

Nondeterministic Testing with Linear Model-Checker Counterexamples

(Research Paper)

Gordon Fraser and Franz Wotawa*
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria
Email: {fraser,wotawa}@ist.tugraz.at

Abstract

In model-based testing, software test-cases are derived from a formal specification. A popular technique is to use traces created by a model-checker as test-cases. This approach is fully automated and flexible with regard to the structure and type of test-cases. Nondeterministic models, however, pose a problem to testing with model-checkers. Even though a model-checker is able to cope with nondeterminism, the traces it returns make commitments at nondeterministic transitions. If a resulting test-case is executed on an implementation that takes a different, valid transition at such a nondeterministic choice, then the test-case would erroneously detect a fault. This paper discusses the extension of available model-checker based test-case generation methods so that the problem of nondeterminism can be overcome.

Keywords: Automated test-case generation, nondeterministic systems, testing with model-checkers

Relevant Topics: Software testing, model checking, software verification

1. Introduction

Nondeterminism is used in abstract models and specifications to represent implementation choice. A system is nondeterministic if given the same inputs at different times, different outputs can be produced. While nondeterminism is not problematic for the modeling of a system, it does make testing of the system more complicated.

*This work has been partially supported by the Austrian Federal Ministry of Economics and Labour, the competence network Softnet Austria (<http://www.soft-net.at>), and the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-809446.

The use of model-checkers for test-case generation has recently gained popularity. Model-checker based techniques are fully automated, flexible, and under certain preconditions very efficient. They usually force the model-checker to create counterexamples to negated claims. When it comes to testing nondeterministic systems, however, model-checker based methods have not been an option so far. Current model-checkers create linear sequences of states as counterexamples. Test data and expected results are extracted for each state of such a sequence. Linear sequences created from nondeterministic models cannot be used directly for testing, as they can falsely identify valid nondeterministic choices as incorrect.

While model-checkers only create linear sequences as counterexamples, they do generally support the modeling of nondeterministic systems. The aim of this paper therefore is to make use of nondeterministic models and create test-cases with model-checkers that can be executed on corresponding implementations.

A common approach to software testing of nondeterministic systems is to create such test-cases that recognize different solution paths that can occur due to nondeterministic choices (e.g., [11]). If an implementation deviates from a test-case because of a nondeterministic choice, then the conclusion that can be drawn from the execution is neither that the implementation failed or passed the test-case, but that the result is inconclusive.

In [8], we proposed a simple method that allows to use regular model-checker counterexamples to be used as test-cases, and discussed coverage analysis of nondeterministic systems. In this paper, we extend the previous work with a detailed formal background and by analyzing the details of how nondeterministic test-cases can be extended with alternative branches when encountering an inconclusive result.

This paper is organized as follows: First, Section 2 presents the necessary theoretical background of testing with model-checkers and nondeterministic systems in general. The proposed solution is described in Section 3, and

Section 4 shows how it can be implemented with currently available model-checking tools. Section 5 introduces a non-deterministic example application, and describes the setup and results of an evaluation of the presented method. The results are discussed in Section 6. Finally, the paper is concluded with an outlook in Section 7.

2. Nondeterministic Testing with Model-Checkers

This section first describes the necessary theoretical background of testing with model-checkers. Then, nondeterminism is introduced into these concepts.

2.1. Preliminaries

A model-checker is a tool that determines whether temporal logic properties hold on models. Model-checkers are based on Kripke structures as model formalism:

Definition 1 (Kripke Structure) *A Kripke structure K is 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.*

Common model-checkers return a linear trace as a counterexample when they determine that a model K violates a property ϕ . The counterexample is a sequence of states beginning in the initial state, and illustrates how the property is violated. It is a finite prefix of an execution sequence (path) of the model:

Definition 2 (Path) *A path $\pi := \langle s_0, s_1, \dots \rangle$ of Kripke structure K is a finite or infinite sequence such that $\forall i \geq 0 : (s_i, s_{i+1}) \in T$ for K .*

Properties are specified using temporal logics. In this paper, we use future time Linear Temporal Logic (LTL) [16]. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The temporal operator " \bigcirc " refers to the *next* state. E.g., " $\bigcirc a$ " 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 " \diamond " is the *eventually* operator that requires a certain condition to eventually hold at some time in the future. The syntax of LTL is given as follows, with $a \in AP$:

$$\phi ::= a \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \mathcal{U} \phi_2 \mid \bigcirc\phi \mid \square\phi \mid \diamond\phi$$

Other Boolean operators can be defined using \vee and \neg . The semantics of LTL is expressed for infinite traces of a

Kripke structure, where $K, \pi \models \phi$ means that trace π of Kripke structure K satisfies the LTL formula ϕ . π_i denotes the suffix of the trace π starting from the i -th state, and $\pi(i)$ denotes the i -th state of the trace π , with $i \in \mathbb{N}_0$. The initial state of a trace is $\pi(0)$. The notation $K \models \phi$ is used as an abbreviation to denote that there exists a path π such that $K, \pi \models \phi$.

