega$, where $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$. That is, in every time step i , the Mealy machine reads the input letter $x_i \in \Sigma_I$, responds with an output letter $\lambda(q_i, x_i) \in \Sigma_O$, and updates its state to $q_{i+1} = \delta(q_i, x_i)$. A Mealy machine can directly model synchronous hardware designs, but also other systems with inputs and outputs evolving in discrete time steps. We write $\text{Mealy}(I, O)$ for the set of all Mealy machines with inputs I and outputs O .

Moore machines We use Moore machines to describe test strategies. A *Moore machine* is a special Mealy machine with $\forall q \in Q. \forall x, x' \in \Sigma_I. \lambda(q, x) = \lambda(q, x')$. That is, $\lambda(q, x)$ is insensitive to x , i.e., becomes a function $\lambda : Q \rightarrow \Sigma_O$. This means that the input x_i at step i can affect the next state q_{i+1} and thus the next output $\lambda(q_{i+1})$ but not the current output $\lambda(q_i)$. We write $\text{Moore}(I, O)$ for the set of all Moore machines with inputs I and outputs O .

Composition Given Mealy machines $\mathcal{S}_1 = (Q_1, q_{0,1}, 2^I, 2^{O_1}, \delta_1, \lambda_1) \in \text{Mealy}(I, O_1)$ and $\mathcal{S}_2 = (Q_2, q_{0,2}, 2^{I \cup O_1}, 2^{O_2}, \delta_2, \lambda_2) \in \text{Mealy}(I \cup O_1, O_2)$, we write $\mathcal{S} = \mathcal{S}_1 \circ \mathcal{S}_2$ for their sequential composition $\mathcal{S} = (Q_1 \times Q_2, (q_{0,1}, q_{0,2}), 2^I, 2^{O_1 \cup O_2}, \delta, \lambda)$, where $\mathcal{S} \in \text{Mealy}(I, O_1 \cup O_2)$ with $\delta((q_1, q_2), x) = (\delta_1(q_1, x), \delta_2(q_2, x \cup \lambda_1(q_1, x)))$ and $\lambda((q_1, q_2), x) = \lambda_1(q_1, x) \cup \lambda_2(q_2, x \cup \lambda_1(q_1, x))$. Note that $x \in 2^I$.

Systems and test strategies A *reactive system* \mathcal{S} is a Mealy machine. An (*adaptive*) *test strategy* is a Moore machine $\mathcal{T} = (T, t_0, \Sigma_O, \Sigma_I, \Delta, \Lambda)$ with input and output alphabet swapped. That is, \mathcal{T} produces values for input signals and reacts to values of output signals. A test strategy \mathcal{T} can be *run* on a system \mathcal{S} as follows. In every time step i (starting with $i = 0$), \mathcal{T} first computes the next input $x_i = \Lambda(t_i)$. Then, the system computes the output $y_i = \lambda(q_i, x_i)$. Finally, both machines compute their next state $t_{i+1} = \Delta(t_i, y_i)$ and $q_{i+1} = \delta(q_i, x_i)$. We write $\bar{\sigma}(\mathcal{T}, \mathcal{S}) = (x_0 \cup y_0)(x_1 \cup y_1) \dots \in \Sigma^\omega$ for the resulting execution trace. If $\mathcal{T} = (T, t_0, 2^{O'}, \Sigma_I, \Delta, \Lambda) \in \text{Moore}(O', I)$ can observe only a subset $O' \subseteq O$ of the outputs, we define $\bar{\sigma}(\mathcal{T}, \mathcal{S})$ with $t_{i+1} = \Delta(t_i, y_i \cap O')$. A *test suite* is a set $\text{TS} \subseteq \text{Moore}(O, I)$ of adaptive test strategies.

Realizability A Mealy machine $\mathcal{S} \in \text{Mealy}(I, O)$ *realizes* an LTL formula φ , written $\mathcal{S} \models \varphi$, if $\forall \mathcal{M} \in \text{Moore}(O, I). \bar{\sigma}(\mathcal{M}, \mathcal{S}) \models \varphi$. An LTL formula φ is *Mealy-realizable* if there exists a Mealy machine that realizes it. A Moore machine $\mathcal{M} \in \text{Moore}(I, O)$ realizes

φ , written $\mathcal{M} \models \varphi$, if $\forall \mathcal{S} \in \text{Mealy}(O, I) . \bar{\sigma}(\mathcal{M}, \mathcal{S}) \models \varphi$. A *model checking procedure* checks if a given Mealy (Moore) machine \mathcal{S} (\mathcal{M}) realizes an LTL specification φ and returns true iff $\mathcal{S} \models \varphi$ ($\mathcal{M} \models \varphi$) holds. We denote the call of a model checking procedure by $\text{modelcheck}(\mathcal{S}, \varphi)$ ($\text{modelcheck}(\mathcal{M}, \varphi)$).

Reactive synthesis We use reactive synthesis [47] to compute test strategies. A *reactive (Moore, LTL) synthesis procedure* takes as input a set I of Boolean inputs, a set O of Boolean outputs, and an LTL specification φ over these signals. It produces a Moore machine $\mathcal{M} \in \text{Moore}(I, O)$ that realizes φ , or the message *unrealizable* if no such Moore machine exists. We denote this computation by $\text{synt}(I, O, \varphi)$.

Synthesis with partial information [33] is defined similarly, but this problem takes a subset $I' \subseteq I$ of the inputs as an additional input. As output, the synthesis procedure produces a Moore machine $\mathcal{M}' = \text{synt}_p(I, O, \varphi, I')$ with $\mathcal{M}' \in \text{Moore}(I', O)$ that realizes φ while only observing the inputs I' , or the message *unrealizable* if no such Moore machine exists. We assume that both synthesis procedure, synt and synt_p , can be called *incrementally* with an additional parameter Θ , where Θ denotes a set of Moore machines. The incremental synthesis procedures $\mathcal{M} = \text{synt}(I, O, \varphi, \Theta)$ and $\mathcal{M}' = \text{synt}_p(I, O, \varphi, I', \Theta)$ compute Moore machines \mathcal{M} and \mathcal{M}' , respectively, as before but with the additional constraints that $\mathcal{M}, \mathcal{M}' \notin \Theta$.

In synthesis, we often use *assumptions* A and *guarantees* G . The assumptions are meant to state the requirements on the environment under which the guarantees should be met by the synthesized system. Technically, we synthesize a system \mathcal{M} that fulfills the specification $A \rightarrow G$. Obviously, whenever the environment violates the assumptions, the implication is trivially satisfied and the behavior of the system is irrelevant.

For the purposes of this paper, we take synthesis as a black box. We will not describe the technical details of synthesis here but rather refer the interested reader to [9] for details.

Fault versus failure A Mealy machine $\mathcal{S} \in \text{Mealy}(I, O)$ is *faulty* with respect to LTL formula φ (specification) iff $\mathcal{S} \not\models \varphi$, i.e., $\exists \mathcal{M} \in \text{Moore}(O, I) . \bar{\sigma}(\mathcal{M}, \mathcal{S}) \not\models \varphi$. We call a trace $\bar{\sigma}(\mathcal{M}, \mathcal{S})$ that uncovers a faulty behavior of \mathcal{S} a *failure* and a deviation between \mathcal{S} and any correct realization \mathcal{S}' , i.e., $\mathcal{S}' \models \varphi$, a *fault*. For a fixed faulty \mathcal{S} , there are multiple correct \mathcal{S}' that realize φ and thus a fault in \mathcal{S} can be characterized by multiple, different ways. As a simplification, we assume that in practice every faulty \mathcal{S} is close to a correct \mathcal{S}' and only deviates in a simple fault. In the next section, we will show how this idea can be leveraged to determine test suites independent of the implementation and the concrete fault manifestation.

5 Synthesis of adaptive test strategies

This section presents our black-box testing approach for synthesizing adaptive test strategies for reactive systems specified in LTL. First, we elaborate on the coverage objective we aim to achieve. Then we present our strategy synthesis algorithm. Finally, we discuss extensions and variants of the algorithm.

5.1 Coverage objective for test strategy computation

Many coverage metrics [37] exist to assess the quality of a test suite. Since the goal in testing is to detect bugs, we follow a fault-centered approach: a test suite has high quality if it reveals

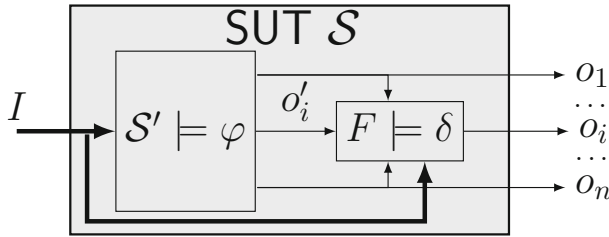


Fig. 4 Coverage goal illustration for fault

certain kinds of faults in a system. In contrast to existing approaches such as mutation testing which model potential faults in a concrete implementation, we provide a novel fault model that models faults on the specification-level, agnostic of the concrete implementation. We assume that the SUT is “almost correct” and contains only simple faults that propagate to at most one output.³ As illustrated in Fig. 4, we formalize this assumption on specification-level and model the SUT as composed of a correct implementation S' of the specification φ and a fault F that affects one output. In order to make our approach flexible, we allow the user to define the considered faults as an LTL formula δ . Through δ , the user can define both permanent and transient faults of various types. For instance, $\delta = F(o_i \leftrightarrow \neg o'_i)$ describes a bit-flip that occurs at least once, $GF \neg o_i$ models a stuck-at-0 fault that occurs infinitely often, and $G(X(o_i) \leftrightarrow o'_i)$ models a permanent shift by one time step. We strive for a test suite that reveals every fault that satisfies δ for every realization of φ . This renders the test suite independent of the implementation and the concrete fault manifestation. The following definition formalizes this intuition into a coverage objective.

Definition 1 A test suite $TS \subseteq \text{Moore}(O, I)$ for a system with inputs I , outputs O , and specification φ is *universally complete*⁴ with respect to a given fault model δ iff

$$\forall o_i \in O . \forall S' \in \text{Mealy}(I, O \cup \{o'_i\} \setminus \{o_i\}) . \forall F \in \text{Mealy}(I \cup O \cup \{o'_i\} \setminus \{o_i\}, \{o_i\}) . \exists \mathcal{T} \in TS . \left((S' \models \varphi[o_i \leftarrow o'_i] \wedge F \models \delta) \rightarrow (\overline{\sigma}(\mathcal{T}, S' \circ F) \not\models \varphi) \right). \quad (5)$$

That is, for every output o_i , system $S' \models \varphi[o_i \leftarrow o'_i]$, and fault $F \models \delta$, TS must contain a test strategy \mathcal{T} that reveals the fault by causing a specification violation (Fig. 4). Note that the test strategies $\mathcal{T} \in TS \subseteq \text{Moore}(O, I)$ cannot observe the signal o'_i . The reason is that this signal o'_i does not exist in the real system implementation(s) on which we run our tests—it was only introduced to define our coverage objective.

There can be an unbounded number of system realizations $S' \models \varphi[o_i \leftarrow o'_i]$ and faults $F \models \delta$. Computing a separate test strategy for each combination is thus not a viable option. We rather strive for computing only one test strategy per output variable.

Theorem 1 A *universally complete test suite* $TS \subseteq \text{Moore}(O, I)$ with respect to fault model δ exists for a system with inputs I , outputs O , and specification φ if

³ This fault model is different from the standard fault model in mutation testing, which considers simple faults in a concrete implementation that can affect multiple outputs.

⁴ The word “complete” indicates that every considered fault is revealed at every output. The word “universal” indicates that this is achieved for every (otherwise correct) system.

