Abstract—Nondeterminism is used as a means of underspecification or implementation choice in specifications, and it is often necessary if part of a system or the environment is unpredictable. The use of model-checker counterexamples as test-cases is a popular technique in model-based testing. Even though model-checkers can handle nondeterministic models for verification purposes, the use of nondeterministic models for test-case generation is not directly possible. A counterexample derived from such a model is only one example execution path, although alternative paths might be possible with the same inputs. Consequently, testing could falsely identify correct implementations as erroneous. This paper describes a technique that enables the use of linear model-checker counterexamples derived from nondeterministic models as test-cases for deterministic and nondeterministic systems by applying post-processing to the counterexamples. The influence of nondeterminism on coverage measurement with model-checkers is analyzed, and known coverage criteria are adapted. This is especially useful for the execution of test-cases on nondeterministic systems, where special treatment is necessary.

A testing technique that has recently gained popularity is to derive test-cases using model-checkers, tools developed for formal verification of system specifications. Such testing techniques are very convenient; they are fully automated, offer great flexibility with regard to the type and structure of test-cases, and under certain preconditions they are very efficient.

While nondeterminism is common in protocol testing [15] with state machines, and also possible in other approaches such as testing of labeled transition systems (e.g., [11]), nondeterminism is a problem in the context of testing with model-checkers. The idea of testing with model-checkers is to use counterexamples, linear traces illustrating property violations, as test-cases. If there is nondeterminism, then a linear trace contains commitment to one particular nondeterministic choice. Applying such a test-case to an implementation that makes a different but valid choice would falsely report a fault.

This paper presents a solution that overcomes this problem. Nondeterministic choice in counterexamples is made explicit to prevent false verdicts. Extension of test-cases with alternative branches is used to fully cover nondeterministic specifications. The described methods can be applied to any known model-checker based testing technique. The descriptions given in this paper are based on NuSMV [7], a popular, freely available model-checker. The contributions of this paper are:

- A practical method to prevent false verdicts for test-cases derived from nondeterministic models with model-checker counterexamples is described.
- It is shown how linear test-cases are extended to include alternative, nondeterministic branches.
- Coverage with regard to nondeterministic behavior is discussed, and new coverage criteria are defined. Such coverage criteria are also useful to guide the execution of test-cases on nondeterministic systems.
- The techniques are evaluated on example models.

This paper is organized as follows: Section II introduces testing with model-checkers and nondeterminism in NuSMV. Then, Section III describes test-cases creation with nondeteministic models. Coverage of nondeterministic systems is discussed in Section IV. Section V shows the results of an empirical evaluation. Finally, Section VI concludes the paper.
II. Preliminaries

A. Testing with Model-Checkers

A model-checker is a tool used for formal verification. It takes as input an automaton-based model of a system and a property specified with temporal logics. Efficient techniques are then applied to examine the complete state space of the model in order to determine whether the property holds. If a violation is detected, then the model-checker returns a counterexample. Current model-checkers return linear counterexamples, which are sequences of states beginning in an initial state of the model and leading to a state such that the property violation is illustrated.

The idea of testing with model-checkers is to interpret counterexamples as test-cases. The state of a model is described by a set of variables. Therefore, each state in a counterexample can be interpreted as value assignments to all variables. The values of variables that represent the input the system receives from its environment are used as test data, and the values of output variables that are calculated by the model are used as test oracle. A test-case execution framework distinguishes between input and output values, and also interprets the counterexamples with regard to time; that is, it decides at what time the output values are measured and inputs are provided. If the observed output matches the expected values, then the verdict is pass, else it is fail. A typical kind of application for model-checker testing are reactive systems, which have a cyclic behavior such that the states of a counterexample can easily be mapped to the system behavior.

For test-case generation the model-checker is forced to create counterexamples. One way to achieve this is by formulating trap properties, which are negated claims about the model that are expected to be inconsistent with a correct model. For example, a trap property that claims that variable $x$ is never assigned value $a$ results in a trace that contains a state where $x = a$. In the literature on model-checker based testing, these trap properties are mostly based on structural coverage criteria [6, 9, 10, 17], but there are similar approaches based on mutation [5] or vacuity analysis [13].