$$K, \pi \models a \quad \text{iff} \quad a \in L(\pi(0)) \quad (1)$$

$$K, \pi \models \neg\phi \quad \text{iff} \quad K, \pi \not\models \phi \quad (2)$$

$$K, \pi \models \phi_1 \vee \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \vee K, \pi \models \phi_2 \quad (3)$$

$$K, \pi \models \phi_1 \mathcal{U} \phi_2 \quad \text{iff} \quad \exists i \in \mathbb{N}_0 : K, \pi_i \models \phi_2 \wedge \forall 0 \leq j < i : K, \pi_j \models \phi_1 \quad (4)$$

$$K, \pi \models \bigcirc\phi \quad \text{iff} \quad K, \pi_1 \models \phi \quad (5)$$

$$K, \pi \models \square\phi \quad \text{iff} \quad \forall j \in \mathbb{N}_0 : K, \pi_j \models \phi \quad (6)$$

$$K, \pi \models \diamond\phi \quad \text{iff} \quad \exists j \in \mathbb{N}_0 : K, \pi_j \models \phi \quad (7)$$

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. In the literature on model-checker based testing, these trap properties are mostly based on structural coverage criteria [5, 9, 10, 17], but there are similar approaches based on mutation [4] or vacuity analysis [13].

A trap property ϕ is supposed to be violated by a correct model, resulting in a trace t that is usable as a test-case. For example, in order to create a state coverage test-suite a trap property for each possible state a of every variable x is needed, claiming that the value is not taken: $\square \neg(x = a)$. A counterexample to such an example trap property is any trace that contains a state where $x = a$.

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 [1, 2, 7, 14]. Here, the model-checker is used to examine the effects of injected faults on specification properties, or to illustrate the differences between models with injected faults and the original model.

Regardless of which approach is used, a counterexample is always the result of model-checking a pair of a model K and a property ϕ . Therefore, we define the requirement of a test-case as the pair (K, ϕ) :

Definition 3 (Test Requirement) *A test requirement is a tuple $R_T = (K, \phi)$, where K is a Kripke structure, and ϕ is a property.*

For a given test requirement $R_T = (K, \phi)$, the property ϕ is supposed to be violated by the Kripke structure K , such that model-checking produces a counterexample c ($K, c \not\models \phi$) that can be used for testing.

Counterexamples resulting from test requirements are used as test-cases. A test-case t is a finite prefix of a path π .

It consists of the test data and the expected output. The set of test-cases resulting from model-checking all test requirements is a *test-suite*. Test-cases created by model-checkers are deterministic, therefore they cannot handle nondeterministic behavior in their basic form.

Definition 4 (Linear Test-Case) A linear test-case t is a finite prefix of a path π of Kripke structure K .

Each state of a linear test-case consists of value assignments for all variables in the model. When executing a test-case created with a model-checker, the test execution framework has to distinguish between input and output variables. Any variable or atomic proposition provided by the environment and not calculated by the system is considered as input. The system response is referred to as its output. The input values of a counterexample are used as test data, and the output values of the counterexample represent the expected values. The system under test is exercised with the test data, and the resulting outputs are compared with the expected values. If the expected values are correct, then the implementation *passes* the test-case, else it *fails*.

2.2. Nondeterministic Testing

While test-case creation with model-checkers assumes deterministic models, Kripke structures can be used to describe both deterministic and nondeterministic systems. When a nondeterministic model is presented with the same inputs at different times, different outputs may be generated. In automaton-based formalisms nondeterminism is often explicit, for example if there are two transitions with the same label or input action in a finite state machine (FSM). In Kripke structures this is not the case, because there is no distinction between input and output.

Therefore, we interpret the Kripke structure model using the formal framework described by Rayadurgam and Heimdahl [17]. In this framework, the system state is uniquely determined by the values of n variables $V = \{x_1, x_2, \dots, x_n\}$. Each variable x_i has a domain D_i , and consequently the reachable state space of a system is a subset of $D = D_1 \times D_2 \times \dots \times D_n$. The state of the system is described with a state vector $\vec{s} = (x_1, x_2, \dots, x_n)$, which represents a valuation of the n variables. The set of variables V can be partitioned into input, output, and state (internal) variables V_I , V_O , and V_S , respectively. Accordingly, D can be partitioned into D_I , D_O , and D_S to denote the domains of the input, output, and state variables, respectively. The set of initial values for the variables is defined by a logical expression ρ . The valid transitions between states are described by the transition relation, which is a subset of $D \times D$. The transition relation is defined separately for each variable using logical conditions. For variable x_i , the condition $\alpha_{i,j}$ defines the possible pre-states of the j -th transition, and $\beta_{i,j}$ is the j -th post-state condition. A simple transition

for a variable x_i is a conjunction of $\alpha_{i,j}$, $\beta_{i,j}$ and a guard condition $\gamma_{i,j}$: $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$.

The disjunction of all simple transitions for a variable x_i is a complete transition δ_i . The transition relation Δ is the conjunction of the complete transitions of all the variables $\{x_1, \dots, x_n\}$. Consequently, a basic transition system is defined as follows:

Definition 5 (Transition system) A transition system M_T over variables $\{x_1 \dots x_n\}$ is a tuple $M_T = (D, \Delta, \rho)$, with $D = D_1 \times D_2 \times \dots \times D_n$, $\Delta = \bigwedge_i \delta_i$, and the initial state expression ρ .