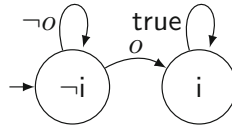


Fig. 5 Test strategy \mathcal{T}_5

$$\forall o_i \in O . \exists \mathcal{T} \in \text{Moore}(O, I) . \forall \mathcal{S} \in \text{Mealy}(I, O \cup \{o'_i\}) . \bar{\sigma}(\mathcal{T}, \mathcal{S}) \models ((\varphi[o_i \leftarrow o'_i] \wedge \delta) \rightarrow \neg\varphi). \quad (6)$$

Proof Equation 6 implies

$$\forall o_i \in O . \forall \mathcal{S} \in \text{Mealy}(I, O \cup \{o'_i\}) . \exists \mathcal{T} \in \text{Moore}(O, I) . (\mathcal{S} \models \varphi[o_i \leftarrow o'_i] \wedge \delta) \rightarrow (\bar{\sigma}(\mathcal{T}, \mathcal{S}) \not\models \varphi) \quad (7)$$

because (a) going from $\exists \mathcal{T} \forall \mathcal{S}$ to $\forall \mathcal{S} \exists \mathcal{T}$ can only make the formula weaker, and (b) $\mathcal{S} \models \varphi[o_i \leftarrow o'_i] \wedge \delta$ implies $\bar{\sigma}(\mathcal{T}, \mathcal{S}) \models \varphi[o_i \leftarrow o'_i] \wedge \delta$ for all \mathcal{T} , which can only make the left side of the implication stronger. In turn, Eq. 7 is equivalent to

$$\begin{aligned} \forall o_i \in O . \forall \mathcal{S}' \in \text{Mealy}(I, O \cup \{o'_i\} \setminus \{o_i\}) . \\ \forall F \in \text{Mealy}(I \cup O \cup \{o'_i\} \setminus \{o_i\}, \{o_i\}) . \exists \mathcal{T} \in \text{Moore}(O, I) . \\ (\mathcal{S}' \models \varphi[o_i \leftarrow o'_i] \wedge F \models \delta) \rightarrow (\bar{\sigma}(\mathcal{T}, \mathcal{S}' \circ F) \not\models \varphi) \quad (8) \end{aligned}$$

because for a given $\mathcal{S}' \models \varphi[o_i \leftarrow o'_i]$ and $F \models \delta$ from Eq. 8 we can define an equivalent system $\mathcal{S} = (\mathcal{S}' \circ F) \in \text{Mealy}(I, O \cup \{o'_i\})$ for Eq. 7 such that $\mathcal{S} \models \varphi[o_i \leftarrow o'_i] \wedge \delta$ is satisfied. Also, for a given $\mathcal{S} \models \varphi[o_i \leftarrow o'_i] \wedge \delta$ from Eq. 7 we can define a corresponding $\mathcal{S}' \models \varphi[o_i \leftarrow o'_i]$ and $F \models \delta$ by stripping off different outputs.

Theorem 1 states that Eq. 6 is a sufficient condition for a universally complete test suite to exist. If it were also a necessary condition, then computing one test strategy per output signal would be enough. Unfortunately, this is not the case in general.

Example 1 Consider a system with input $I = \{i\}$, output $O = \{o\}$, and specification $\varphi = (\mathbf{G}(i \rightarrow \mathbf{G}i) \wedge \mathbf{F}i) \rightarrow (\mathbf{G}(o \rightarrow \mathbf{G}o) \wedge \mathbf{F}o \wedge \mathbf{G}(i \vee \neg o))$. The left side of the implication assumes that the input i is set to true at some point, after which i remains true. The right side requires the same for the output o . In addition, o must not be raised while i is still false. This specification is realizable (e.g., by always setting $o = i$). The test suite $\text{TS} = \{\mathcal{T}_5\}$ with \mathcal{T}_5 shown in Fig. 5 is universally complete with respect to fault model $\delta = \mathbf{F}(o \leftrightarrow \neg o')$, which requires the output to flip at least once: as long as i is false, any correct system implementation $\mathcal{S}' \in \text{Mealy}(\{i\}, \{o'\}) \models \varphi[o_i \leftarrow o'_i]$ must keep the output $o' = \text{false}$. Eventually, $F \models \delta$ must flip the output o to true. When this happens, i is set to true by \mathcal{T}_5 so that the resulting trace $\bar{\sigma}(\mathcal{T}, \mathcal{S}' \circ F)$ violates φ . Still, Eq. 6 is false.⁵ Strategy \mathcal{T}_5 does not satisfy Eq. 6 because for the system $\mathcal{S} \in \text{Mealy}(\{i\}, \{o, o'\})$ that sets $o' = \text{true}$ and $o = \text{false}$ in all time steps, we have $\bar{\sigma}(\mathcal{T}_5, \mathcal{S}) \models (\varphi[o_i \leftarrow o'_i] \wedge \delta \wedge \varphi)$. The reason is that i stays false, so $\varphi[o_i \leftarrow o'_i]$ and φ are vacuously satisfied by $\bar{\sigma}(\mathcal{T}_5, \mathcal{S})$. The formula δ is satisfied because $o \leftrightarrow \neg o'$ holds in all time steps. Thus, \mathcal{S} is a counterexample to \mathcal{T}_5 satisfying Eq. 6. Similar counterstrategies exist for all other test strategies.

⁵ This is (at least partially) confirmed by our test strategy synthesis tool: it reports that no test strategy with less than 12 states can satisfy Eq. 6.

The fact that Eq. 6 is not a necessary condition for a universally complete test suite to exist is somewhat surprising, especially in the light of the following two lemmas. Based on these lemmas, the subsequent propositions will show that Eq. 6 is both sufficient and necessary (i.e., one test per output is enough) for many interesting cases.

The following lemma, which is based on the determinacy of complete-information games, states that the following two conditions are equivalent: (1) there is a single test strategy that shows a fault in any implementation and (2) for any implementation there is a strategy that shows the fault. This means that in certain settings, a single test strategy suffices to find a fault.

Lemma 1 *For every LTL specification ψ over some inputs I and outputs O , we have that $\exists T \in \text{Moore}(O, I) . \forall S \in \text{Mealy}(I, O) . \bar{\sigma}(T, S) \models \psi$ holds if and only if $\forall S \in \text{Mealy}(I, O) . \exists T \in \text{Moore}(O, I) . \bar{\sigma}(T, S) \models \psi$ holds.*

Proof Synthesis from LTL specifications under complete information is (finite memory) determined [36], which means that either $\exists T \in \text{Moore}(O, I) . \forall S \in \text{Mealy}(I, O) . \bar{\sigma}(T, S) \models \psi$ or $\exists S \in \text{Mealy}(I, O) . \forall T \in \text{Moore}(O, I) . \bar{\sigma}(T, S) \models \neg\psi$ holds, but not both. Less formally we can say that either there exists a test strategy T that satisfies ψ for all systems S , or there exists a system S that can violate ψ for all test strategies T . From that, it follows that

$$\begin{aligned} & \exists T \in \text{Moore}(O, I) . \forall S \in \text{Mealy}(I, O) . \bar{\sigma}(T, S) \models \psi \\ \text{iff } & \neg \exists S \in \text{Mealy}(I, O) . \\ & \forall T \in \text{Moore}(O, I) . \bar{\sigma}(T, S) \models \neg\psi \\ \text{iff } & \forall S \in \text{Mealy}(I, O) . \exists T \in \text{Moore}(O, I) . \bar{\sigma}(T, S) \models \psi. \end{aligned}$$

The second lemma is again limited to perfect information. It states that the following two conditions are equivalent: (1) for any system that fulfills an assumption A , there is a test strategy that elicits behavior satisfying a guarantee G and (2) for any system there is a test strategy that elicits behavior satisfying the LTL property $A \rightarrow G$. This lemma implies that in the case of complete information, an LTL synthesis tool suffices.

Lemma 2 *For all LTL specifications A, G over inputs I and outputs O , we have that*

$$\begin{aligned} & \forall S \in \text{Mealy}(I, O) . \exists T \in \text{Moore}(O, I) . \\ & (\mathcal{S} \models A) \rightarrow (\bar{\sigma}(T, S) \models G) \end{aligned} \tag{9}$$

$$\text{iff } \forall S \in \text{Mealy}(I, O) . \exists T \in \text{Moore}(O, I) . \bar{\sigma}(T, S) \models A \rightarrow G. \tag{10}$$

Proof Direction \Rightarrow : We show that Eq. 10 being false contradicts with Eq. 9 being true.

$$\begin{aligned} & \neg \forall S \in \text{Mealy}(I, O) . \exists T \in \text{Moore}(O, I) . \\ & \bar{\sigma}(T, S) \models A \rightarrow G \\ \text{iff } & \exists S \in \text{Mealy}(I, O) . \forall T \in \text{Moore}(O, I) . \\ & \bar{\sigma}(T, S) \models A \wedge \neg G \\ \text{iff } & \exists S \in \text{Mealy}(I, O) . \mathcal{S} \models (A \wedge \neg G), \text{ which implies} \\ & \exists S \in \text{Mealy}(I, O) . \forall T \in \text{Moore}(O, I) . \\ & (\mathcal{S} \models A) \wedge (\bar{\sigma}(T, S) \models \neg G). \end{aligned}$$

Direction \Leftarrow : Using the LTL semantics, we can rewrite $\bar{\sigma}(\mathcal{T}, \mathcal{S}) \models A \rightarrow G$ in Eq. 10 as $(\bar{\sigma}(\mathcal{T}, \mathcal{S}) \models A) \rightarrow (\bar{\sigma}(\mathcal{T}, \mathcal{S}) \models G)$. Since $\mathcal{S} \models A$ implies $\bar{\sigma}(\mathcal{T}', \mathcal{S}) \models A$ for every $\mathcal{T}' \in \text{Moore}(I, O)$, the assumption in Eq. 9 is not weaker, so Eq. 9 is not stronger.

Yet, in our setting, test strategies $\mathcal{T} \in \text{Moore}(O, I)$ have incomplete information about the system $\mathcal{S} \in \text{Mealy}(I, O \cup \{o'_i\})$ because they cannot observe o'_i . Still, \mathcal{T} must enforce $(\varphi[o_i \leftarrow o'_i] \wedge \delta) \rightarrow \neg\varphi$, which refers to this hidden signal. Thus, Lemma 1 and 2 cannot be applied to Eq. 6 in general. However, in cases where there is (effectively) no hidden information, the lemmas can be used to prove that Eq. 6 is both a necessary and a sufficient condition for a universally complete test suite to exist. The following propositions show that this holds for many cases of practical interest.

The intuitive reason is that $\varphi[o_i \leftarrow o'_i]$ can be rewritten to $\varphi[o_i \leftarrow \psi]$ in Eq. 6, which eliminates the hidden signal such that Lemmas 1 and 2 can be applied.