The second category of test-case generation approaches uses fault injection to change the model such that it is inconsistent with a given specification or test-case generation specific properties. Here, the model-checker is used to examine the effects of injected faults on specification properties [2, 3, 8], or to illustrate the differences between models with injected faults and the original model [14].

B. NuSMV and Nondeterministic Models

Model-checkers are based on Kripke structures as model formalism. A Kripke structure $K$ is defined as a tuple $K = (S, S_0, T, L)$, where $S$ is the set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state. $AP$ is the countable set of atomic propositions.

When a nondeterministic model is presented with the same inputs at different times, different outputs may be generated. In automaton-based formalisms nondeterminism if often explicit, for example if there are two transitions with the same label or input action in a finite state machine (FSM). When using Kripke structures this is not the case, because there is no distinction between input and output. Consequently, counterexamples contain no indication about nondeterministic choice.

Kripke structures, however, are usually not specified explicitly but with a more intuitive description method. For example, in many model-checkers models are specified by describing the transition relations of the variables that make up a state, using logical expressions over these variables. At this level of abstraction, distinction between input and output is possible.

In this paper, we use the language of the model-checker NuSMV [7]. The transition relation of a variable is either defined in a $\text{TRANS}$ or $\text{ASSIGN}$ section. We use the $\text{ASSIGN}$ method in this paper, but as $\text{TRANS}$ is only a syntactic variation there are no limitations when applying the presented techniques to it. Listing 1 shows how such an $\text{ASSIGN}$ section looks like. The first transition described in Listing 1 is deterministic, that is, whenever condition$_1$ is encountered, $var$ is assigned $\text{next}\_\text{value}_1$ in the next state. Here, condition$_1$ can be any logical formula on the variables defined in the model.

NuSMV allows nondeterministic assignments for variables, where the assigned value is chosen out of a set expression or a numerical range. The second transition in Listing 1 is nondeterministic. Upon condition$_2$, $var$ is nondeterministically assigned either $\text{next}\_\text{value}_a$ or $\text{next}\_\text{value}_b$. With regard to the Kripke structure, each condition corresponds to a set of transitions $T'$, where for each $(s_i, s_j) \in T'$ the condition has to be fulfilled by $L(s_i)$, and the proposition $var = \text{next}\_\text{value}$ is contained in $L(s_j)$.

```
ASSIGN
next(var); := case
  condition$_1$: \text{next_value}_1;
  condition$_2$: (\text{next_value}_a, \text{next_value}_b);
...
esac;
```

Listing 1. $\text{ASSIGN}$ section of an SMV file. The transition relation of a variable $var$ is given as a set of conditions and next values. The first transition is deterministic, while the second is nondeterministic.

A model-checker verifies whether properties hold on a model. One of the supported logics for property specifications in NuSMV is future time Linear Temporal Logic (LTL) [16]. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The temporal operator "$\diamond$" refers to the next state. E.g., "$\diamond a$" expresses that $a$ has to be true in the next state. "$\mathcal{U}\$" is the until operator, where "$a \mathcal{U} b$" means that $a$ has to hold from the current state up to a state where $b$ is true. "$\square$" is the always operator, stating that a condition has to hold at all states of a trace, and "$\lozenge$" is the eventually operator that requires a certain condition to eventually hold at some time in the future.
III. Creating Test-Cases with Nondeterministic Models

If the model that is used to derive test-cases contains nondeterminism, then the direct interpretation of counterexamples as test-cases is not always possible. A linear counterexample contains a commitment to one possibility at each nondeterministic choice. If such a counterexample is used as a test-case and is executed on an implementation that makes a different but valid choice, then the execution framework reports a fault, as the expected output is not observed.

To overcome this problem, we describe a solution that consists of several steps: First, an initial set of test-cases is produced with any traditional model-checker testing technique. Then, the test-cases are extended so that nondeterministic choice can be identified. If the observed values do not match the expected values because of nondeterminism, the verdict is inconclusive instead of fail. If test-case execution leads to too many inconclusive verdicts, then the result of the execution of a set of test-cases is less expressive. Therefore, we show how test-cases can be extended such that alternative execution paths can be considered, and inconclusive results be resolved.