A transition system $M_T = (D, \Delta, \rho)$ is nondeterministic if there exists states s , s' , and s'' , such that there are transitions $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$ and $\delta_{i,k} = \alpha_{i,k} \wedge \beta_{i,k} \wedge \gamma_{i,k}$ for variable x_i , such that $\alpha_{i,j}(s) \wedge \alpha_{i,k}(s)$, and $\gamma_{i,j}(s, s') \wedge \gamma_{i,k}(s, s'')$, but $\neg(\beta_{i,j}(s, s') \wedge \beta_{i,k}(s, s''))$. That is, if at the same state the guard conditions of two different transitions are enabled, but the post-state predicates differ. A counterexample created from a nondeterministic transition system represents a choice made at nondeterministic transitions, because a trace can only have either (s, s') or (s, s'') at one step in the sequence.

The normal, deterministic execution mode of model-checker generated test-cases can fail in the context of nondeterministic systems. If the counterexample contains the transition (s, s') at one step, but the implementation responds with (s, s'') , then the common approach to model-checker based testing would declare the test to have failed. If, however, (s, s'') is a valid transition, then the verdict should in fact be *inconclusive*. Therefore, we extend the definition of linear test-cases to linear nondeterministic test-cases.

Definition 6 (Linear Nondeterministic Test-Case) A linear nondeterministic test-case t_{NL} for transition system $M_T = (D, \Delta, \rho)$ is a tuple $t_{NL} = (t, R)$, where t is a finite sequence $t := \langle s_0, s_1, \dots, s_n \rangle$, and $R = D_O \times \mathbb{N} \rightarrow \{\text{pass, fail, inconclusive}\}$ maps observed outputs and positions within the test-case to the verdicts pass, fail and inconclusive.

A linear nondeterministic test-case extends a regular linear test-case with a function that maps observations during execution with the verdicts *pass*, *fail* and *inconclusive*. During test-case execution the outputs of the system under test are observed. Let O denote the observed values at the i -th state; then $R(O, i)$ evaluates to *pass*, iff the observed outputs are those expected by the linear test-case. If O differs from the expected values because of a nondeterministic choice, then the verdict is *inconclusive*. For all other cases, the verdict is *fail*. Note that the execution of a linear nondeterministic test-case can be inconclusive even though the implementation is erroneous, if the deviation is only observable in a nondeterministic transition.

The use of linear nondeterministic test-cases instead of deterministic test-cases achieves that testing is sound: No correct implementation is rejected.

If the execution of a test-suite on an implementation results in too many inconclusive verdicts, then this reduces the effectiveness of testing. Consequently, instead of only returning an inconclusive verdict, a test-case should also be able to interpret alternative paths that occur as a consequence of nondeterministic decisions. A test-case could therefore be seen as a tree-like structure of alternative execution paths instead of a linear trace. We define nondeterministic test-cases in the style of Hierons’s adaptive test-cases [11]. The set \mathcal{T} denotes the domain of nondeterministic test-cases.

Definition 7 (Nondeterministic Test-Case) A *nondeterministic test-case* $t_{ND} \in \mathcal{T}$ is recursively defined as one of the following:

- pass
- fail
- inconclusive
- (s, R) , where $s \in S$, and $R = S \times \mathcal{T}$ is a function that maps states to test-cases.

Intuitively, a nondeterministic test-case can be interpreted as a tree, where leaf nodes are either *pass*, *fail* or *inconclusive*, and all other nodes contain states. A nondeterministic test-case is fully expanded if it contains no inconclusive verdicts. A test-case that is not fully expanded can easily be extended by adding a new branch to an inconclusive node. A linear nondeterministic test-case is a special case of a nondeterministic test-case, where every node except the final state has exactly one child node. Nondeterministic test-cases are executed similarly to linear nondeterministic test-cases, except that there are alternatives for the next input at nondeterministic branches, depending on the observed outputs. Execution stops, when a *pass*, *fail*, or *inconclusive* node is reached.

Nondeterministic test-cases can be extended at nondeterministic branching points. Even if there are several branches, a deviation at the state of these branches is still inconclusive, unless the branch is fully expanded; that is, if the branch contains all possible alternatives.

3. Deriving Test-Cases from Nondeterministic Models

This section describes how model-checkers can be used to create nondeterministic test-cases. Rather than inventing a completely new approach to test-case generation, the presented method is an extension to currently used techniques.

The normal approach to test-case generation with model-checkers is to create a set of test requirements, and then to

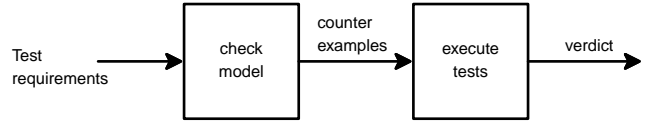


Figure 1. Traditional test-case generation with model-checkers.

call the model-checker for each of these test requirements. The result is a set of counterexamples, which are used as test-cases. This is illustrated in Figure 1.