Proposition 1 *Given a fault model of the form $\delta = G(o'_i \leftrightarrow \psi)$, where ψ is an LTL formula over I and O , a universally complete test suite $TS \subseteq \text{Moore}(O, I)$ with respect to δ, I, O , and φ exists if and only if Eq. 6 holds.*

Proof $\varphi[o_i \leftarrow o'_i] \wedge G(o'_i \leftrightarrow \psi)$ is equivalent to $\varphi[o_i \leftarrow \psi] \wedge G(o'_i \leftrightarrow \psi)$. Thus, Eq. 6 becomes

$$\forall o_i \in O . \exists \mathcal{T} \in \text{Moore}(O, I) . \forall \mathcal{S} \in \text{Mealy}(I, O \cup \{o'_i\}) . \bar{\sigma}(\mathcal{T}, \mathcal{S}) \models (\varphi[o_i \leftarrow \psi] \wedge G(o'_i \leftrightarrow \psi)) \rightarrow \neg\varphi,$$

which is equivalent to

$$\forall o_i \in O . \exists \mathcal{T} \in \text{Moore}(O, I) . \forall \mathcal{S} \in \text{Mealy}(I, O) . \bar{\sigma}(\mathcal{T}, \mathcal{S}) \models \varphi[o_i \leftarrow \psi] \rightarrow \neg\varphi.$$

Because of the G operator, a unique value for o'_i exist in all time steps and thus, o'_i is just an abbreviation for ψ . Whether this abbreviation o'_i is available as output of \mathcal{S} or not is irrelevant, because \mathcal{T} cannot observe o'_i anyway. Since o'_i no longer occurs, Lemmas 1 and 2 can be applied to prove equivalence between Eq. 6 and

$$\forall o_i \in O . \forall \mathcal{S} \in \text{Mealy}(I, O) . \exists \mathcal{T} \in \text{Moore}(O, I) . (\mathcal{S} \models \varphi[o_i \leftarrow \psi]) \rightarrow (\bar{\sigma}(\mathcal{T}, \mathcal{S}) \not\models \varphi).$$

As \mathcal{T} cannot observe o'_i , it is irrelevant whether the truth value of ψ is available as additional output o'_i of \mathcal{S} or not. Hence, the above formula is equivalent to

$$\forall o_i \in O . \forall \mathcal{S} \in \text{Mealy}(I, O \cup \{o'_i\}) . \exists \mathcal{T} \in \text{Moore}(O, I) . (\mathcal{S} \models (\varphi[o_i \leftarrow \psi] \wedge G(o'_i \leftrightarrow \psi))) \rightarrow (\bar{\sigma}(\mathcal{T}, \mathcal{S}) \not\models \varphi)$$

and

$$\forall o_i \in O . \forall \mathcal{S} \in \text{Mealy}(I, O \cup \{o'_i\}) . \exists \mathcal{T} \in \text{Moore}(O, I) . (\mathcal{S} \models (\varphi[o_i \leftarrow o'_i] \wedge \delta)) \rightarrow (\bar{\sigma}(\mathcal{T}, \mathcal{S}) \not\models \varphi),$$

i.e., to Eq. 7. The remaining steps can be taken from the proof of Theorem 1.

Proposition 1 entails that computing one test strategy per output $o_i \in O$ is enough for fault models such as permanent bit flips (defined by $\delta = G(o'_i \leftrightarrow \neg o_i)$).

Proposition 2 *If the fault model δ does not reference o'_i , a universally complete test suite $TS \subseteq \text{Moore}(O, I)$ with respect to δ, I, O , and φ exists iff Eq. 6 holds.*

Proof We show that Eq. 6 holds if and only if Eq. 7 holds. The remaining steps have already been proven for Theorem 1.

Lemma 3 *Equation 6 holds if and only if*

$$\begin{aligned} \forall o_i \in O . \exists T \in \text{Moore}(O, I) . \forall S \in \text{Mealy}(I, O) . \\ \bar{\sigma}(T, S) \models \delta \rightarrow \neg\varphi. \end{aligned} \tag{11}$$

Proof Direction \Leftarrow is obvious because Eq. 6 contains stronger assumptions (and $\forall S \in \text{Mealy}(I, O)$ can be changed to $\forall S \in \text{Mealy}(I, O \cup \{o'_i\})$ in Eq. 11 because $\delta \rightarrow \neg\varphi$ does not contain o'_i).

Direction \Rightarrow : We show that Eq. 11 being false contradicts with Eq. 6 being true.

$$\begin{aligned} \neg\forall o_i \in O . \exists T \in \text{Moore}(O, I) . \\ \forall S \in \text{Mealy}(I, O) . \bar{\sigma}(T, S) \models \delta \rightarrow \neg\varphi \end{aligned} \tag{12}$$

$$\begin{aligned} \text{iff } \exists o_i \in O . \forall T \in \text{Moore}(O, I) . \\ \exists S \in \text{Mealy}(I, O) . \bar{\sigma}(T, S) \models \delta \wedge \varphi \end{aligned} \tag{13}$$

$$\begin{aligned} \text{iff } \exists o_i \in O . \exists S \in \text{Mealy}(I, O) . \\ \forall T \in \text{Moore}(O, I) . \bar{\sigma}(T, S) \models (\delta \wedge \varphi) \end{aligned} \tag{14}$$

$$\text{iff } \exists o_i \in O . \exists S \in \text{Mealy}(I, O) . S \models \delta \wedge \varphi \tag{15}$$

$$\begin{aligned} \text{iff } \exists o_i \in O . \exists S' \in \text{Mealy}(I, O \cup \{o'_i\}) . \\ S' \models \varphi[o_i \leftarrow o'_i] \wedge \delta \wedge \varphi, \end{aligned} \tag{16}$$

$$\begin{aligned} \text{iff } \exists o_i \in O . \exists S' \in \text{Mealy}(I, O \cup \{o'_i\}) . \\ \forall T \in \text{Moore}(O \cup \{o'_i\}, I) . \\ \bar{\sigma}(T, S) \models \varphi[o_i \leftarrow o'_i] \wedge \delta \wedge \varphi, \end{aligned} \tag{17}$$

$$\begin{aligned} \text{iff } \exists o_i \in O . \forall T \in \text{Moore}(O \cup \{o'_i\}, I) . \\ \exists S' \in \text{Mealy}(I, O \cup \{o'_i\}) . \\ \bar{\sigma}(T, S) \models \varphi[o_i \leftarrow o'_i] \wedge \delta \wedge \varphi, \end{aligned} \tag{18}$$

$$\begin{aligned} \Rightarrow \exists o_i \in O . \forall T \in \text{Moore}(O, I) . \\ \exists S' \in \text{Mealy}(I, O \cup \{o'_i\}) . \\ \bar{\sigma}(T, S) \models \varphi[o_i \leftarrow o'_i] \wedge \delta \wedge \varphi, \end{aligned} \tag{19}$$

which contradicts Eq. 6. (13) \Leftrightarrow (14) holds because of Lemma 1 and (Eq. 15) \Leftrightarrow (Eq. 16) holds because $\delta \wedge \varphi$ does not contain o'_i , so S' can be S with $o'_i \leftrightarrow o_i$. (Eq. 17) \Leftrightarrow (Eq. 18) holds because of Lemma 1. Finally, (Eq. 18) implies (Eq. 19) because T has less information in (Eq. 19).

Lemma 4 *Equation 11 holds if and only if Eq. 7 holds.*

Proof Direction \Rightarrow : is obvious because Eq. 11 is equivalent to Eq. 6 (Lemma 3) and Eq. 6 implies Eq. 7 (see proof for Theorem 1).

Direction \Leftarrow : we show that Eq. 11 being false contradicts Eq. 7 being true. Equation 11 being false implies Eq. 16 (see above). As $S' \models (\varphi[o_i \leftarrow o'_i] \wedge \delta \wedge \varphi)$ implies $(S' \models$

$\varphi[o_i \leftarrow o'_i] \wedge \delta) \wedge (\bar{\sigma}(\mathcal{T}, S) \models \varphi)$ for all $\mathcal{T} \in \text{Moore}(O \cup \{o'_i\}, I)$ and thus also for all $\mathcal{T} \in \text{Moore}(O, I)$, Eq. 7 cannot hold.

Thus, the assumption $S' \models \varphi[o_i \leftarrow o'_i]$ can be dropped from Eq. 5 if the fault model does not reference o'_i . Correspondingly, $\bar{\sigma}(\mathcal{T}, S) \models ((\varphi[o_i \leftarrow o'_i] \wedge \delta) \rightarrow \neg\varphi)$ simplifies to $\bar{\sigma}(\mathcal{T}, S) \models (\delta \rightarrow \neg\varphi)$ in Eq. 6. Since o'_i is now gone, Lemmas 1 and 2 apply. In general, the assumption $S' \models \varphi[o_i \leftarrow o'_i]$ is needed to prevent a faulty system $S' \not\models \varphi[o_i \leftarrow o'_i]$ from compensating the fault $F \models \delta$ such that $S' \circ F \models \varphi$. E.g., for $I = \emptyset$, $O = \{o\}$, $\varphi = G o$ and $\delta = G(o \leftrightarrow \neg o')$, Eq. 5 would be false without $S' \models \varphi[o_i \leftarrow o'_i]$ because there exists an S' that always sets $o' = \text{false}$, in which case $S' \circ F$ has o correctly set to true. However, if δ does not reference o' , such a fault compensation is not possible.

Proposition 2 applies to permanent or transient stuck-at-0 or stuck-at-1 faults (e.g., $\delta = F \neg o_i$ or $\delta = GF o_i$), but also to faults where o_i keeps its previous value (e.g., $\delta = F(o_i \leftrightarrow X o_i)$) or takes the value of a different input or output (e.g., $\delta = GF(o_i \leftarrow i_3)$). Together with Proposition 1, it shows that computing one test strategy per output is enough for many interesting fault models. Finally, even if neither Propositions 1 nor 2 applies, computing one test strategy per output may still suffice for the concrete φ and δ at hand. In the next section, we thus rely on Eq. 6 to compute one test strategy per output in order to obtain universally complete test suites.

5.2 Test strategy computation

Basic idea Our test case generation approach builds upon Theorem 1: for every output $o_i \in O$, we want to find a test strategy $\mathcal{T}_i \in \text{Moore}(O, I)$ such that $\forall S \in \text{Mealy}(I, O \cup \{o'_i\}). \bar{\sigma}(\mathcal{T}_i, S) \models (\varphi[o_i \leftarrow o'_i] \wedge \delta) \rightarrow \neg\varphi$ holds. Recall from Sect. 4 that a synthesis procedure $\mathcal{M} = \text{synt}_p(I, O, \psi, I', \Theta)$ with partial information computes a Moore machine $\mathcal{M} \in \text{Moore}(I', O) \setminus \Theta$ with $I' \subseteq I$ such that a certain LTL objective ψ is enforced in all environments, i.e., $\forall S \in \text{Mealy}(O, I). \bar{\sigma}(\mathcal{M}, S) \models \psi$. If no such \mathcal{M} exists, synt_p returns unrealizable. Also recall that a test strategy is a Moore machine with input and output signals swapped. We can thus call $\mathcal{T}_i := \text{synt}_p(O \cup \{o'_i\}, I, (\varphi[o_i \leftarrow o'_i] \wedge \delta) \rightarrow \neg\varphi, O, \Theta)$ for every output $o_i \in O$ in order to obtain a universally complete test suite with respect to fault model δ for a system with inputs I , outputs O , and specification φ . If synt_p succeeds (does not return unrealizable) for all $o_i \in O$, the resulting test suite $\text{TS} = \{\mathcal{T}_i \mid o_i \in O\}$ is guaranteed to be universally complete. However, since Theorem 1 only gives a sufficient but not a necessary condition, this procedure may fail to find a universally complete test suite, even if one exists, in general. In cases where Propositions 1 or 2 applies, it is both sound and complete, though.

Fault models In order to simplify the user input, we split the fault model δ in our coverage objective from Definition 1 into two parts: the fault kind κ and the fault frequency frq (Fig. 6). The fault kind κ is an LTL formula that is given by the user and defines *which* faults we consider. For instance, $\kappa = \neg o_i$ describes a stuck-at-0 fault, $\kappa = o_i \leftrightarrow \neg o'_i$ defines a bit-flip, and $\kappa = o'_i \leftrightarrow X o_i$ describes a delay by one time step. The fault frequency frq describes *how often* a fault of the specified kind occurs, and is chosen by our algorithm, unless it is specified by the user. We distinguish 4 fault frequencies, which we describe using temporal LTL operators.

- Fault frequency G means that the fault is permanent.