A. Identifying Nondeterministic Choice

A counterexample created from a NuSMV model contains no indication about which choices were made deterministically and which were made nondeterministically. While the Kripke structure does not distinguish between input and output values, this distinction is possible in the NuSMV model description.

Nondeterministic choice can highlighted with a special Boolean variable, which we will call ND_var. For each variable var that has nondeterministic transitions there is a distinct ND_var that indicates whether a nondeterministic choice was made for var. The transition relation of ND_var uses the same conditions as var, and is set to true whenever var has a nondeterministic choice, else it is set to false. The initial value of ND_var depends on whether there are any nondeterministic initializations. If there are, then ND_var is initialized with true/1, else with false/0. For the example transition relation given in Listing 1, this is illustrated in Listing 2. This annotation is straightforward and can easily be automated. It is conceivable to extend this approach such that each ND_var has one distinct value for each possible nondeterministic choice.

Any counterexample created from such an annotated model contains indications of which choices were made nondeterministically: A variable var was changed nondeterministically if ND_var is true. Consequently, if during test-case execution an observed value deviates from the expected value, the test-case execution framework only reports fail if the deviation cannot be caused by nondeterminism; else inconclusive is reported.

This approach guarantees that no valid nondeterministic choice is falsely identified as a fault. However, a test-case execution that really fails at such an execution step can also be reported as inconclusive, as the allowed alternative values are not contained in the counterexample. Inconclusive results can be verified using a model-checker. For example, assume a test-case $t := (s_0, s_x, s_y, \ldots, s_n)$ executed on an implementation that instead of $(s_x, s_y)$ takes the transition $(s_z, s_z)$. Assuming that this leads to an inconclusive verdict, the model-checker can be queried whether $(s_x, s_z)$ is a valid transition by claiming that such a transition does not exist. For example, this could be expressed as follows:

$$\Box s_x \rightarrow \neg s_y$$

Here, $s_x$ and $s_z$ represent the observed states as a logical expression (e.g., conjunction of all atomic propositions valid in that state). If this property is checked against the model, then a counterexample indicates that the transition is valid (the test-case is really inconclusive), else the transition is not valid (a fault was detected).

Even though a set of linear nondeterministic test-cases can directly be executed on an actual implementation, the result might not be satisfactory if there are too many inconclusive verdicts. Therefore, inconclusive verdicts are resolved by extending test-cases with alternative branches.

B. Extending Nondeterministic Test-Cases

The chances of reaching a pass or fail verdict instead of an inconclusive verdict with a test-case are higher, if the test-case can handle alternative execution branches caused by nondeterminism. Such a test-case can intuitively be interpreted as a tree instead of a linear sequence. As model-checkers only create linear sequences, a possibility to create tree-like test-cases is to perform iterative extension.

Whenever an inconclusive verdict is observed during test-case execution, the according test-case is extended with an alternative branch. Consequently, the number of inconclusive results is iteratively reduced by extending and re-executing inconclusive test-cases, until the execution is conclusive.

Test-case extension with a model-checker is quite simple. When a test-case is first created, it is the result of checking a model against a property. The same model and property are used to extend the test-case, but the initial state of the model is set to the state that caused the inconclusive result. For example, if test-case $t := (s_0, \ldots, s_x, s_y, \ldots, s_n)$ is inconclusive because the implementation takes $(s_x, s_z)$ instead of $(s_x, s_y)$, then the initial state of the model is set to $s_z$. In NuSMV this can be done using $\text{init}(\text{var}) := \text{value}$ expressions for all variables, where value is the value of var in $s_z$.

Listing 2. Nondeterministic choice is marked with a dedicated variable. The conditions equal those of the variable var that is observed with regard to nondeterministic choice.
A counterexample created from this new model and property fulfills the same purpose as the original test-case (i.e., it shows the same property violation), but begins in \(s_z\). Consequently, it can be used as an alternative branch of the original test-case \(t\).

IV. COVERAGE OF NONDETERMINISTIC SYSTEMS

Test-cases can easily be extended using the presented techniques, but even if the execution of a test-suite results in no inconclusive verdicts this does not guarantee thorough testing. A weak test-suite might only explore a small subset of the possible behavior and avoid nondeterministic transitions. Therefore, this section considers coverage measurement for nondeterministic systems.