If this approach is applied to a nondeterministic model, then the model-checker makes decisions at the nondeterministic transitions. Resulting counterexamples are linear sequences representing these choices, and can as such only be used as deterministic test-cases. However, there is no guarantee that an implementation will behave identically as described by the test-case.

Therefore, the standard approach is extended. Figure 2 illustrates an approach that can be used to create nondeterministic test-cases. In detail, the steps are as follows:

1. **Create counterexamples:** The first step is identical to normal test-case generation. A set of test requirements is created, each consisting of a model and a property. By calling the model-checker on each of the test requirements, a set of linear counterexamples is created.
2. **Create linear nondeterministic test-cases:** Using information about the nondeterministic choices in the model, the linear counterexamples are enhanced to linear nondeterministic test-cases.
3. **Execute test-cases:** The resulting test-cases are executed. Whenever an inconclusive verdict is encountered, the test requirement and the state causing the verdict are recorded.
4. **Create new test requirements:** For all inconclusive results, new test requirements are created from the original, unfulfilled test requirements. These test requirements consist of a model, where the initial state is set to the state that caused the inconclusive result, and the original property.
5. **Enhance test-cases:** The model-checker is called on the new test requirements. Then, each linear sequence is used to extend the corresponding original nondeterministic test-case. This is achieved by attaching the new test-case at the position of the nondeterministic choice in the nondeterministic test-case.
6. **Execute extended test-cases:** The extended test-cases are executed. If inconclusive verdicts are encountered, then the method proceeds with step 4, until no more inconclusive verdicts remain.

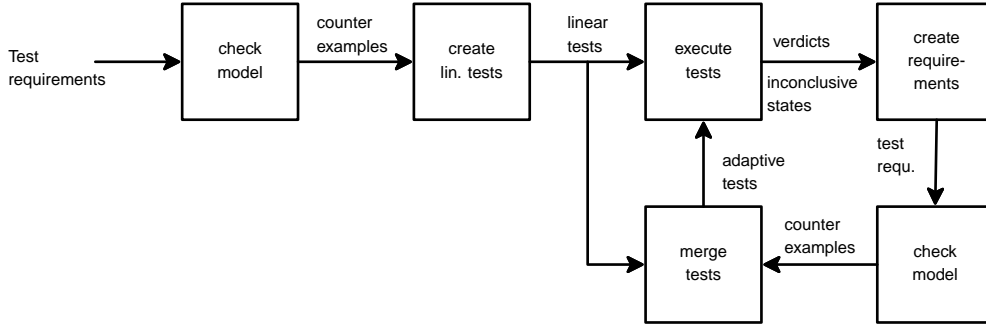


Figure 2. Creating test-cases from a nondeterministic model iteratively.

A problem is conceivable in the following scenario: Consider a model that consists of a nondeterministic choice with regard to the transitions (s_l, s_m) and (s_l, s_n) , and an according linear nondeterministic test-case $t_{NL} = (t, R)$, where the described sequence is $t := \langle s_0, \dots, s_l, s_m, \dots \rangle$. If the test-case is executed on an implementation which always chooses (s_l, s_n) rather than (s_l, s_m) , then the verdict is inconclusive, and the test-case has to be extended. While this extension t_e does begin with (s_l, s_n) and therefore resolves one inconclusive verdict, it could still look like this: $t_e := \langle s_l, s_n, \dots, s_l, s_m, \dots \rangle$; that is, it could include the same nondeterministic transition again. Assuming an implementation that never chooses (s_l, s_m) , the test requirement cannot be fulfilled on this path.

To overcome this problem, it is necessary to avoid the occurrence of the transition (s_l, s_m) when extending test-cases. If the implementation is deterministic, then this can be achieved by refining the model with regard to the nondeterministic transition. However, a test-case extension t_e made for the nondeterministic choice of (s_l, s_n) over (s_l, s_m) might itself contain another nondeterministic choice with regard to, e.g., (s_x, s_y) and (s_x, s_z) , possibly requiring yet another branch. This branch might contain another nondeterministic choice, and so on. Consequently, many refinement iterations would be necessary. A possible alternative is to force the model-checker to create only deterministic traces (e.g., using invariants) as extensions. In the worst case, the process has to be stopped after a certain number of iterations.

4. Nondeterministic Testing with NuSMV

The previous section introduced an approach to create test-cases from nondeterministic models. The approach is based on linear counterexamples created with model-checkers. This section describes concrete methods to implement these ideas using the model-checker NuSMV [6]. Modeling of nondeterminism is a basic feature of NuSMV, so there remain two problems that need to be solved. First, the nondeterministic steps within a test-case need to be

identified in order to create linear nondeterministic test-cases from counterexamples. Second, existing test-cases need to be extended with alternative paths in order to resolve inconclusive verdicts.