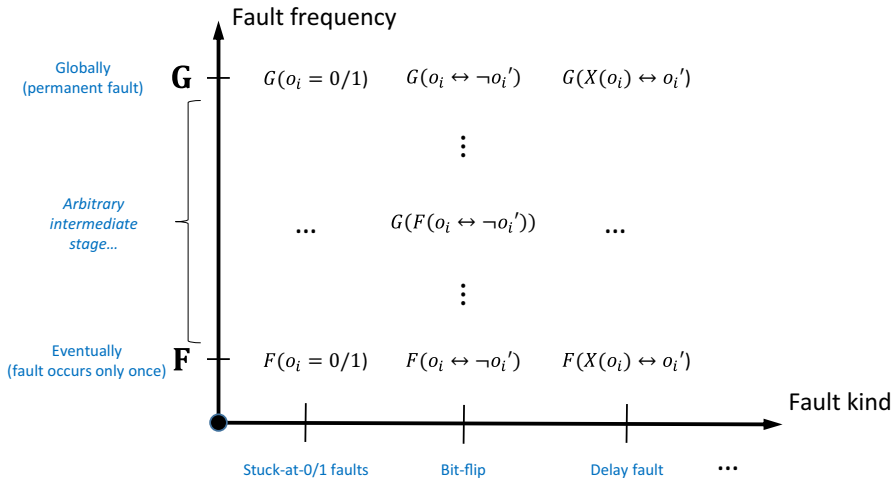


Fig. 6 Examples of different coverage objectives

Algorithm 1 SYNTLTLTEST: Synthesizes a universally complete test suite from an LTL specification for all outputs in O

```

1: procedure SYNTLTLTEST( $I, O, \varphi, \kappa$ ), returns: A set TS of test strategies
2:   TS :=  $\emptyset$ 
3:   for each  $o_i \in O$  do
4:     TS := TS  $\cup$  SYNTLTLITERATE( $I, O, \varphi, o_i, \kappa, \emptyset$ );
5:   return TS

```

- Frequency FG means that the fault occurs from some time step i on permanently. Yet, we do not make any assumptions about the precise value of i .
- Frequency GF states that the fault strikes infinitely often, but not when exactly.
- Frequency F means that the fault occurs at least once.

The fault model δ is then defined as $\delta = \text{frq}(\kappa)$. Note that there is a natural order among our 4 fault frequencies: a fault of kind κ that occurs permanently (frequency G) is just a special case of the same fault κ occurring from some point onwards (frequency FG), which is in turn a special case of κ occurring infinitely often (frequency GF), which is a special case of κ occurring at least once. Thus, a test strategy that reveals a fault that occurs at least once (without knowing when) will also reveal a fault that occurs infinitely often, etc. We say that F is the lowest and G is the highest fault frequency. In our approach, we thus compute test strategies to detect faults at the lowest frequency for which a test strategy can be found. Figure 6 presents different examples of the fault model.

Algorithm The procedure SYNTLTLTEST in Algorithm 1 formalizes our approach using the procedure SYNTLTLITERATE in Algorithm 2 as a helper. The input consists of (1) the inputs I of the SUT, (2) the outputs O of the SUT, (3) an LTL specification φ of the SUT, and (4) a fault kind κ . The result of SYNTLTLTEST is a test suite TS. The algorithm iterates over all outputs $o_i \in O$ (Line 3) and invokes the procedure SYNTLTLITERATE (Line 4). The procedure SYNTLTLITERATE then iterates over the 4 fault frequencies (Line 2), starting with the lowest one, and attempts to compute a strategy to reveal a fault (Line 3). If such a strategy

Algorithm 2 SYNTLTLITERATE: Synthesize an adaptive test strategy from an LTL specification with the lowest fault occurrence frequency

```

1: procedure SYNTLTLITERATE( $I, O, \varphi, o_i, \kappa, \Theta$ ), returns: A singleton  $\{\mathcal{T}\}$  with a test strategy  $\mathcal{T}$  on success
   or  $\emptyset$ 
2: for each frq from (F, GF, FG, G) in this order do
3:    $\mathcal{T} := \text{synt}_p(O \cup \{o'_i\}, I, (\varphi[o_i \leftarrow o'_i] \wedge \text{frq}(\kappa)) \rightarrow \neg\varphi, O, \Theta)$ 
4:   if  $\mathcal{T} \neq \text{unrealizable}$  then
5:     return  $\{\mathcal{T}\}$ ;
6: return  $\emptyset$ 
    
```

exists, it is returned to Algorithm 1 and added to TS. Otherwise, the procedure proceeds with the next higher fault frequency.

Sanity checks Note that our coverage goal in Eq. 5 is vacuously satisfied by any test suite if φ or δ is unrealizable. The reason is that the test suite must reveal every fault F realizing δ for every system S' realizing φ . If there is no such fault or system, this is trivial. As a sanity check, we thus test the (Mealy) realizability of φ and $G\kappa$ before starting Algorithm 1 (because if $G\kappa$ is realizable, then so are $FG\kappa$, $GF\kappa$ and $F\kappa$).

Handling unrealizability If, for some output, Line 3 of Algorithm 2 returns unrealizable for the highest fault frequency $\text{frq} = G$, we print a warning and suggest that the user examines these cases manually. There are two possible reasons for unrealizability. First, due to limited observability, we do not find a test strategy although one exists (see Example 1). Second, no test strategy exists because there is some $S' \models \varphi[o_i \leftarrow o'_i]$ and $F \models \delta$ such that the composition $S = S' \circ F$ (see Fig. 4) is correct, i.e., $S' \circ F \models \varphi$. In other words, for some realization, adding the fault may result in an equivalent mutant in the sense that the specification is still satisfied. For example, in case of a stuck-at-0 fault model, there may exist a realization of the specification that has the considered output $o_i \in O$ fixed to false. Such a high degree of underspecification is at least suspicious and may indicate unintended vacuities [7] in the specification φ , which should be investigated manually. If Proposition 1 or 2 applies, or if $\text{synt}(O \cup \{o'_i\}, I, (\varphi[o_i \leftarrow o'_i] \wedge G(\kappa)) \rightarrow \neg\varphi, \Theta)$ returns unrealizable, we can be sure that the second reason applies. Then, we can even compute additional diagnostic information in the form of two Mealy machines $S' \models \varphi[o_i \leftarrow o'_i]$ and $F \models \delta$ (by synthesizing some Mealy machine $S \models (\varphi[o_i \leftarrow o'_i] \wedge G(\kappa) \wedge \varphi)$ and splitting it into S' and F by stripping off different outputs). The user can then try to find inputs for $S' \circ F$ such that the resulting trace violates the specification. Failing to do so, the user will understand why no test strategy exists (see also [32]).

If the specification is as intended but no test strategy exists, we could use “collaborative” strategies. Among such strategies, we can choose one that requires as little collaboration from the adversary as necessary [19,20]. In our setting, this means that we weaken the requirement that we find the fault regardless of the implementation of the system but rather require that we find it for maximal classes of implementations. This is not unusual in testing, which is typically explorative and does not make the guarantees that we attempt to give. For instance, if the specification is $G(r \rightarrow Fg)$ with input r and output g and the fault model is $GF \neg g$, then there is no test strategy that finds this fault for all implementations. Yet, an input sequence in which r is always true is a better test sequence than one in which r is always false, because the former strategy will find the fault in some implementations, whereas the latter will not find the fault in any implementation. We leave the extension to collaborative strategies to future work.

Complexity Both $\text{synt}_p(O, I, \psi, O', \Theta)$ and $\text{synt}(O, I, \psi, \Theta)$ are 2EXPTIME complete in $|\psi|$ [33], so the execution time of Algorithm 2, and consequently also Algorithm 1, are at most doubly exponential in $|\varphi| + |\kappa|$.

Theorem 2 For a system with inputs I , outputs O , and LTL specification φ over $I \cup O$, if the fault kind κ is of the form $\kappa = \psi$ or $\kappa = (o'_i \leftrightarrow \psi)$, where ψ is an LTL formula over I and O , $\text{SYNTLTLTEST}(I, O, \varphi, \kappa)$ will return a universally complete test suite with respect to the fault model $\delta = G(\kappa)$ if such a test suite exists.

Proof Since $G(\kappa)$ implies $\text{frq}(\kappa)$ for all $\text{frq} \in \{F, GF, FG, G\}$, Theorem 1 and the guarantees of synt_p entail that the resulting test suite TS is universally complete with respect to $\delta = G(\kappa)$ if $|\text{TS}| = |O|$, i.e., if SYNTLTLTEST found a strategy for every output. It remains to be shown that $|\text{TS}| = |O|$ for $\kappa = \psi$ or $\kappa = (o'_i \leftrightarrow \psi)$ if a universally complete test suite for $\delta = G(\kappa)$ exists: either Propositions 1 or 2 states that Eq. 6 holds with $\delta = G(\kappa)$. Thus, synt_p cannot return unrealizable in SYNTLTLITERATE with $\text{frq} = G$, so $|\text{TS}|$ must be equal to $|O|$ in this case.

Theorem 2 states that SYNTLTLTEST is not only sound but also complete for many interesting fault models such as stuck-at faults or permanent bit-flips. For $\kappa = \psi$, Theorem 2 can even be strengthened to hold for all $\delta = \text{frq}(\kappa)$ with $\text{frq} \in \{F, GF, FG, G\}$.

5.3 Extensions and variants

A test suite computed by SYNTLTLTEST for specification φ and fault model δ is universally complete and detects all faults with respect to φ and δ independent of the implementation and the concrete fault manifestation if the fault manifests at one of the observable outputs as illustrated in Fig. 4.

In this section, we discuss some alternatives and extensions of our approach to improve fault coverage and performance.

User-specified fault frequencies Besides the four fault frequencies (G, FG, GF, and F), other fault frequencies (with different precedences) may be of interest, e.g., if a specific time step is of special interest. Algorithm 2 supports full LTL and thus the procedure can be extended by replacing Line 2 by “**for each** frq from Frq in this order”, where Frq is an additional parameter provided by the user.

Faults at inputs In the fault model in the previous section, we only consider faults at the outputs. However, considering SUTs that behave as if they would have read a faulty input is possible as well (by changing Line 3 in Algorithm 1 to “**for each** $o \in I \cup O$ **do**”).

Multiple faults Faults that occur simultaneously at multiple (inputs or) outputs $\{o_1, \dots, o_k\} \subseteq O$ can be considered by computing a test strategy

$$\mathcal{T} := \text{synt}_p \left(O \cup \{o'_1, \dots, o'_k\}, I, (\varphi[o_1 \leftarrow o'_1, \dots, o_k \leftarrow o'_k] \wedge \bigwedge_{i=1}^k \delta_i) \rightarrow \neg\varphi, O, \Theta \right),$$

where the fault model δ_i can be different for different outputs $o_i \in \{o_1, \dots, o_k\}$.

Faults within a SUT If a fault manifests in a *conditional fault* in a system implementation, a universally complete TS may not be able to uncover the fault (see Example 2).

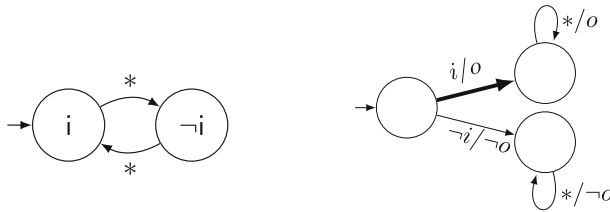


Fig. 7 Test strategy \mathcal{T}_6 and a faulty system implementation of the specification $\varphi = G((i \leftrightarrow X\neg i) \rightarrow Xo)$

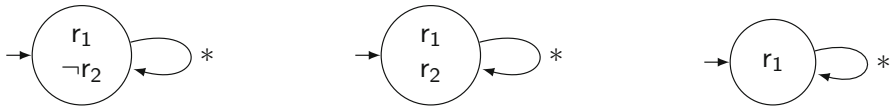


Fig. 8 Test strategy \mathcal{T}_7 on the left, \mathcal{T}_8 in the middle and \mathcal{T}_9 on the right

Example 2 Consider a system with input $I = \{i\}$, output $O = \{o\}$, and specification $\varphi = G((i \leftrightarrow X\neg i) \rightarrow Xo)$. The specification enforces o to be set to true whenever input i alternates between true and false in consecutive time steps. Consider a stuck-at-0 fault $\delta = GF \neg o$ at the output o . The test suite $TS = \{\mathcal{T}_6\}$ with the test strategy \mathcal{T}_6 illustrated in Fig. 7 (on the left) is universally complete with respect to δ . The test strategy \mathcal{T}_6 flips input i in every time step and thus forces the system to set $o = \text{true}$ in the second time step. Now consider the concrete and faulty system implementation in Fig. 7 (on the right) of φ . The test strategy \mathcal{T}_6 , when executed, first follows the bold edge and then remains forever in the same state. As a consequence, the fault in the system implementation, i.e., o stuck-at-0, is not uncovered. To uncover the fault, i has to be set to false in the initial state.