The use of nondeterminism does not only have an influence on the test-case generation, but also on coverage measurement. In general, coverage criteria are used to evaluate how well certain aspects of a system are exercised by a test-suite. For example, transition coverage measures how many of a system’s transitions have been executed.

If a model has nondeterministic transitions, this does not automatically mean that an implementation under test (IUT) has to implement all possible transitions. Therefore, when testing a deterministic implementation, coverage of all possible nondeterministic transitions might not be possible.

In contrast, if the IUT is nondeterministic there is no guarantee that the behavior observed during testing is the same that will occur at runtime. It is therefore common to execute test-cases repeatedly to increase certainty that the majority of possible behavior has been observed. In this case, it is advantageous to include all possible outcomes of a nondeterministic choice in the coverage criterion.

Coverage analysis with model-checkers represents test-cases as models by adding a special state counter variable, and setting all other variables depending only on the value of this state counter [1]. Normally, coverage of test-suites created with model-checkers can be measured without an implementation. Coverage of nondeterministic systems is not measured on the test-suite itself but on execution traces created by the test-case execution, because even if a test-case can cover a nondeterministic transition, there is no guarantee that the IUT also does so. The execution traces can be represented as verifiable models just like test-cases, as described by Ammann and Black [1]: in the model, the values of all variables of the trace are set according to a special state counter variable.

We consider the coverage criteria defined by Rayadurgam and Heimdahl [17]. The model is interpreted as a transition system \(M = (D, \Delta, \rho)\), where \(D\) represents the state space, \(\Delta\) represents the transition relation and \(\rho\) characterizes the initial system state. A transition is defined as a tuple of logical predicates \((\alpha, \beta, \gamma)\), specifying pre-state, post-state, and guard, respectively. A NuSMV model is also a transition system, where conditions in the NuSMV source represent guard predicates. To allow for nondeterministic transitions, we assume that there are multiple different post-states, and extend the definition of a transition to \((\alpha, \beta, \gamma, B)\), where \(B\) is a set of predicates describing possible post-states.

\[ \begin{align*}
\alpha &\land \beta(s_i, s_{i+1}) \land \gamma(s_i, s_{i+1}) \text{ holds.}
\end{align*} \]

Here, \(s_i\) denotes the \(i\)-th state of test-case \(s\).

When testing with model-checkers, a coverage criterion can be represented as a set of trap properties. These trap properties can either be used to create a test-suite that satisfies the coverage criterion, or to measure the coverage of a given test-suite. For example, consider the NuSMV transition description given in Listing 1. For variable \(\text{var}\), there is one transition to \(\text{next}_1\) upon \(\text{condition}_1\). This transition can be represented as a single trap property, which claims that upon \(\text{condition}_1\), \(\text{var}\) never equals \(\text{next}_1\) in the next state:

\[ \Box \text{condition}_1 \rightarrow \neg(\text{var} = \text{next}_1) \]

Checking such a trap property against a model results in a counterexample that takes the transition described by this property, and model-checking test-cases against this trap property results in a counterexample if the transition is taken by a test-case. The idea of formulating test-cases as NuSMV models and then model-checking these against trap properties is described in detail in [1].

The second condition in Listing 1 contains a nondeterministic choice. Upon \(\text{condition}_2\), \(\text{var}\) can either be assigned \(\text{next}_a\) or \(\text{next}_b\). Accordingly, we can define different versions of simple transition coverage:

**Deterministic Simple Transition Coverage** assumes a deterministic implementation. Therefore, if there is a nondeterministic transition in the model, this transition is covered if any of the possible transitions is taken. With regard to the transition model, this requires for each transition \((\alpha, \beta, \gamma)\) that there exists a test-case \(s\) such that for some \(i\):

\[ \exists \beta \in B : \alpha(s_i) \land \beta(s_i, s_{i+1}) \land \gamma(s_i, s_{i+1}) \]

\(s_i\) denotes the \(i\)-th state of test-case \(s\). For Listing 1, this results in the following trap properties:

\[ \Box \text{condition}_1 \rightarrow \neg(\text{var} = \text{next}_1) \]

\[ \Box \text{condition}_2 \rightarrow \neg(\text{var} = \text{next}_a) \lor \neg(\text{var} = \text{next}_b) \]

