Eliminating flakiness: deterministic control for validating nondeterministic Asmeta specifications

Andrea Bombarda[1 [0000−0003−4244−9319]], Silvia Bonfanti[1 [0000−0001−9679−4551]], Angelo Gargantini[1 [0000−0002−4035−0131]], and Nico Pellegrinelli[1 [0009−0000−4944−6845]]

University of Bergamo, Bergamo, Italy {andrea.bombarda, silvia.bonfanti, angelo.gargantini, nico.pellegrinelli}@unibg.it

Post-Acceptance Updated Version

This version of the paper includes a revised Section 5 that extends the algorithm presented in the original version for the *17th NASA Formal Methods Symposium (NFM)*, 2025. While the results and overall conclusions remain valid, the originally presented translation of pick statements from `Avalla` to the corresponding `Asmeta` code was found to be limited. Accordingly, the portion of Section 5 describing this translation has been rewritten, and minor adjustments have been made to Section 4, Section 2, and the conclusions. *April 2025*

Abstract. Formal methods are increasingly used in the development of safety-critical systems, offering a rigorous approach from model to implementation. However, in the validation process, the nondeterminism is a hindrance in their application, as it can lead to flaky tests or *flaky scenarios*. Scenarios written for models that implement nondeterminism produce unpredictable outcomes by complicating model validation and reducing developer confidence. In this paper, we present an approach to address the nondeterminism in the validation phase when using the `Asmeta` framework. We extend the `Avalla` language, used for scenario specification in `Asmeta`, to allow deterministic control over nondeterministic choices. This extension ensures that scenarios written for nondeterministic models execute predictably by eliminating flakiness. We demonstrate our approach using a running example of an automatic coffee vending machine.

Keywords: Scenario-based Validation · Flaky Tests · Nondeterminism.

1 Introduction

Formal methods play a crucial role in software engineering, especially in the development of safety-critical systems [3,17] such as those in the automotive [26], aviation [5,11], and medical domains [10]. These systems demand high reliability and correctness to ensure safety, as errors can lead to catastrophic consequences [34]. Formal methods provide a mathematically rigorous approach to specifying, modeling, and verifying systems, reducing the likelihood of errors not detected during development. They are increasingly being adopted to meet strict regulatory standards and improve the robustness and reliability of safety-critical systems [19,21].

Formal notations very often allow the use of nondeterminism, which may be employed to account for the environment or uncertain events [8], such as timed behavior or stochastic decisions [29], or to keep the model more abstract [7]. More specifically, we can distinguish two types of nondeterminism: *external* and *internal*. External nondeterminism arises from the uncertainty associated with inputs, whose values cannot always be predicted. Internal nondeterminism, on the other hand, refers to the nondeterministic execution of a system, regardless of the inputs, i.e., when the system itself has multiple possible behaviors or choices at a particular point in its execution, without external influence. In this work, we focus on *internal* nondeterminism.

Many formal methods support scenario-based validation, which is crucial for ensuring the reliability and robustness of formal specifications in system design. Practitioners can test how the specifications handle diverse and often complex operational conditions by simulating scenarios. This approach helps testers and developers uncover ambiguities, inconsistencies, or malfunctioning [22,23]. When using scenarios to validate formal models is possible and allowed by the chosen formalism, the presence of nondeterminism can hinder developers' adoption of these methodologies. The abstract tests generated or written for formal models may reveal unknown behaviors, making it challenging to predict how the model will behave. This problem is a well-explored issue in software engineering and software testing, even beyond the formal methods community, and is normally known as the problem of *flaky tests* [24,28]. The literature classifies a test as being *flaky* when its outcome is nondeterministic with respect to a given software version. Flaky tests can cause several problems, especially during regression testing [24]. Firstly, test failures caused by flaky tests are difficult to reproduce. Secondly, flaky tests may prevent the identification of genuine bugs. If a flaky test fails repeatedly, developers tend to overlook its failures, potentially missing real bugs [30].