4.1. Identifying Nondeterministic Choice

In NuSMV and other model-checkers, models are specified as transition systems. NuSMV allows nondeterministic assignments for variables, where the assigned value is chosen out of a set expression or a numerical range. In this paper, we use the ASSIGN syntax of NuSMV to describe the transition relation; as TRANS is only a syntactic variation there are no limitations when applying the presented techniques to it. Listing 1 shows an example ASSIGN section. The first transition described in Listing 1 is deterministic, that is, whenever $condition_1$ is encountered, var is assigned $next_value_1$ in the next state. Here, $condition_1$ can be any logical formula on the variables defined in the model. The second transition is nondeterministic. Upon $condition_2$, var is assigned either $next_value_a$ or $next_value_b$. Each condition represents $\alpha_i \wedge \gamma_{i,j}$ of one transition $\delta_{i,j}$ for variable x_i in the transition system. If there are multiple next values, then each next value represents one different $\beta_{i,j}$ for $\delta_{i,j} = \alpha_i \wedge \beta_{i,j} \wedge \gamma_{i,j}$. We can therefore think of a nondeterministic transition as $\delta_{i,j} = \alpha_i \wedge B_{i,j} \wedge \gamma_{i,j}$, where $B_{i,j}$ is a set of post-state conditions.

```

ASSIGN
  next(var) := case
    condition1: next_value1;
    condition2: {next_valuea, next_valueb};
    ...
  esac;

```

Listing 1. Example ASSIGN section of a NuSMV model.

A counterexample created from such a NuSMV model contains no indication about which choices were made deterministically, and which were made nondeterministically. While a basic Kripke structure does not distinguish between input and output values, it is still conceivable that the model-checker itself is extended such that the counterexamples it returns are annotated to indicate nondeterministic choice, as such a choice is explicit in the transition system model. As we want to use the model-checker NuSMV to illustrate our approach but not to make our approach specific to NuSMV, we describe a method that works independently of the chosen model-checker.

Nondeterministic choice is detected by adding a special Boolean variable `ND_var` for each variable `var` that has nondeterministic transitions. This special variable is set to true whenever a nondeterministic choice is made. To achieve this, the transition relation of `ND_var` is defined such that for each nondeterministic transition in the model, the corresponding condition is used to set `ND_var` to true. In all other cases, 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. This is illustrated in Listing 2. The annotation is straight forward, and can easily be automated.

```
VAR ND_var: boolean;
ASSIGN
  init(ND_var) := 0;
  next(ND_var) := case
    condition1: 0;
    condition2: 1;
    ...
  esac;
```

Listing 2. Marking nondeterministic choice in Listing 1 with a dedicated variable.

Counterexamples resulting from such an annotated model can be transformed to linear nondeterministic test-cases using the values of `ND_var`. If for a given counterexample at a given state the observed values match the expected deviate from the expected values, then the verdict is pass. If the values do not match, the verdict is inconclusive if `ND_var` is true for that variable where the expected value differs from the observation.

The verdict function only recognizes if an unknown transition occurred at an execution step where a nondeterministic transition is possible. This means that a test-case execution that really fails at such an execution step can be reported as inconclusive. As a consequence, an inconclusive verdict has to be verified before trying to extend the linear test-case. For example, this can be achieved by model-

checking a special property:

$$\phi = \Box s \rightarrow \bigcirc \neg s'$$

Here, s and s' represent the observed values in the pre and post states as logical expressions (e.g., conjunction of $variable = value$ propositions). If ϕ results in a counterexample when checked against the model, then the transition is valid (the test-case is really inconclusive), else the transition is not valid (a fault was detected).

If ϕ results in a counterexample when checked against the model, this means that the observation represents a valid transition. Consequently, it is truly inconclusive and an alternative branch can be calculated to extend the test-case. If the model is consistent with ϕ , then the observations really do not represent a valid next state, and the test-case failed.

4.2. Extending Nondeterministic Test-Cases

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 expanding them in the nondeterministic test-cases.

Each test-case is related to one or more test requirements. Therefore, when a test-case execution results in an inconclusive verdict at state s_n , a new test requirement can be derived in order to extend the test-case. The test requirement $R'_T = (K', \phi)$ is derived from test requirement $R_T = (K, \phi)$. The model $K = (S, S_0, T, L)$ is adapted such that the initial state is set to the inconclusive state s_n of the test-case execution, resulting in $K' = (S, \{s_n\}, T, L)$.

If $K' \not\models \phi$, then a linear nondeterministic test-case t'_{NL} can be created from the resulting counterexample. The original nondeterministic test-case t_{ND} is extended with the new test-case t'_{NL} .

4.3. Improving the Test-Case Extension Process

There is no guarantee that a test-case extension calculated upon an inconclusive verdict is free of nondeterministic transitions. As was noted above, this might make it necessary to run through many iterations of calculating test-case extensions. If fairness with regard to the nondeterministic choice is not guaranteed, then the approach can theoretically not terminate at all. Therefore, model refinement and avoidance of nondeterministic choice were proposed as improvements.