**Nondeterministic Simple Transition Coverage** measures coverage of only the nondeterministic transitions. It considers transitions \((\alpha, \beta, \gamma)\) with \(|B| > 1\). For Listing 1, this results in the following trap properties:

\[ \Box \text{condition}_2 \rightarrow \neg(\text{var} = \text{next}_a) \]

\[ \Box \text{condition}_2 \rightarrow \neg(\text{var} = \text{next}_b) \]

If the IUT is nondeterministic, then testing has to be repeated until the tester is confident that all possible behaviors have been observed. Nondeterministic coverage can be used to guide such a decision. For example, as a minimal criterion, testing might be continued at least until all nondeterministic transitions have been fully covered.
Full Simple Transition Coverage is based on all possible transitions; i.e., if a transition is nondeterministic, all possible outcomes are included. It is a combination of deterministic and nondeterministic complete transition coverage. With regard to the transition model, this requires for each transition \((\alpha, B, \gamma)\) that there exists a test-case \(s\) for every \(\beta \in B\), such that for some \(i\):

\[ \alpha(s_i) \land \beta(s_i, s_{i+1}) \land \gamma(s_i, s_{i+1}) \]

For Listing 1, this results in the following trap properties:

- \(\Box\text{condition}_1 \rightarrow \neg(\text{var} = \text{next\_value}_1)\)
- \(\Box\text{condition}_2 \rightarrow \neg(\text{var} = \text{next\_value}_a)\)
- \(\Box\text{condition}_2 \rightarrow \neg(\text{var} = \text{next\_value}_b)\)

b) Simple Guard Coverage: Simple guard coverage extends simple transition coverage such that a transition is covered, if there is a test-case where it is taken, and one where it is not taken. Similarly to the versions of transition coverage, the original definition of simple guard coverage in [17] can be extended to the following:

**Deterministic Simple Guard Coverage** requires for each transition \((\alpha, B, \gamma)\) that there exist test-cases \(s\) and \(t\) such that for some \(i\) and \(j\):

\[ \exists \beta_1 \in B : \quad \alpha(s_i) \land \beta_1(s_i, s_{i+1}) \land \gamma(s_i, s_{i+1}) \]

\[ \exists \beta_2 \in B : \quad \alpha(t_j) \land \neg \beta_2(t_j, t_{j+1}) \land \neg \gamma(t_j, t_{j+1}) \]

**Nondeterministic Simple Guard Coverage** requires for each nondeterministic transition \((\alpha, B, \gamma)\) where \(|B| > 1\) that there exist test-cases \(s\) and \(t\) for every \(\beta \in B\), such that for some \(i\) and \(j\):

\[ \alpha(s_i) \land \beta(s_i, s_{i+1}) \land \gamma(s_i, s_{i+1}) \]

\[ \alpha(t_j) \land \neg \beta(t_j, t_{j+1}) \land \neg \gamma(t_j, t_{j+1}) \]

Full simple guard coverage can be defined as a combination of deterministic and nondeterministic simple guard coverage.

c) Complete Guard Coverage: Complete guard coverage is similar to the multiple condition coverage criterion applied to source code. A guard or condition consists of one or more clauses. Complete guard coverage requires all possible combinations of truth values of the guard clauses to be tested.

**Deterministic Complete Guard Coverage** requires for each transition \((\alpha, B, \gamma)\), where \(\gamma\) consists of clauses \(\{c_1, ..., c_l\}\), that there is a test-case \(t\) for any given boolean vector \(u\) of length \(l\), such that for some \(i\):

\[ \exists \beta \in B : \quad \bigwedge_{k=1}^{l} (c_k(t_i, t_{i+1}) = u_k) \]

**Nondeterministic Complete Guard Coverage** requires for each transition \((\alpha, B, \gamma)\), where \(|B| > 1\) and \(\gamma = \{c_1, ..., c_l\}\), that there is a test-case \(t\) for every \(\beta \in B\) and any given boolean vector \(u\) of length \(l\), such that for some \(i\):

\[ \bigwedge_{k=1}^{l} (c_k(t_i, t_{i+1}) = u_k) \]

Full complete guard coverage can be defined as a combination of deterministic and nondeterministic complete guard coverage.

d) Clause-wise Guard Coverage: Clause-wise guard coverage is a variant of MC/DC. The definition given in [17] can be adapted similarly to the previous criteria, and is therefore omitted because of space limitations.