For the work we present in this paper, we build on the `Asmeta` [13] framework, an open-source framework defining modeling notations and tools inspired by the well-known formal method of the Abstract State Machines (ASMs) [15,16]. `Asmeta` supports model editing, visualization, simulation, animation, validation through scenarios, verification, as well as source code generation from formal models. It has been used for the modeling and verification of many safety-critical systems (e.g., a Mechanical Ventilator [12], a Pill-Box [10], an airplane arrival manager [11], a landing gear system [5], a hemodialysis device [4], and the Hybrid European Rail Traffic Management System [20]). As in many other formal methods, `Asmeta` allows developers to embed nondeterminism in formal models. More specifically, `Asmeta` supports a specific type of rule, called `choose` rule, which randomly extracts a value meeting a specific condition from a domain. However, if scenarios are used to validate models containing nondeterminism, their usage is not suitable, as scenarios will fail or not depending to random reasons. We call these scenarios *flaky scenarios*.

In this paper, we introduce an extension to the `Avalla` language, the language used to write test scenarios in `Asmeta`. This extension addresses the issue of nondeterminism and eliminates the flakiness associated with scenarios. This stands in contrast to the conventional approach to solving nondeterminism, which involves analyzing all possible outcomes of random choices [33]. While this method is effective, it can be computationally intensive and prone to errors, particularly when recursive calls are involved. This is because test scenarios are, in general, manually written by testers. With the contribution we provide in this paper, it is now possible to force nondeterministic choices to have deterministic values. As a result, scenarios can be written in relation to nondeterministic formal models and executed without encountering flakiness. We present our approach and the extension we have implemented. Moreover, we exemplify our contribution by exploiting a simple running example of a coffee machine, that randomly dispenses coffee, milk, or tea, and we show how we have addressed the problem of flaky scenarios.

The paper is structured as follows. Section 2 describes the `Asmeta` framework, details the `Avalla` language used for writing scenarios, and outlines the characteristics of our running example of the automatic coffee vending machine (ACVM). Section 3 shows the `Asmeta` specifications including nondeterminism for the ACVM. Section 4 explains how the `Avalla` grammar has been extended in order to address the problem of scenario flakiness, while the semantics and the way in which `Avalla` deterministic scenarios are handled by `Asmeta` are presented in Section 5. Section 6 discusses the impact of controlling the nondeterminism on testing results and scenarios effectiveness. Then, in Section 7 we present related works in test flakiness in formal methods and software engineering. Finally, Section 8 concludes the paper.

2 The `Asmeta` framework

In this section, we introduce the `Asmeta` framework [3,13], which is based on Abstract State Machines (ASM), an extension of Finite State Machines (FSM) in which unstructured control states are replaced by states with arbitrarily complex data. As shown in Figure 1, the `Asmeta` framework includes various tools designed to assist developers in different stages of the software life-cycle: Design, Development, and Operation. The *design* phase includes activities such as modeling, validation, and verification, beginning with the system requirements. Once these requirements are formalized and verified, iterative refinements can incorporate additional details into the model. Subsequently, the *development* and *operation* phases can advance independently. The development phase involves generating code or tests from models, while the operation phase supports runtime simulation and monitoring.

This paper focuses on the *Modeling* and *Validation* phases. In the modeling phase, the user implements the system models using the `AsmetaL` language