Refinement of the model can only be done if the implementation is deterministic. Then, if a valid nondeterministic transition (s_d, s_n) is observed, the model can be refined to also choose this transition. In NuSMV, the conditions

within a `case`-statement are ordered such that, for example, the second condition is only evaluated if the first condition evaluates to false. Therefore, we can add a new transition description at the beginning of the `case`-statement, where the condition is the conjunction of all atomic propositions in $L(s_d)$, and the next state is the value n of var in $L(s_n)$, i.e., $var = n \in L(s_n)$. This is illustrated in Listing 3 for the example transition relation first given in Listing 1. This refinement can be performed after each inconclusive verdict. The next time the model-checker calculates a counterexample it will take the same transition as the implementation does, which increases the chances of a pass or fail verdict.

```

ASSIGN
  init( $var$ ) := 0;
  next( $var$ ) := case
     $\bigwedge_{x \in L(s_d)} x$ :  $n$ ;
     $condition_1$ :  $next\_value_1$ ;
     $condition_2$ :  $\{next\_value_a, next\_value_b\}$ ;
    ...
  esac;

```

Listing 3. Refinement of a nondeterministic choice with regard to a deterministic implementation, that chooses next value n after state s_d .

If the implementation is not deterministic, then we can only terminate after a certain number of iterations if a looping behavior occurs, or try to avoid nondeterministic transitions in the first place. In NuSMV, the latter can easily be achieved by adding invariants to the model:

```

INVAR ND_ $var$  != 1

```

This example invariant achieves that NuSMV only considers states where the variable var does not have to make a nondeterministic choice. Of course there is no guarantee that the test requirement can be fulfilled on such a path, so it might be necessary to call the model-checker a second time without the invariant.

5. Experiment

To illustrate the presented techniques, we use the Safety Injection System [3], which has frequently been used for test-case generation research (e.g., [9]). Tables 1, 2 and 4 show the SCR (Software Cost Reduction) specification of this model, originating from [3]. The system is responsible for injecting reserve water in a nuclear reactor safety system if the water pressure is too low. Depending on various conditions, the system is overridden. The notation $@T(expr)$

signifies $expr$ becoming true. Refer to [3] for a detailed description of the system.

Table 1. SCR Mode Transition Table for Pressure [3].

Current Mode	Event	New Mode
<i>TooLow</i>	$@T(WaterPres \geq Low)$	<i>Permitted</i>
<i>Permitted</i>	$@T(WaterPres \geq Permit)$	<i>High</i>
<i>Permitted</i>	$@T(WaterPres < Low)$	<i>TooLow</i>
<i>High</i>	$@T(WaterPres \leq Permit)$	<i>Permitted</i>

Initial State: Mode = *TooLow*, *WaterPres* < *Low*

Table 2. Deterministic SCR Event Table for Overridden [3]. *Inmode* is true, if the mode described by the row is entered.

Mode	Events	
<i>High</i>	<i>False</i>	$@T(Inmode)$
<i>TooLow</i> , <i>Permitted</i>	$@T(Block = On)$ <i>WHEN</i> <i>Reset</i> = <i>Off</i>	$@T(Inmode)$ OR $@T(Reset = On)$
<i>Overridden</i>	<i>True</i>	<i>False</i>

As an example, we assume that the exact behavior with regard to overriding is not fully specified (underspecification); e.g., it might not be of interest to the verification process; it could also be too early in the development process so that the full behavior is not known, or maybe it simply is an allowed implementation choice. The specification should only require the safety injection to work correctly. For this, a nondeterministic version of the specification is created. Table 3 shows a nondeterministic event table for the variable override, adapted from the deterministic version in Table 2.

The SCR specification can be automatically converted to an SMV model; e.g., Gargantini and Heitmeyer use an SMV model of the SIS example in [9]. Listing 4 lists the relevant section containing the nondeterministic decisions of the SMV model. The SMV model is used to derive test-cases with different coverage criteria. For each criterion a set of trap properties is created. Trap properties are created from the deterministic model. In our experiments, test-cases are executed on a simple Python implementation that conforms to the deterministic model.

Test requirements are created for automatically generated trap properties. *State* coverage requires each variable to take all its values, *Transition* coverage requires all transitions described in the SMV 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 SMV model to be taken. Finally, *Reflection* describes a set of trap properties that are created by reflecting the transition relation

Table 3. Nondeterministic Event Table for *Overridden*. The last column represents a nondeterministic choice between *True* and *False* (Not in original SCR notation).

Mode	Events		
<i>High</i>	<i>False</i>	<i>False</i>	@T(<i>Inmode</i>)
<i>TooLow</i>	@T(<i>Block = On</i>) WHEN <i>Reset = Off</i>	@T(<i>Inmode</i>)OR @T(<i>Reset = On</i>)	<i>False</i>
<i>Permitted</i>	@T(<i>Block = On</i>) WHEN <i>Reset = Off</i>	<i>False</i>	@T(<i>Inmode</i>) OR @T(<i>Reset = On</i>)
<i>Overridden</i>	<i>True</i>	<i>False</i>	<i>True</i> or <i>False</i>

Table 4. SCR Condition Table for *SafetyInjection* [3].

Mode	Conditions	
<i>High, Permitted</i>	<i>True</i>	<i>False</i>
<i>TooLow</i>	<i>Overridden</i>	<i>NOT Overridden</i>
<i>SafetyInjection</i>	<i>Off</i>	<i>On</i>

Table 5. Results of the test-case generation using a deterministic implementation.