Faults within a system implementation can be considered by computing more than one test strategy for a given test objective. We extend Algorithm 1 to generate a bounded number b of test strategies by setting $\Theta = TS$ in Line 4 and enclosing the line by a **while**-loop that uses an additional integer variable c to count the number of test strategies generated per output o_i . The **while**-loop terminates if no new test strategy could be generated or if c becomes equal to b . Note that this approach is correct in the sense that all computed test strategies are universally complete with respect to the fault model $\text{frq}(\kappa)$; however, in many cases it is more efficient to determine the lowest fault frequency first in Line 4 of Algorithm 2 and then generate multiple test strategies with the same (or higher) frequency by enclosing Line 3 with the **while**-loop.

Test strategy generalization A synthesis procedure usually assigns concrete values to all variables in every state of the generated test strategy. In many cases, however, not all assignments are necessary to enforce a test objective (see Example 3).

Example 3 Consider a system with inputs $I = \{r_1, r_2\}$ and outputs $O = \{g_1, g_2\}$, which implements the specification of a two-input arbiter $\varphi = G(r_1 \rightarrow Fg_1) \wedge G(r_2 \rightarrow Fg_2) \wedge G(\neg g_1 \vee \neg g_2)$, i.e., every request r_i shall eventually be granted by setting g_i to true and there shall never be two grants at the same time. A valid test strategy \mathcal{T}_7 that tests for a stuck-at-0 fault of signal g_1 from some point in time onwards may simply set $r_1 = \text{true}$ and $r_2 = \text{false}$ all the time (see Fig. 8). This forces the system in every time step to eventually grant this one request by setting $g_1 = \text{true}$. Another valid test strategy \mathcal{T}_8 sets $r_1 = \text{true}$ and $r_2 = \text{true}$ all the time (see Fig. 8). Now the system has to grant both requests eventually. Both \mathcal{T}_7 and

Algorithm 3 GENERALIZE: Generalize a test strategy.

```

1: procedure GENERALIZE( $I, O, \varphi, o_i, \text{frq}, \kappa, \mathcal{T}$ ), returns: A generalization of  $\mathcal{T}$ 
2:   for each  $q_i \in \mathcal{T}$  do
3:     for each  $x_i \in \Sigma_I$  do
4:        $T' :=$  remove assignment to  $x_i$  from state  $q_i$  in  $\mathcal{T}$ 
5:       if  $\text{modelcheck}(T', (\varphi[o_i \leftarrow o'_i] \wedge \text{frq}(\kappa)) \rightarrow \neg\varphi)$  then
6:          $\mathcal{T} := T'$ 
7:   return  $\mathcal{T}$ 

```

\mathcal{T}_8 test for the defined stuck-at-0 fault of signal g_1 from some point in time onwards but will likely execute different paths in the SUT. Thus, considering the more general strategy \mathcal{T}_9 (see Fig. 8) that sets $r_1 = \text{true}$ all the time but puts no restrictions on the value of r_2 , allows the tester to evaluate different paths in the SUT while still testing for the defined fault class.

The procedure in Algorithm 3 generalizes a given test strategy \mathcal{T} by systematically removing variable assignments from states and employing a model-checking procedure to ensure that the generalized test strategy still enforces the same test objective. The procedure loops in Line 2 over all states of \mathcal{T} and in Line 3 over all inputs. In Line 4 the assignment to the input x_i in a state is removed such that the corresponding variable becomes non-deterministic. If the resulting test strategy still enforces the test objective, then \mathcal{T} is replaced by its generalization. Otherwise, the change is reverted. Algorithm 3 is integrated into Algorithm 2 and applied in Line 5 to generalize each generated test strategy.

Note that generalizing a test strategy is a special way of computing multiple concrete test strategies, which was discussed in the previous section. However, generalization may fail when computing multiple strategies succeeds (by following different paths).

Optimization for full observability If we restrict our perspective to the case with no partial information, i.e., all signals are fully observable, we can employ the optimization discussed in Proposition 2 to improve the performance of test strategy generation. In Line 3 of Algorithm 2 we drop a part of the assumption and simplify the synthesis step to $\mathcal{T}_i := \text{synt}(O, I, \text{frq}(\kappa) \rightarrow \neg\varphi, \Theta)$ for cases in which κ does not refer to a hidden signal o'_i . Also, for a fault model δ that describes a fault of kind $\kappa = (o'_i \leftrightarrow \psi)$, where ψ is an LTL formula over I and O , we can drop the part of the assumption according to Proposition 1 if $\text{frq} = \text{G}$. This simplifies Line 3 of Algorithm 2 to $\mathcal{T}_i := \text{synt}(O, I, \varphi[o_i \leftarrow \psi] \rightarrow \neg\varphi, \Theta)$. These simplifications, moreover, no longer require a synthesis procedure with partial information and thus, a larger set of synthesis tools is supported.

Other specification formalisms We worked out our approach for LTL, but it works for other languages if (1) the language is closed under Boolean connectives (\wedge, \neg), (2) the desired fault models are expressible, and (3) a synthesis procedure (with partial information) is available. These prerequisites do not only apply to many temporal logics but also to various kinds of automata over infinite words.

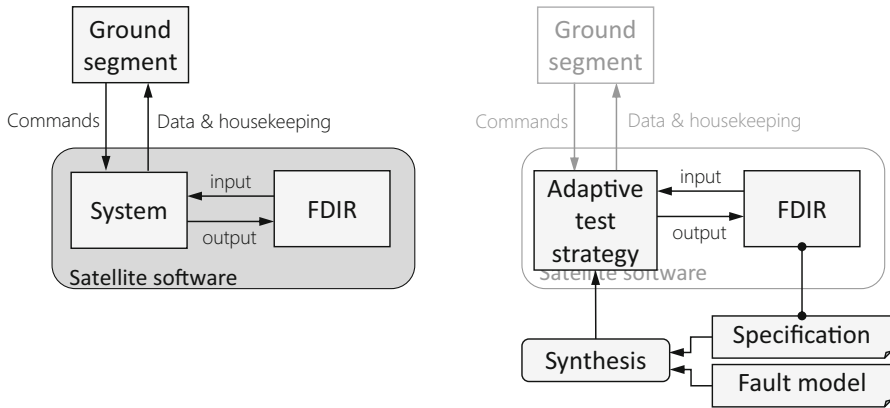


Fig. 9 FDIR in practice (left) and the intended test setup (right)

6 Case study

To evaluate our approach, we apply it in a case study on a real component of a satellite system that is currently under development. We first present the system under test and specify a version of the respective component in LTL. Using this specification, we compute a set of test strategies and evaluate them on a real implementation. Additional case studies can be found in [10].

6.1 Eu:CROPIS FDIR specification

An important task of each space and satellite system is to maintain its health state and react on failure. In modern space systems, this task is encapsulated in the *Fault Detection, Isolation, and Recovery* (FDIR) component, which collects information from all relevant sensors and on-board computers, analyzes and assesses the data in terms of correctness and health, and initiates recovery actions if necessary. The FDIR component is organized hierarchically in multiple levels [51] with the overall objective of maximizing the life-time and correct operation of the system.

In this section, we focus on system-level FDIR and present a high-level abstraction of a part of the FDIR mechanisms used in the Eu:CROPIS satellite mission as a case study for adaptive test strategy generation. On the system-level, the FDIR mechanism deals with coarse-grained anomalies of the system behavior such as erroneous sensor data or impossible combinations of signals. Likewise the recovery actions are limited to restarting certain sub-systems, switching between redundant sub-systems if available, or switching into the satellite’s safe mode. The FDIR component is highly safety- and mission-critical. If recovery on this level fails, in many cases the mission has to be considered lost.

Eu:CROPIS FDIR In Fig. 9 we illustrate where the FDIR component for the magnetic torquers of the Eu:CROPIS on-board computing system is placed in practice and in Fig. 10, we give a high-level overview of the FDIR component and its environment. The FDIR component regularly obtains housekeeping information from two redundantly-designed control units, S_1 and S_2 , which control the magnetic torquers of the satellite, and interacts with them via the electronic power system, EP. The control units S_1 and S_2 have the same functionality, but

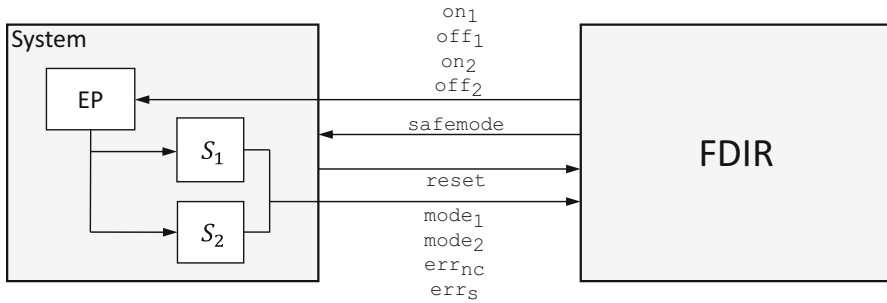


Fig. 10 High-level overview of the satellite software of Eu:CROPIS

only one of them is active at any time. The other control unit serves as a backup that can be activated if necessary. The FDIR component signals the activation (or deactivation) of a control unit to the EP which regulates the power supply.

We distinguish two types of errors, called *non-critical error* and *severe error*, signaled to the FDIR component via housekeeping information. In case of a non-critical error, two recovery actions are possible. Either the erroneous control unit is disabled for a short time and enabled afterwards again or the erroneous control unit is disabled and the redundant control unit is activated to take over its task. In case of the severe error, however, only the latter recovery action is allowed, i.e., the erroneous control unit has to be disabled and the redundant control unit has to be activated. If this happens more than once and the redundant control unit as well shows erroneous behavior, the FDIR component initiates a switch of the satellite mode into safe mode. The safe mode is a fall-back satellite mode designed to give the operators on ground the maximum amount of time to analyze and fix the problem. It is only invoked once a problem cannot be solved on-board and requires input from the operators to restore nominal operations.

LTL specification We model the specification of the FDIR component in LTL. Let $I_{FDIR} = \{\text{mode}_1, \text{mode}_2, \text{err}_{nc}, \text{err}_s, \text{reset}\}$ and $O_{FDIR} = \{\text{on}_1, \text{off}_1, \text{on}_2, \text{off}_2, \text{safemode}\}$ be the Boolean variables corresponding to the input signals and the output signals of the FDIR component, respectively.

These Boolean variables are abstractions of the real hardware/software implementation. The values of the Boolean variables are automatically extracted from the housekeeping information which is periodically collected from EP ($\text{mode}_1, \text{mode}_2$) and S_1 or S_2 ($\text{err}_{nc}, \text{err}_s$). The two error variables encompass multiple error conditions (e.g. communication timeouts, invalid responses, electrical errors like over-current or under-voltage, etc.) which are detected by the sub-system. The *reset* variable corresponds to a telecommand sent from ground to the FDIR component. For the output direction the values of the variables are used to generate commands which are sent to the EP or the satellite mode handling component. Additionally, we use the auxiliary Boolean variables $O' = \{\text{lastup}, \text{allowswitch}\}$ to model state information on specification level. These auxiliary variables do not correspond to real signals in the system, but are used as unobservable outputs of the FDIR component. In Table 1, we present a summary of the Boolean variables involved in the specification and describe their meaning.