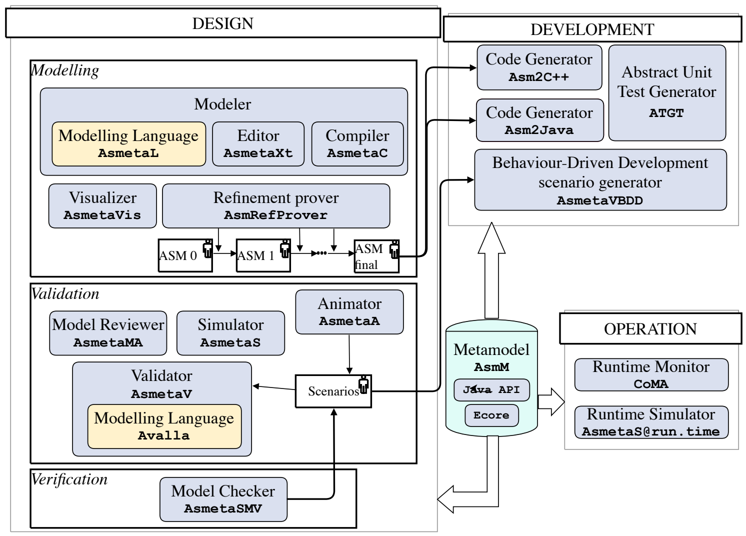

Fig. 1: The ASM development process powered by the `Asmeta` framework

and the editor `AsmetaXt`, which provides editing support. The model simulator `AsmetaS` supports the validation process, which enables users to execute their models. Additionally, the animator `AsmetaA` provides users with step-by-step execution, facilitated by a graphical interface. Furthermore, the model reviewer `AsmetaMA` verifies the meta-properties of the specification under analysis. Lastly, the `AsmetaV` executes scenarios written using the `Avalla` language [18]. Each scenario comprises the anticipated system behavior, and the tool meticulously checks whether the machine operates correctly.

In this paper, we address the problem of validating `Asmeta` specifications including nondeterminism. In `Asmeta`, it is possible to model two different types of nondeterminism: external or internal. The former employs monitored functions to make choices from the environment, whereas the latter uses algorithms to make those choices. Here, we focus on the second type, implemented in `AsmetaL` through the `choose` rule.

---

choose v1 **in** D1, ..., vn **in** Dn **with** Gv1,...,vn **do** Rv1,...,vn [**ifnone** P]

This rule allows the simulator to select a set of random variables `v1`, ..., `vn` from the specified domains or sets of elements `D1`, ..., `Dn`, that can fulfill the defined condition `Gv1,...,vn`, and then execute the rule `Rv1,...,vn` using the selected variables. In case no elements satisfies the condition `Gv1,...,vn`, the `P` transition rule is executed, which is assumed to be `skip`[1] as default.

Validating a specification containing a `choose` rule using an `Avalla` scenario can be challenging because it's impossible to guarantee that the execution under

[1] In `Asmeta`, the `skip` rule is a rule doing nothing.

Listing 1: Ground model for the ACVM

```
1 asm ACVMGround
2 import StandardLibrary
3
4 signature:
5 enum domain Product = {COFFEE | TEA |
      MILK}
6 controlled dispensed : Product
8 definitions:
9  main rule r_Main =
10    choose $p in Product with true do
11      dispensed := $p
12
13 default init s0:
14  function dispensed = undef
```

test will select the same value as the tester intends when writing the scenario. So in this paper, we address this issue.

2.1 Automatic Coffee Vending Machine

The running example we use to test and explain our approach for addressing scenario flakiness in `Asmeta` is a simple automatic coffee vending machine (ACVM) [14].

The ACVM distributes coffee, tea, or milk and accepts only 1 Euro and 2 Euros coins. If the user inserts 1 Euro, the machine distributes milk if available; if the user inserts 2 Euros the machine randomly distributes coffee or tea, if available. When the machine distributes a drink, its availability is decremented and the money is preserved in the machine. At the start, the machine has 10 coffees, 10 teas, and 10 milks. The machine can contain 25 coins at maximum; When this limit is reached the machine does not distribute products any longer.

3 ACVM implementation

As supported and suggested by the `Asmeta` workflow, we developed our ASM by taking advantage of model refinement techniques [6]. In this section, we describe the two refinements we implemented and demonstrate how scenarios for them can be flaky. It is worth noting that we separated the development of the `Asmeta` models into two distinct refinements to streamline the development process and accommodate specifications with an unconstrained choose rule, in one case, and with a constrained choose rule in the other.