Method	Initial Tests	Inconclusive	Iterations
State	211	0 (0%)	1
Transition	13	1 (7.7%)	1
Condition	27	3 (11.1%)	1
Trans.Pair	87	21 (24.1%)	3
Reflection	289	32 (11.1%)	4

```

next(Overridden) := case
  Pressure=TooLow & Pressure! =next(Pressure): 0;
  Pressure=TooLow & Reset!=On & next(Reset)=On: 0;
  Pressure=TooLow & Block!=On & next(Block)=On &
    Reset=Off: 1;
  Pressure=Permitted &
    Pressure! =next(Pressure): {0,1};
  Pressure=Permitted & Reset!=On &
    next(Reset)=On: {0,1};
  Pressure=Permitted & Block!=On & next(Block)=On &
    Reset=Off: 1;
  Pressure=High & Pressure! =next(Pressure): {0,1};
  1: Overridden;
esac;

```

Listing 4. Nondeterministic transition relation for variable *Overridden*.

as properties, and then applying mutation to these properties [4]. The trap properties were automatically generated using the deterministic model. For the experiment, only falsifiable trap properties were used. Different trap properties sometimes result in identical or subsumed test-cases, these were not removed in the experiment, therefore the number of test requirements corresponds to the number of test-cases created in the initial test-suite.

Figure 3 shows an example trace generated for the trap property $\square(Pressure = High \wedge \neg((WaterPres < Permitted)) \rightarrow \bigcirc(\neg(WaterPres < Permit) \rightarrow (Pressure = Permitted))$, edited for brevity. To the right of the counterexample, the according linear non-

deterministic test-case is depicted. Rather than listing the set of valid atomic propositions, states are abbreviated with their number according to the counterexample. The nondeterministic choice is highlighted with a gray node. This nondeterministic choice differs from how the implementation performs, therefore the test-case execution is inconclusive (*Overridden* is true in state 34). Figure 4 shows the counterexample resulting from the new test requirement, where the initial state equals the choice made the implementation (state 34). The original test-case and the new test-case are merged (the new counterexample is appended as alternative branch to state 33), and now the implementation passes the test-case.

The results of the test-case generation are listed in Table 5. The number of inconclusive verdicts is given as a percentage of the size of the initial test-suite. The fourth column contains the number of iterations that were necessary until all inconclusive results were resolved. State coverage is a very weak criterion and results in very short test-cases, all of which can be covered without taking a nondeterministic transition. Transition pair coverage creates the longest test-cases of the considered coverage criteria, and therefore nondeterministic transitions occur more often than with the other criteria. In average 10.8% of the test-cases resulted in an inconclusive verdict when executed on a correct implementation. As this number is relatively small, we conclude that an iterative approach is feasible. Of course, the actual number of inconclusive results will depend on the amount of nondeterminism in the model.

```

-> State: 1.1 <-
  Reset = On
  Overridden = 0
  Block = Off
  WaterPres = 2
  Pressure = TooLow
  SafetyInjection = On
  ND_Overridden = 0
-> State: 1.2 <-
  WaterPres = 5
-> State: 1.3 <-
  WaterPres = 8
...
-> State: 1.31 <-
  WaterPres = 92
  Pressure = Permitted
  SafetyInjection = Off
-> State: 1.32 <-
  WaterPres = 95
-> State: 1.33 <-
  WaterPres = 98
-> State: 1.34 <-
  Overridden = 1
  WaterPres = 101
  Pressure = High
  ND_Overridden = 1
-> State: 1.35 <-
  Block = On
  ND_Overridden = 0

```

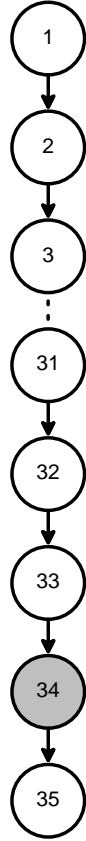


Figure 3. NuSMV Counterexample for the trap property $\square(Pressure = High \wedge \neg((WaterPres < Permitted)) \rightarrow \bigcirc(\neg(WaterPres < Permit) \rightarrow (Pressure = Permitted)))$, **and corresponding linear test-case. States are abbreviated by their numbers in the counterexample. Nondeterministic choice is highlighted with a gray node.**

6. Discussion

The performance overhead created by the iterative process very much depends on the number of inconclusive verdicts. Creation of linear nondeterministic test-cases from linear traces is straight forward, and even when using the proposed model rewriting the state space increase is minimal. The actual performance overhead therefore is determined by the number of extension traces that need to be calculated.

The feasibility of the presented approach depends on the amount of nondeterminism in the model. We believe that nondeterminism as an implementation choice is not a problem. However, nondeterminism as occurs in asynchronous,

```

-> State: 1.1 <-
  Reset = On
  Overridden = 0
  Block = Off
  WaterPres = 101
  Pressure = High
  SafetyInjection = Off
  ND_Overridden = 0
-> State: 1.2 <-
  Reset = Off
  WaterPres = 109

```

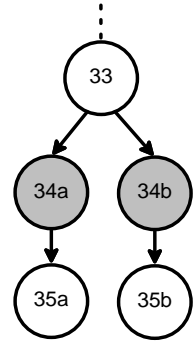


Figure 4. NuSMV Counterexample for the same trap property $\square(Pressure = Permitted \wedge \neg((WaterPres < Low)) \rightarrow \bigcirc((WaterPres < Low) \rightarrow \neg(Pressure = TooLow)))$ **but new initial state. The counterexample is used to extend the nondeterministic test-case.**

distributed systems is likely to cause too many inconclusive results in order for the method to be feasible. Synchronous systems, on the other hand, should not pose a problem.