Table 1 Descriptions of inputs and outputs of the FDIR component

Boolean variable	Description
$mode_1$	true iff S_1 is activated
$mode_2$	true iff S_2 is activated
err_{nc}	true iff a non-critical error is signaled by S_1 or S_2
err_s	true iff a severe error is signaled by S_1 or S_2
$reset$	true iff the FDIR component is reset
on_1	true iff S_1 shall be switched on
off_1	true iff S_1 shall be switched off
on_2	true iff S_2 shall be switched on
off_2	true iff S_2 shall be switched off
$safemode$	true iff the FDIR component initiates the safemode of the satellite
$lastup$	true if the last active system was S_1 and false if the last active system was S_2
$allowswitch$	true iff a switch of S_1 to S_2 or S_2 to S_1 is allowed

The LTL specification of the FDIR component is of form

$$\left(\bigwedge_{i=1}^6 A_i \right) \rightarrow \left(\bigwedge_{i=1}^{13} G_i \right)$$

and consists of the six assumptions A_1 – A_6 and the thirteen guarantees G_1 – G_{13} . All properties are listed in Table 2, expressing the following intentions:

- A_1 Whenever both systems are off, then there is no running system that can have an error. Thus, the error signals have to be low as well.
- A_2 The error signals are mutual exclusive. If the environment enforces a reset then both error signals have to be low, because we assume that ground control has taken care of the errors.
- A_3 After a reset enforced by the environment, one of the two systems has to be running and the other has to be off.
- A_4 Whenever the FDIR component sends on_1 , we assume that in the next time step system number one is running ($mode_1$) and the state of the second system ($mode_2$) does not change. The same assumption applies analogously for on_2 .
- A_5 Whenever the FDIR component sends off_1 , we assume that in the next time step system number one is off ($\neg mode_1$) and the state of the second system ($mode_2$) does not change. The same assumption applies analogously for off_2 .
- A_6 We assume that the environment, more specifically the electronic power unit, is not immediately free to change the state of the systems when there is no message from the FDIR component. It has to wait for one more time step (with no messages of the FDIR component).
- G_1 This guarantee stores which system was last activated by the FDIR component.
- G_2 We require the signals on_1 , off_1 , on_2 and off_2 to be mutually exclusively set to high.
- G_3 Whenever both systems are off, then the FDIR component eventually requests to switch on one of the systems (on_1 , on_2) or activates $safemode$ or observes a $reset$.
- G_4 We restrict the FDIR component to not enter $safemode$ as long as the component can switch to the backup system.

Table 2 Temporal specification of system-level FDIR component in LTL**Assumptions A_1 – A_6**

- A_1 $G((\neg \text{mode}_2 \wedge \neg \text{mode}_1) \rightarrow \neg \text{err}_{nc} \wedge \neg \text{err}_s)$
 A_2 $G(\neg \text{err}_{nc} \vee \neg \text{err}_s) \wedge G(\text{reset} \rightarrow \neg \text{err}_{nc} \wedge \neg \text{err}_s)$
 A_3 $G(\text{reset} \rightarrow X(\text{mode}_2 \oplus \text{mode}_1))$
 A_4 $G(\neg \text{mode}_1 \wedge \text{on}_1 \wedge \neg \text{off}_1 \wedge \neg \text{on}_2 \wedge \neg \text{off}_2 \wedge \neg \text{reset} \wedge \neg \text{safemode}) \rightarrow$
 $X\text{mode}_1 \wedge (\text{mode}_2 \leftrightarrow X\text{mode}_2)$
 $G(\neg \text{mode}_2 \wedge \neg \text{on}_1 \wedge \neg \text{off}_1 \wedge \text{on}_2 \wedge \neg \text{off}_2 \wedge \neg \text{reset} \wedge \neg \text{safemode}) \rightarrow$
 $X\text{mode}_2 \wedge (\text{mode}_1 \leftrightarrow X\text{mode}_1)$
 A_5 $G(\text{mode}_1 \wedge \neg \text{on}_1 \wedge \text{off}_1 \wedge \neg \text{on}_2 \wedge \neg \text{off}_2 \wedge \neg \text{reset} \wedge \neg \text{safemode}) \rightarrow$
 $X\neg \text{mode}_1 \wedge (\text{mode}_2 \leftrightarrow X\text{mode}_2)$
 $G(\text{mode}_2 \wedge \neg \text{on}_1 \wedge \neg \text{off}_1 \wedge \neg \text{on}_2 \wedge \text{off}_2 \wedge \neg \text{reset} \wedge \neg \text{safemode}) \rightarrow$
 $X\neg \text{mode}_2 \wedge (\text{mode}_1 \leftrightarrow X\text{mode}_1)$
 A_6 $G((\neg(\neg \text{on}_2 \wedge \neg \text{off}_1 \wedge \neg \text{on}_1 \wedge \neg \text{off}_2) \wedge X(\neg \text{on}_2 \wedge \neg \text{off}_1 \wedge \neg \text{on}_1 \wedge \neg \text{off}_2)) \wedge$
 $(\neg \text{reset} \wedge X\neg \text{reset} \wedge \neg \text{safemode} \wedge X\neg \text{safemode}) \rightarrow$
 $X((\text{mode}_2 \leftrightarrow X\text{mode}_2) \wedge (\text{mode}_1 \leftrightarrow X\text{mode}_1))$

Guarantees G_1 – G_{13}

- G_1 $G((\text{on}_1 \wedge \neg \text{on}_2) \rightarrow (X\text{lastup}))$
 $G((\neg \text{on}_1 \wedge \text{on}_2) \rightarrow (X\neg \text{lastup}))$
 $G((\neg \text{on}_1 \wedge \neg \text{on}_2) \rightarrow (\text{lastup} \leftrightarrow X\text{lastup}))$
 G_2 $G(\text{on}_1 \rightarrow \neg \text{off}_1 \wedge \neg \text{on}_2 \wedge \neg \text{off}_2)$
 $G(\text{off}_1 \rightarrow \neg \text{on}_1 \wedge \neg \text{on}_2 \wedge \neg \text{off}_2)$
 $G(\text{on}_2 \rightarrow \neg \text{on}_1 \wedge \neg \text{off}_1 \wedge \neg \text{off}_2)$
 $G(\text{off}_2 \rightarrow \neg \text{on}_1 \wedge \neg \text{on}_2 \wedge \neg \text{off}_1)$
 G_3 $G(\neg \text{mode}_2 \wedge \neg \text{mode}_1 \rightarrow F(\text{reset} \vee \text{on}_2 \vee \text{on}_1 \vee \text{safemode}))$
 G_4 $G(\text{allowswitch} \rightarrow \neg \text{safemode})$
 G_5 $G((\text{mode}_2 \vee \text{mode}_1) \rightarrow \neg \text{on}_1 \wedge \neg \text{on}_2)$
 G_6 $G(\neg \text{allowswitch} \wedge \text{lastup} \rightarrow \neg \text{on}_2)$
 $G(\neg \text{allowswitch} \wedge \neg \text{lastup} \rightarrow \neg \text{on}_1)$
 G_7 $G(\neg \text{reset} \wedge \text{allowswitch} \wedge \text{lastup} \wedge \text{on}_2 \rightarrow X\neg \text{allowswitch})$
 $G(\neg \text{reset} \wedge \text{allowswitch} \wedge \neg \text{lastup} \wedge \text{on}_1 \rightarrow X\neg \text{allowswitch})$
 G_8 $G((\text{allowswitch} \wedge \neg(((\text{lastup} \wedge \text{on}_2) \vee (\neg \text{lastup} \wedge \text{on}_1)))) \rightarrow X\text{allowswitch})$
 G_9 $G(\text{reset} \rightarrow X\text{allowswitch})$
 G_{10} $G(\text{safemode} \rightarrow (\neg \text{on}_1 \wedge \neg \text{on}_2))$
 G_{11} $G(\neg \text{allowswitch} \wedge \neg \text{reset} \rightarrow X\neg \text{allowswitch})$
 G_{12} $G((\text{err}_s \wedge \text{mode}_1 \wedge \neg \text{reset}) \rightarrow$
 $F(\text{reset} \vee \text{safemode} \vee \text{mode}_2 \vee (\text{mode}_1 \text{ U } (\text{mode}_1 \wedge \neg \text{err}_s))))$
 $G((\text{err}_s \wedge \text{mode}_2 \wedge \neg \text{reset}) \rightarrow$
 $F(\text{reset} \vee \text{safemode} \vee \text{mode}_1 \vee (\text{mode}_2 \text{ U } (\text{mode}_2 \wedge \neg \text{err}_s))))$
 G_{13} $G((\text{err}_{nc} \wedge \text{mode}_1 \wedge \neg \text{reset}) \rightarrow F(\text{reset} \vee \text{safemode} \vee \text{mode}_2 \vee (\text{mode}_1 \wedge \neg \text{err}_{nc})))$
 $G((\text{err}_{nc} \wedge \text{mode}_2 \wedge \neg \text{reset}) \rightarrow F(\text{reset} \vee \text{safemode} \vee \text{mode}_1 \vee (\text{mode}_2 \wedge \neg \text{err}_{nc})))$

- G_5 The FDIR component must not request to switch on one of the systems (on_1, on_2) as long as one of the systems is running.
- G_6 Whenever the FDIR component is not allowed anymore to switch to the backup system, then it must not request to switch the backup system on.
- G_7 Once the FDIR component switches to the backup system it is not allowed anymore to switch again (unless the environment performs a reset, see G_9).
- G_8 As long as the FDIR component only restarts the same system it is still allowed to switch in the future.
- G_9 A reset by the environment allows the FDIR component again to switch to the backup system if required.
- G_{10} Whenever the FDIR component is in `safemode` it must not request to switch-on one of the systems (on_1, on_2).
- G_{11} Once a switch is not allowed anymore and the environment does not perform a reset, then the switch is also not allowed in the next time step.
- G_{12} Whenever the FDIR component observes a server error (err_s), it must eventually switch to the backup system or activate `safemode` unless the environment performs a reset or the error disappears by itself (without restarting the system).
- G_{13} Whenever the FDIR component observes a non-critical error (err_{nc}), it must eventually switch to the backup system or activate `safemode` or the error disappears (restarting the currently running system is allowed).

6.2 Experimental results

In this section, we present experimental results for generating test strategies for the LTL specification of the Eu:CROPIS FDIR component. We first analyze runtime and memory consumption of test strategy synthesis, and then evaluate the effectiveness of the generated test strategies on a concrete implementation of the FDIR component. The proposed test strategy synthesis approach, however, is a black-box testing technique, independent of the concrete implementation and can be applied even if no implementation is available. The synthesized test strategies do not contain the test oracle; for the experiments, we use a concrete implementation, that was manually verified, as test oracle.

6.2.1 Test strategy computation

Experimental setting All experiments for computing test strategies are conducted in a virtual machine with a 64 bit Linux system using a single core of an Intel i5 CPU running at 2.60GHz. We use the synthesis procedure PARTY [31] as black-box, which implements SMT-based bounded synthesis for full LTL and, thus, we call our tool PARTYStrategy.⁶

Test strategy computation From the previously described LTL specification, we compute test strategies for the outputs on_1, off_1 and `safemode` of the FDIR component considering the fault models stuck-at-0, stuck-at-1, and bit-flip with the lowest possible fault frequencies. These are general fault assumptions and cover faults where the specification is violated with this signal being high (stuck-at-1), faults where the specification is violated with this signal being low (stuck-at-0) and faults where the specification is violated with this signal having the wrong polarity (bit-flip). We do not synthesize test strategies for the outputs on_2 and off_2 because they behave identical to on_1 and off_1 , respectively, if the role of S_1 and S_2

⁶ PARTYStrategy, <https://www.iaik.tugraz.at/content/research/scos/tools/>.

Table 3 Results for the FDIR specification. The suffix “k” multiplies by 10^3

Fault	o_i	freq	$ \mathcal{T} $	Time (s)	Peak memory (MB)
S-a-0	on_1	FG	4	1.2k	400
	off_1	FG	3	517	396
	$safemode$	FG	4	934	324
S-a-1	on_1	GF	4	438	222
	off_1	FG	4	753	378
	$safemode$	GF	3	169	192
Bit-Flip	on_1	GF	4	26k	3.6k
	off_1	FG	4	98.9k	4.3k
	$safemode$	GF	3	13.1k	4.3k

are mutually interchanged. For synthesizing test strategies, both, the bound for the maximal number of states of a test strategy and the bound for the maximal number of test strategies, are set to four. We chose the bound to be four, because for this bound there exist strategies for all our chosen fault models and output signals. The size for the maximum number of strategies per variable and fault model is set arbitrarily to four and could also be set to a different value.