3.1 Ground model

In the ground model of the ACVM, we modeled a simplified version of the coffee machine. The machine randomly distributes one product among coffee, tea, and milk at each step. The `Asmeta` specification is reported in Listing 1, where the distributable products (`COFFEE`, `TEA`, and `MILK`) are listed in the `Product` enumerative domain (Line 5). Once the ACVM has randomly chosen a product

Listing 2: Flaky `Avalla` scenario for the specification in Listing 1

```
scenario scenario1
load ACVMGround.asm

step
check dispensed = TEA;
```

Listing 3: Non-flaky `Avalla` scenario for the specification in Listing 1

```
scenario scenario1
load ACVMGround.asm
step
check dispensed = TEA or dispensed = COFFEE or
        dispensed = MILK;
```

Listing 4: Successful execution

```
[...]
<State 1 (controlled)>
    dispensed = TEA
    result = 1
    step__ = 1
</State 1 (controlled)>
check succeeded: dispensed = TEA
```

Listing 5: Failing execution

```
[...]
<State 1 (controlled)>
    dispensed=MILK
    result = 1
    step__ = −1
</State 1 (controlled)>
CHECK FAILED: dispensed = TEA at step 1
```

`$p` in the `Product` domain (Line 10), the product is stored in the `dispensed` controlled function[2] (Line 11).

This `Asmeta` specification exhibits a nondeterministic behavior, and this can cause every scenario to be flaky. An example of a flaky scenario for this specification is reported in Listing 2: The ACVM machine does one execution step and then checks whether the dispensed product is the `TEA`.

Listing 4 and Listing 5 present an excerpt of the output obtained from executing the scenario in Listing 2 with the `AsmetaV` validator. In the first case, the ACVM chooses to dispense `TEA`, and the scenario terminated without errors. Instead, in the second case, the ACVM choses to dispense `MILK` and a check failed is signaled by the `Asmeta` validator. An example of an alternative scenario that would never be flaky is reported in Listing 3: Instead of checking the equivalence between `dispensed` and a specific value, the scenario checks its equivalence with any of the elements in the domain specified in the `choose` rule (i.e., in the `Product` domain). The two scenarios in Listing 2 and Listing 3 assess different behaviors of the ACVM, capturing a subset of behaviors in one scenario compared to the other. A machine consistently dispensing `TEA` would pass both tests; A machine consistently dispensing `COFFEE` would only pass the second test. Moreover, writing scenarios always with a set of `OR` to avoid flakiness may be infeasible, as the user loses the control to verify that a function takes on a specific value, and the number of possible paths after a nondeterministic choice may grow exponentially. Therefore, the issue of flaky scenarios is inescapable and cannot be resolved effortlessly by modifying checks within scenarios.

[2] A *controlled* function is a function whose value is set by the machine and read by the environment.

Listing 6: Refined model for the ACVM

```
1 asm ACVMRef
2 import StandardLibrary
3
4 signature:
5  enum domain Product = {COFFEE | TEA | MILK}
6  domain QuantityDomain subsetof Integer
7  domain CoinDomain subsetof Integer
8  domain InputCoinDomain subsetof Integer
9  controlled dispensed: Product
10 controlled available: Product -> QuantityDomain
11 controlled coins: CoinDomain
12 monitored insertedCoin: InputCoinDomain
13 static price: Product -> InputCoinDomain
14
15 definitions:
16  domain QuantityDomain = {0 : 10}
17  domain InputCoinDomain = {1 : 2}
18  domain CoinDomain = {0 : 25}
19
20  function price($prod in Product) = switch $prod
21    case COFFEE: 2
22    case TEA: 2
23    case MILK: 1
24  endswitch
25
26  rule r_serveProduct($p in Product) = par
27    dispensed := $p
28    available($p) := available($p) - 1
29    coins := coins + price($p)
30  endpar
31
32  main rule r_Main =
33   if(coins < 25) then
34    choose $p in Product with price($p) =
35          insertedCoin and available($p) > 0
36              do r_serveProduct[$p]
37    ifnone
38     dispensed := undef
39   endif
40
41 default init s0:
42  function coins = 0
43  function dispensed = undef
44  function available($p in Product) = 10
```

3.2 Refined model

In this refined model, we implemented the full behavior of the ACVM, by including coins and considering the availability of each product. In this manner, the products will only be dispensed by the ACVM if the number of inserted coins matches the specific product's price, and the quantity of the chosen product is sufficient. Listing 6 reports the refined model of the ACVM. In addition to what already introduced in the ground model, this refined version creates three more domains (Line 6-Line 8) to restrict the amount of products available (`QuantityDomain`), the coins accepted (`InputCoinDomain`), and those contained in the ACVM (`CoinDomain`). Furthermore, this specification adds new controlled functions (Line 10-Line 11) to store the availability for each product (`available`) and the amount of coins contained in the ACVM (`coins`). Considering that this refinement level allows the interaction with the user, we added a monitored[3] function `insertedCoin` which stores the coin type inserted by the user (Line 12). Finally, the price for each product is statically defined (Line 20).