The approach applies to both deterministic and nondeterministic systems under test. Model refinement, as proposed to reduce the number of iterations necessary to remove all inconclusive verdicts, cannot be applied when the implementation itself is nondeterministic. In that case, the options in case of reoccurring nondeterministic transitions are reduced to searching for purely deterministic paths, and to stopping after a certain number of iterations. In general, testing nondeterministic implementations has additional requirements. For example, there is no guarantee that the behavior observed during testing is the same during runtime. Therefore, test-cases have to be repeated often to increase certainty that all possible outcomes have occurred. The number of inconclusive results is likely to be higher with nondeterministic implementation. This raises the question whether it would be more efficient to aim at creating fully expanded test-cases for nondeterministic implementations in the first place, instead of applying an iterative method.

The iterative method can be used to extend any known method to derive test-cases with a model-checker, as long as passing test-cases are created; that is, such test-cases that describe the expected correct output, as opposed to failing test-cases that describe a possible error.

7. Conclusions

In this paper, we have proposed a method to extend model-checker based test-case generation techniques to be applicable to nondeterministic models. Current model-

checkers produce linear counterexamples, therefore testing was restricted to deterministic models up to now.

The method determines nondeterministic choices in linear test-cases, which achieves that during test-case execution valid deviations from the expected behavior are detected as *inconclusive*, not *fail*. To resolve inconclusive results from the test-case execution, test-cases are extended to tree-like test-cases, which include alternative branches that are the result of nondeterministic choices. Feasibility of the approach was illustrated on an example application.

The proposed solution solves a problem that is caused by the nature of model-checker counterexamples. Some test-case generation techniques based on other formalisms or using other types of tools cope with nondeterminism easier. For example, testing on nondeterministic labeled transition systems (LTS) is achieved by representing test-cases as nondeterministic LTS as well in the tool TGV [12]. Testing of nondeterministic FSMs is also common in protocol testing [15].

The model-checker NuSMV was used in this paper to illustrate how the described approach can be implemented using currently available tools. Such an implementation would profit from an extension of counter example generation techniques. For example, it would be interesting to have the model-checker create such counterexamples where all possible alternative choices at a nondeterministic decision are listed. This would allow to more efficiently detect inconclusive results, as the model rewriting would not be necessary. Furthermore, this would remove the need to check whether an inconclusive verdict is in fact a *fail*.

A further step would be to adapt the model-checker to create fully expanded test-cases, where all inconclusive results are removed. However, the question is how such an approach would perform if the number of alternative branches caused by nondeterminism is high.

References

- [1] P. Ammann, W. Ding, and D. Xu. Using a Model Checker to Test Safety Properties. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS 2001)*, pages 212–221, Skovde, Sweden, 2001. IEEE.
- [2] P. E. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, 1998.
- [3] R. Bharadwaj and C. L. Heitmeyer. Model Checking Complete Requirements Specifications Using Abstraction. *Automated Software Engineering*, 6(1):37–68, 1999.
- [4] P. E. Black. Modeling and Marshaling: Making Tests From Model Checker Counterexamples. In *Proc. of the 19th Digital Avionics Systems Conference*, pages 1.B.3–1–1.B.3–6 vol.1, 2000.
- [5] J. R. Callahan, S. M. Easterbrook, and T. L. Montgomery. Generating Test Oracles Via Model Checking. Technical report, NASA/WVU Software Research Lab, 1998.
- [6] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.
- [7] G. Fraser and F. Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006.
- [8] G. Fraser and F. Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA'07)*, 2007. To appear.
- [9] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests From Requirements Specifications. In *ES-EC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687, pages 146–162. Springer, September 1999.
- [10] G. Hamon, L. de Moura, and J. Rushby. Generating Efficient Test Sets with a Model Checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 261–270, 2004.
- [11] R. M. Hierons. Applying adaptive test cases to nondeterministic implementations. *Inf. Process. Lett.*, 98(2):56–60, 2006.
- [12] C. Jard and T. Jron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:297–315, 2005.
- [13] I. L. Li Tan, Oleg Sokolsky. Specification-based testing with linear temporal logic. In *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI'04)*, pages 493–498, 2004.
- [14] V. Okun, P. E. Black, and Y. Yesha. Testing with Model Checker: Insuring Fault Visibility. In N. E. Mastorakis and P. Ekel, editors, *Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, pages 1351–1356, 2003.
- [15] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In *Selected proceedings of the IFIP TC6 9th international workshop on Testing of communicating systems*, pages 125–140, London, UK, 1996. Chapman & Hall, Ltd.
- [16] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA*, pages 46–57. IEEE, 1977.
- [17] S. Rayadurgam and M. P. E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91, Washington, DC, April 2001. IEEE Computer Society.