In Table 3, we list the time and memory consumption for synthesizing the test strategies with our synthesis tool PARTYStrategy. The more freedom there is for implementations of the specification, the harder it becomes to compute a strategy. The search for strategies that are capable of detecting a bit-flip is the most difficult one as we cannot make use of our optimization for full observability of the output signals. For all signals with a stuck-at-0 fault and for the off_1 signal with one of the other two faults we are able to derive test strategies that can detect the fault if it is permanent from some point onwards. For the signals on_1 and $safemode$ we are able to derive strategies for stuck-at-1 faults and bit-flips also at a lower frequency, i.e., we can detect those faults also if they occur at least infinitely often.

Illustration of a computed strategy We illustrate and explain one derived strategy in detail. The strategy derived for the signal $safemode$ being stuck at 0 computed consists of four states. Figure 11 illustrates the strategy. In the first state (state 0) we have the first system running ($mode_1$) and set the err_{nc} flag, i.e., we raise a non-critical error that requires the component to restart until the error is gone or to switch to the other system. We loop in this state until the FDIR component, if it behaves according to the specification, switches off the running system. In the next state (state 1), we do not set any input and wait for the FDIR component to eventually switch on one of the systems. If the component switches on the same system, then we go back to the previous state (state 0), if it switches on the other system we go into the next state (state 3). In this state we have the second system running ($mode_2$) and set again the err_{nc} flag, i.e., we again raise a non-critical error. We loop in this state until the FDIR component reacts and, if it conforms to the specification, switches off the running system. Continuing according to the strategy we always raise a non-critical error whatever system the FDIR component activates. Eventually the FDIR component has to activate $safemode$ or violate the specification. State 2 is only entered when the FDIR violates G5. In this state, it is irrelevant how the test strategy behaves because the specification has already been violated (which is easy to detect during test execution).

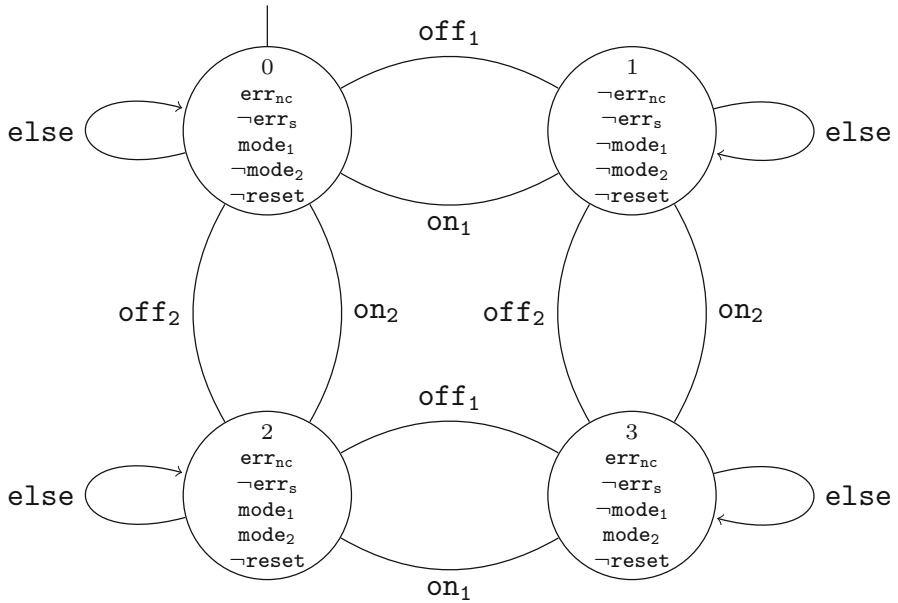


Fig. 11 Test strategy that tests for a stuck-at-0 fault of signal safemode

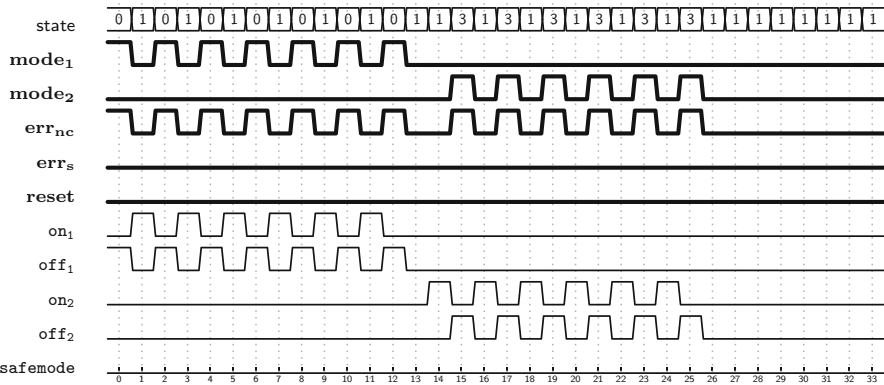


Fig. 12 Execution trace from a faulty system under the strategy that tests for a stuck-at-0 fault of signal safemode. Bold signals are controlled by the strategy

6.2.2 Test strategy evaluation

Test setting In the Eu:CROPIS satellite the FDIR component is implemented in software in the programming language C++. The implementation for the magnetic torquer FDIR handling is not an exact realization of the specification in Table 2 but extends it by allowing commands to the EP to be lost (e.g. due to electrical faults). This is accommodated by adding timeouts for the execution of the switch-on/off commands and reissuing the commands if the timeout is triggered.

The implementation is designed with testability and portability in mind and uses an abstract interface to access other sub-systems of the satellite. This allows engineers to exchange the used interface with a set of test adapters which connect to the signals generated by the test strategies. As we are only interested in the functional properties of the implementation, we can run the code on a normal Linux system, instead of the microprocessor which is used in the satellite. This gives access to all Linux based debugging and testing tools and allows us to use `gcov` to measure the line and branch coverage of the source code.

A time step of a test run consists of the following operations: request values for the input variables I_{FDIR} from the test strategy; feed the values to the test adapter from which they are read by the FDIR implementation; run the FDIR implementation for one cycle; extract the output values O_{FDIR} from the test adapter and feed them back to the test strategy to get new input values. For each time step, the execution trace—the values assigned to the inputs I_{FDIR} and outputs O_{FDIR} of the FDIR component—is recorded.

Mutation testing Besides line and branch coverage, we apply mutation analysis to assess the effectiveness, i.e., fault finding abilities, of a test suite. A test suite *kills* a mutant program M if it contains at least one test strategy that, when executed on M and the original program P , produces a trace where at least one output of M differs in at least one time step from the respective output of P (for the same input sequence). A mutant program M is *equivalent* to the original program P if M does not violate the specification. For our evaluation we manually identify and remove equivalent mutants.

We generate mutant programs of the FDIR component by systematically applying the following four mutations to each line of the C++ implementation:

1. Deletion of the line,
2. Replacement of `true` with `false` or `false` with `true`,
3. Replacement of `==` with `!=` or `!=` with `==`, and
4. Replacement of `&&` with `||` or `||` with `&&`

In total, 210 mutant programs are generated. Each having exactly one mutation. We use the GNU compiler `gcc` to remove all mutant programs, which do not compile and do not conform to the C++ programming language. We analyzed the remaining 105 mutants manually and identified 5 equivalent mutant programs, i.e., they do not violate the specification. We further correct this number by removing another 17 mutant programs related to un-specified implementation-specific behavior. Next, we executed all test strategies on the mutant programs for 80 time steps each and log the corresponding execution traces.

In Fig. 12 we illustrate the execution of the test strategy from Fig. 11 on a mutant program. This particular strategy aims at revealing a stuck-at-0 fault for signal `safemode`. The test strategy first forces the FDIR component to eventually switch to the backup system. The switch happens in time step 14 after several restarts of the system. Then the strategy forces the FDIR component to eventually activate `safemode`. However, this mutant program is faulty and instead of activating `safemode` the system remains silent from time step 26 onwards. Thus, violating guarantee $G3$.⁷

From the 83 mutant programs that violate the specification, the synthesized adaptive test strategies are able to kill 65 (78.31%). Since these test strategies are derived from requirements, without any implementation-specific knowledge, they are applicable to any system that claims to implement the specification. The mutation score of 78.31% motivate that the synthesized adaptive test strategies—although computed only for simple specific fault models—are sensitive to other faults.

⁷ Given that the user has decided that we have waited long enough for `safemode` to become true.

Table 4 Mutation coverage by fault models and signal when executing all four derived strategies

Output	Fault model			
	S-a-0 (%)	S-a-1 (%)	Bit-flip (%)	All (%)
<code>on₁</code>	67.47	53.01	8.43	75.90
<code>off₁</code>	13.25	3.61	16.87	16.87
<code>safemode</code>	61.45	13.25	13.25	16.87
All	71.08	55.42	16.87	78.31

Table 5 Overview of coverage and mutation score by testing approach

Metric	Random stimuli				Adapt. strategies	
	R(0.1k) (%)	R(1k) (%)	R(10k) (%)	R(100k) (%)	S(80) (%)	S(80)+R(10k) (%)
Line	91.5	95.7	96.8	100.0	83.0	97.9
Branch	85.4	89.6	89.6	93.8	70.8	91.7
Mutation	88.0	92.8	94.0	98.0	78.3	97.6

In Table 4, we present the mutation scores for the three signals `on1`, `off1`, and `safemode` and the three fault models stuck-at-0 (**S-a-0**), stuck-at-1 (**S-a-1**), and bit-flip (**Bit-flip**). The last column and last row show the mutation scores when considering all three fault models and all three signal, respectively.

Comparison with random testing We compare code coverage and mutation score of the synthesized adaptive test strategies and random testing when executed for 0.1k, 1k, 10k, 100k time steps. The suffix “k” multiplies by 10^3 . We use a uniform random distribution for choosing random values for all input signals, where `reset` is with 10% probability 1, and all other signals are with 50% probability 1.

The coverage and mutation scores are listed in Table 5. Coverage was measured with `gcov`. The table is built as follows: the different testing approaches are shown in the columns. The columns **R(0.1k)**, **R(1k)**, **R(10k)**, **R(100k)** refer to random testing with increasing numbers of input stimuli, and the columns **S(80)** and **S(80)+R(10k)** refer to the synthesized test strategies and the test strategies in combination with **R(10k)**.

Overall, random testing achieves high code and mutation scores when executed on the source code of the FDIR component, but can only be used if a concrete implementation of the system is available. The adaptive test strategies, on the other hand, are directly derived from the specification and independent from a concrete implementation. They can be used to derive tests if the system is still under development. Parts of the implementation which refine the specification, or which are not specified at all are not necessarily covered. The last column **S(80)+R(10k)** also shows that the synthesized test strategies improve coverage and mutation scores over **R(10k)**. Moreover, the test strategies are able to kill three mutants that are missed by all random test sequences. These mutants can only be killed when executing certain input/output sequences and it is very unlikely for random testing to hit one of them.

7 Conclusion

We have presented a new approach to compute adaptive test strategies from temporal logic specifications using reactive synthesis with partial information. The computed test strategies reveal all instances of a user-defined fault class for every realization of a given specification. Thus, they do not rely on implementation details, which is important for products that are still under development or for standards that will be implemented by multiple vendors. Our approach is sound but incomplete in general, i.e., may fail to find test strategies even if they exist. However, for many interesting cases, we showed that it is both sound and complete.

The worst-case complexity is doubly exponential in the specification size, but in our setting, the specifications are typically small. This also makes our approach an interesting application for reactive synthesis. Our experiments demonstrate that our approach can compute meaningful tests for specifications of industrial size and that the computed strategies are capable of detecting faults hidden in paths that are unlikely to be activated by random input sequences.