When executing the ACVM (Line 32), if a new coin is inserted, the machine randomly chooses a product `$p` with `price` equal to the amount of `insertedCoin` and which is still `available`. If such a product exists, it is served by executing the `r_serveProduct` rule; If not, the `dispensed` product is set to `undef`.[4]

As for the ground model, this refined version of the ACVM still contains nondeterminism, and the scenario shown in Listing 7 is *flaky*. More specifically, even if a nondeterministic main rule is executed, the initial scenario step remains

[3] A *monitored* function is a function whose value is set by the environment and read by the machine.

[4] In `Asmeta`, `undef` is equivalent to the *null* value, and can be used in any domains.

Listing 7: `Avalla` scenario for the specification in Listing 6

```
scenario scenario2
load ACVMRef.asm

/* First step. 1 Euro is
 inserted: only MILK can be
 dispensed */
set insertedCoin := 1;
step
check dispensed = MILK;
check available(MILK) = 9;
check available(TEA) = 10;
check available(COFFEE) = 10;

/* Second step. 2 Euros are inserted:
 COFFEE or MILK can be dispensed */
set insertedCoin := 2;
step
check dispensed = COFFEE;
check available(COFFEE) = 9;
check available(MILK) = 9;
check available(TEA) = 10;
```

Listing 8: Scenario with failing execution

```
[...]
<State 2 (controlled)>
    [...]
    dispensed=TEA
</State 2 (controlled)>
CHECK FAILED: dispensed = COFFEE at step 2
CHECK FAILED: available(COFFEE) = 9 at step 2
check succeeded: available(MILK) = 9
CHECK FAILED: available(TEA) = 10 at step 2
```

Listing 9: Scenario with successful execution

```
[...]
<State 2 (controlled)>
    [...]
    dispensed=COFFEE
</State 2 (controlled)>
check succeeded: dispensed = COFFEE
check succeeded: available(COFFEE) = 9
check succeeded: available(MILK) = 9
check succeeded: available(TEA) = 10
```

deterministic because the only product that can be dispensed when inserting 1 Euro is `MILK`. However, in the second step, when the user inserts 2 Euros, the ACVM may choose to dispense `COFFEE` or `TEA` leading to a failing (Listing 9) or successful (Listing 8) execution.

4 Extending `Avalla` language

Flakiness is due to nondeterminsm in `Asmeta` specification, but it is revealed when a tester writes a scenario without considering the nondeterministic behavior. Here, we explain how the `Avalla` language has been extended to allow deterministic control over scenario validation, given a nondeterministic behavior in the specification. The rationale behind the `Avalla` extension is that it enables users to validate the behavior of the model when specific values are selected by `choose` rules. We have defined the `pick` statement to force the nondeterministic choice to a known value:

```
pick $v [in ruleSignature] := value;
```

The logical variable `$v`, used in a `choose` in the specification, is forced to `value`, where `value` is an expression having the same domain `D` as the `$v` variable. Note that the `Asmeta` grammar does not permit the reuse of the same name for two local variables within a single macro rule; reuse across different rules, however, is permitted. To resolve this issue, if more logical variables with the same name are used in the `Asmeta` specification, the rule where the variable is used can be explicitly specified. The `Asmeta` grammar supports overloading of macro rules, i.e., it allows defining multiple macro rules with the same name as

Listing 10: Non-flaky `Avalla` scenario for the specification in Listing 1

```
1 scenario scenario1
2
3 load ACVMGround.asm
4
5 pick $p := TEA;
6 step
7 check dispensed = TEA;
```

Listing 11: Non-flaky `Avalla` scenario for the specification in Listing 6

```
1 [...]
2 // Second step
3 set insertedCoin := 2;
4 pick $p in r_Main := COFFEE;
5 step
6 check dispensed = COFFEE;
7 check available(COFFEE) = 9;
8 check available(MILK) = 9;
9 check available(TEA) = 10;
```

long as their parameters differ. To specify the macro rule signature the format `r_ruleName(domain1,domain2,...)` must be followed, where the domains represent the types of the parameters. If the macro rule does not take any parameter, the parentheses must be omitted.