V. EXPERIMENTAL RESULTS

This section presents the results of an empirical evaluation using two example NuSMV models. The Safety Injection System (SIS) example was introduced in [4] and has previously been used for testing research (e.g., [9]). Cruise Control (CC) is based on a version in [12], and has also been used several times for automated test-case generation, e.g., [2], [3].

In their original versions, the example models are deterministic. Due to space limitations we refer to [4] and [12] for details. We modified both models to be nondeterministic with regard to overriding; i.e., the output is chosen nondeterministically in most cases when overriding is activated.

The deterministic models are used to create sets of trap properties for test-case generation for different common criteria: *State* coverage requires each variable to take all its values, *Transition* coverage requires all transitions described in the NuSMV model to be taken, *Condition* coverage tests the effects of atomic propositions within transition conditions, and *Transition-Pair* coverage requires all possible pairs of transitions described in the NuSMV model to be taken. Finally, *Reflection* describes a set of trap properties that is created by reflecting the transition relation as properties, and then applying various mutation operators to these properties [5].
Test-cases are generated using the trap properties and the nondeterministic models. The models are implemented in Python as both deterministic and nondeterministic versions, in order to experiment with test-case extension and execution.

The results of the test-case generation are listed in Tables I and II. Values for nondeterministic IUTs are averaged over 10 runs. The number of inconclusive verdicts is given as a percentage of the size of the initial test-suite, and the number of iterations necessary to resolve all inconclusive verdicts is also given. State coverage is a weak criterion and leads to very short test-cases, which resulted in no inconclusive verdicts. Transition-pair coverage creates the largest test-cases of the considered trap properties, and therefore nondeterministic transitions occur more often than with the other criteria. With the exception of transition-pair coverage test-suites, the number of inconclusive verdicts is relatively small. The number of iterations necessary to remove all inconclusive verdicts is slightly larger for a nondeterministic implementation, but it is still small enough to make the approach feasible.

Table III lists the coverage values determined after executing the initial test-suites on nondeterministic implementations; the values are again averaged over 10 runs. Simple guard coverage was chosen as an example criterion. If the IUT is deterministic, then deterministic coverage is a more realistic criterion, while full coverage might not be achievable. Deterministic coverage is achieved rather quickly with a nondeterministic IUT, therefore nondeterministic coverage is better suited as an indicator whether a nondeterministic IUT needs more testing iterations.

VI. Conclusions

In this paper, we have presented an extension to model-checker based techniques for test-case generation and coverage measurement that allows the use of nondeterministic models and implementations. Current techniques are limited to deterministic models and deterministic systems, because the counterexamples that are used as test-cases are deterministic, even if the corresponding model is nondeterministic. Using NuSMV as an example language, a straightforward rewriting to add information about nondeterministic choice in counterexamples was presented. This allows to distinguish between test-cases that fail because of errors, and test-cases that are inconclusive because an alternative nondeterministic path was chosen. It was shown how in case of inconclusive results linear test-cases can be extended to tree-like test-cases that can cope with alternative branches. Coverage criteria based on nondeterministic models and coverage measurement of nondeterministic systems were discussed, and several known coverage criteria were adapted to a nondeterministic setting.

The presented methods apply to both deterministic and nondeterministic implementations, and some of the coverage criteria are especially useful when testing nondeterministic implementations. The applicability depends on the amount of nondeterminism in the system that is to be tested. Asynchronous, distributed systems are likely to cause too many inconclusive results in order for the methods to be feasible. Therefore, the intended application domains include nondeterminism as a means of underspecification or implementation choice, and limited nondeterminism in the IUT.

Current model-checkers do not indicate nondeterministic choice in their counterexamples. It is, however, conceivable to extend counterexample generation algorithms such that indicators are included automatically, avoiding the need to rewrite the model.

References