We have applied our approach in a case study to the fault detection, isolation and recovery component of the satellite Eu:CROPIS. The computed test suite, based only on three different types of faults, achieves a line coverage, branch coverage, and mutation score of 83.0%, 70.8%, and 78.3%, respectively, relying on information solely available from the specification. The approach also allows us to detect faults that require complex input sequences and are unlikely detected by using random testing.

Current directions for future work include improving scalability, success-rate, and usability of our approach. To this end, we are investigating using random testing for inputs in the strategies that are not fixed to single values, and best-effort strategies [19,20] for the case that there are no test strategies that can guarantee triggering the fault. Another direction for future work is research on evaluating LTL properties specified on infinite paths on finite traces to improve the evaluation process when executing the derived strategies.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF). This work was supported in part by the European Commission through the Horizon2020 project IMMORTAL (grant no. 644905) funded under H2020- EU.2.1.1.1., the FP7 project eDAS (grant no. 608770) funded under FP7-ICT, the FP7 project STANCE (grant no. 317753) funded under FP7-ICT, and by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23). We thank Ayrat Khalimov for helpful comments and assistance in using PARTY.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Acree AT, Budd TA, DeMillo RA, Lipton RJ, Sayward FG (1979) Mutation analysis. Technical report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia
2. Aichernig BK, Brandl H, Jöbstl E, Krenn W, Schlick R (2015) Killing strategies for model-based mutation testing. *Softw Test Verif Reliab* 25(8):716–748
3. Alur R, Courcoubetis C, Yannakakis M (1995) Distinguishing tests for nondeterministic and probabilistic machines. In: Leighton FT, Borodin A (eds) Proceedings of the twenty-seventh annual ACM symposium on theory of computing, 29 May–1 June 1995, Las Vegas, Nevada, USA. ACM, pp 363–372

4. Ammann P, Ding W, Xu D (2001) Using a model checker to test safety properties. In: 7th International conference on engineering of complex computer systems (ICECCS 2001), 11–13 June 2001. Sweden. IEEE Computer Society, Skövde, pp 212–221
5. Armoni R, Fix L, Flaisher A, Grumberg O, Piterman N, Tiemeyer A, Vardi MY (2003) Enhanced vacuity detection in linear temporal logic. In: Hunt WA Jr, Somenzi F (eds) Proceedings of the 15th international conference on computer aided verification, CAV 2003, Boulder, CO, USA, 8–12 July 2003, volume 2725 of lecture notes in computer science. Springer, Berlin, pp 368–380
6. Bauer A, Leucker M, Schallhart C (2011) Runtime verification for LTL and TLTL. *ACM Trans Softw Eng Methodol* 20(4):14:1–14:64
7. Beer I, Ben-David S, Eisner C, Rodeh Y (2001) Efficient detection of vacuity in temporal model checking. *Formal Methods Syst Des* 18(2):141–163
8. Blass A, Gurevich Y, Nachmanson L, Veanes M Play to test. In: Grieskamp and Weise [26], pp 32–46
9. Bloem R, Chatterjee K, Jobstmann B (2018) Graph games and reactive synthesis. In: Clarke EM, Henzinger TA, Veith H, Bloem R (eds) Handbook of model checking. Springer, Berlin, pp 921–962
10. Bloem R, Könighofer R, Pill I, Röck F (2016) Synthesizing adaptive test strategies from temporal logic specifications. In: Piskac R, Talupur M (eds) 2016 Formal methods in computer-aided design, FMCAD 2016, Mountain View, CA, USA, 3–6 Oct 2016. IEEE, pp 17–24
11. Boroday S, Petrenko A, Groz R (2007) Can a model checker generate tests for non-deterministic systems? *Electr Notes Theor Comput Sci* 190(2):3–19
12. Clarke EM, Emerson EA (1981) Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen D (ed) Logics of programs, workshop, Yorktown Heights, New York, USA, May 1981, volume 131 of lecture notes in computer science. Springer, Berlin, pp 52–71
13. David A, Larsen KG, Li S, Nielsen B (2008) A game-theoretic approach to real-time system testing. In: Sciuto D (ed) Design, automation and test in Europe, DATE 2008, Munich, Germany, March 10–14, 2008. ACM, pp 486–491
14. De Giacomo G, De Masellis R, Montali M (2014) Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: Brodley CE, Stone P (eds) Proceedings of the twenty-eighth AAAI conference on artificial intelligence, July 27–31, 2014, Québec City, Québec, Canada. AAAI Press, pp 1027–1033
15. De Giacomo G, Vardi MY (2013) Linear temporal logic and linear dynamic logic on finite traces. In: Rossi F (ed) IJCAI 2013, Proceedings of the 23rd international joint conference on artificial intelligence, Beijing, China, August 3–9, 2013. IJCAI/AAAI, pp 854–860
16. DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. *IEEE Comput* 11(4):34–41
17. Dillig I, Dillig T, McMillan KL, Aiken A (2012) Minimum satisfying assignments for SMT. In: Madhusudan P, Seshia SA (eds) Proceedings of the 24th international conference on computer aided verification—CAV 2012, Berkeley, CA, USA, July 7–13, 2012, volume 7358 of lecture notes in computer science. Springer, pp. 394–409
18. Ehlers R (2012) Symbolic bounded synthesis. *Form Methods Syst Des* 40(2):232–262
19. Faella M (2008) Best-effort strategies for losing states. CoRR [arXiv:0811.1664](https://arxiv.org/abs/0811.1664)
20. Faella M (2009) Admissible strategies in infinite games over graphs. In: Kráľovic R, Niwinski D (ed) Proceedings of the 34th international symposium on mathematical foundations of computer science 2009, MFCS 2009, Nový Smokovec, High Tatras, Slovakia, August 24–28, 2009. Volume 5734 of lecture notes in computer science. Springer, pp 307–318
21. Finkbeiner B, Schewe S (2013) Bounded synthesis. *STTT* 15(5–6):519–539
22. Fraser G, Ammann P (2008) Reachability and propagation for LTL requirements testing. In: Zhu H (ed) Proceedings of the eighth international conference on quality software, QSIQ 2008, 12–13 August 2008, Oxford, UK. IEEE Computer Society, pp 189–198
23. Fraser G, Wotawa F (2007) Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In: Proceedings of the second international conference on software engineering advances (ICSEA 2007), August 25–31, 2007, Cap Esterel, French Riviera, France. IEEE Computer Society, p 45
24. Fraser G, Wotawa F, Ammann P (2009) Issues in using model checkers for test case generation. *J Syst Softw* 82(9):1403–1418
25. Fraser G, Wotawa F, Ammann P (2009) Testing with model checkers: a survey. *Softw Test Verif Reliab* 19(3):215–261
26. Grieskamp W, Weise C (eds) (2006) Formal approaches to software testing, 5th international workshop, FATES 2005, Edinburgh, UK, July 11, 2005, revised selected papers, vol 3997. Lecture notes in computer science. Springer
27. Havelund K, Rosu G (2001) Monitoring programs using rewriting. In: 16th IEEE international conference on automated software engineering (ASE 2001), 26–29 November 2001, Coronado Island, San Diego, CA, USA. IEEE Computer Society, pp 135–143

28. Hierons RM (2006) Applying adaptive test cases to nondeterministic implementations. *Inf Process Lett* 98(2):56–60
29. Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. *IEEE Trans Softw Eng* 37(5):649–678
30. Jin HS, Ravi K, Somenzi F (2004) Fate and free will in error traces. *STTT* 6(2):102–116
31. Khalimov A, Jacobs S, Bloem R (2013) PARTY parameterized synthesis of token rings. In: Sharygina N, Veith H (eds) Proceedings of the 25th international conference on computer aided verification—CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Volume 8044 of lecture notes in computer science. Springer, pp 928–933
32. Könighofer R, Hofferek G, Bloem R (2013) Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT* 15(5–6):563–583
33. Kupfermant O, Vardit MY (2000) Synthesis with incomplete information. In: Barringer H, Fisher M, Gabbay D, Gough G (eds) Advances in temporal logic. Applied Logic Series, vol 16. Springer, Dordrecht
34. Kupferman O, Vardi MY (2003) Vacuity detection in temporal model checking. *STTT* 4(2):224–233
35. Luo G, von Bochmann G, Petrenko A (1994) Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans Softw Eng* 20(2):149–162
36. Martin DA (1975) Borel determinacy. *Ann Math* 102(2):363–371
37. Mathur AP (2008) Foundations of software testing, 2nd edn. Addison-Wesley, Boston
38. Miyase K, Kajihara S (2004) XID: don't care identification of test patterns for combinational circuits. *IEEE Trans CAD Integr Circuits Syst* 23(2):321–326
39. Morgenstern A, Gesell M, Schneider K (2012) An asymptotically correct finite path semantics for LTL. In: Björner N, Voronkov A (eds) Proceedings of the 18th international conference on logic for programming, artificial intelligence, and reasoning, LPAR-18, Mérida, Venezuela, March 11–15, 2012. Volume 7180 of lecture notes in computer science. Springer, pp 304–319
40. Nachmanson L, Veanes M, Schulte W, Tillmann N, Grieskamp W (2004) Optimal strategies for testing nondeterministic systems. In: Avrunin GS, Rothermel G (eds) Proceedings of the ACM/SIGSOFT international symposium on software testing and analysis, ISSTA 2004, Boston, MA, USA, July 11–14, 2004. ACM, pp 55–64
41. Offutt AJ (1992) Investigations of the software testing coupling effect. *ACM Trans Softw Eng Methodol* 1(1):5–20
42. Petrenko A, da Silva Simão A, Yevtushenko N (2012) Generating checking sequences for nondeterministic finite state machines. In: Antoniol G, Bertolino A, Labiche Y (eds) Fifth IEEE international conference on software testing, verification and validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012. IEEE Computer Society, pp 310–319
43. Petrenko A, Simão A (2015) Generalizing the ds-methods for testing non-deterministic fsm's. *Comput J* 58(7):1656–1672
44. Petrenko A, Yevtushenko N. Conformance tests as checking experiments for partial nondeterministic FSM. In: Grieskamp and Weise [26], pp 118–133
45. Petrenko A, Yevtushenko N (2014) Adaptive testing of nondeterministic systems with FSM. In: 15th international IEEE symposium on high-assurance systems engineering, HASE 2014, Miami Beach, FL, USA, January 9–11, 2014. IEEE Computer Society, pp 224–228
46. Pnueli A (1977) The temporal logic of programs. In: 18th annual symposium on foundations of computer science, Providence, Rhode Island, USA, 31 October–1 November 1977. IEEE Computer Society, pp 46–57
47. Pnueli A, Rosner R (1989) On the synthesis of a reactive module. In: Conference record of the sixteenth annual ACM symposium on principles of programming languages, Austin, Texas, USA, January 11–13, 1989. ACM Press, pp 179–190
48. Queille J-P, Sifakis J (1982) Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini M, Montanari U (eds) Proceedings of the international symposium on programming, 5th colloquium, Torino, Italy, April 6–8, 1982, volume 137 of lecture notes in computer science. Springer, pp 337–351
49. Tretmans J (1996) Conformance testing with labelled transition systems: implementation relations and test generation. *Comput Netw ISDN Syst* 29(1):49–79
50. Tan L, Sokolsky O, Lee I (2004) Specification-based testing with linear temporal logic. In: Zhang D, Grégoire É, DeGroot D (eds) Proceedings of the 2004 IEEE international conference on information reuse and integration, IRI—2004, November 8–10, 2004, Las Vegas Hilton, Las Vegas, NV, USA. IEEE Systems, Man, and Cybernetics Society, pp 493–498
51. Tipaldi M, Bruenjes B (2015) Survey on fault detection, isolation, and recovery strategies in the space domain. *J Aerosp Inf Syst* 12(2):235–256

52. Yannakakis M (2004) Testing, optimization, and games. In: Díaz J, Karhumäki J, Lepistö A, Sannella D (eds) Proceedings of the automata, languages and programming: 31st international colloquium, ICALP 2004, Turku, Finland, July 12–16, 2004. Volume 3142 of lecture notes in computer science. Springer, pp 28–45

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.