Given the flaky scenario reported in Listing 2, where the execution randomly ended with `check succeeded` or `check failed` result, here we present the same scenario using the `pick` statement to force the value selected by the `choose` rule. The scenario is reported in Listing 10, where `TEA` is chosen (see Line 5). Similarly, the scenario in Listing 7 for the refined ACVM model can be modified as shown in Listing 11, by removing its flakiness and forcing the selection of `COFFEE` (see Line 4).

When validating the `Asmeta` specification with the `Avalla` scenarios modified as described above, the outcome of their execution is always the same, and both scenarios pass.

5 Semantics: translating `pick` from `Avalla` to `Asmeta`

As explained in [18], the `Avalla` language semantics is given in terms of `Asmeta` specification. The translation is performed by the `AsmetaV` validator which, given the original `Asmeta` specification as input, translates the scenario into an `Asmeta` model to make it behave as required by the user-defined scenario. Some principles are adopted during the translation process, the main ones of which are as follows.[5] Monitored functions are converted into controlled to set their values based on the `set` commands. `check` commands are translated into conditional rules over controlled functions to check whether the function values are as expected and, if not, signal an error. Furthermore, the `main rule` is overwritten, and a wrapper is created to execute the original rule step by step. At each step, checks and sets are performed.

To maintain consistency with the approach already implemented by `AsmetaV`, we decided to follow the same process for the `pick` statement. More specifically, when defining the translation mechanism we considered three principles:

[5] For further details, refer to [18].

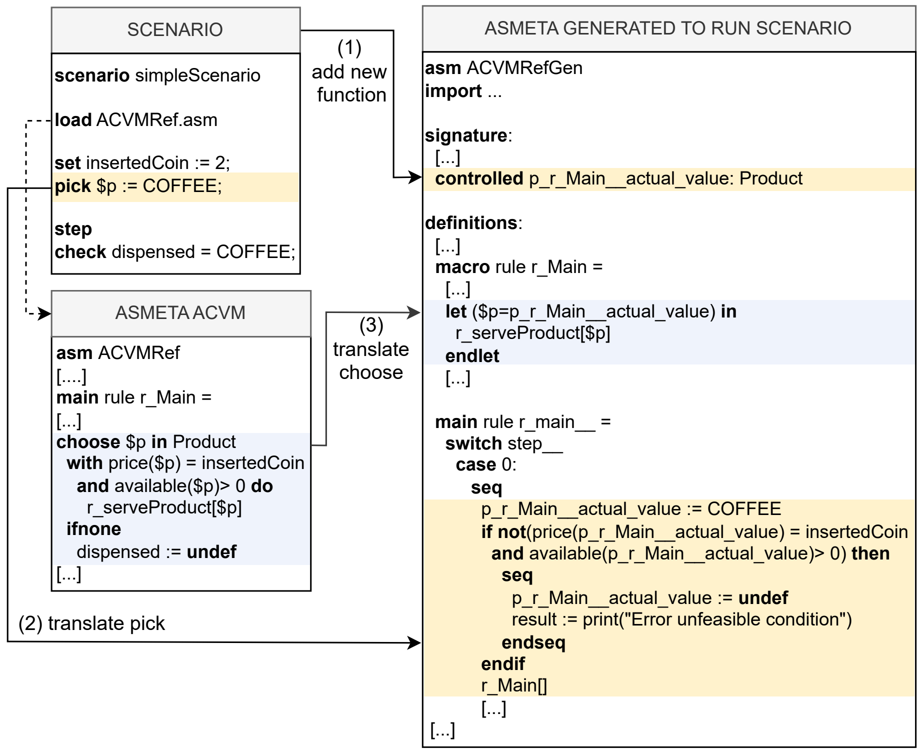

Fig. 2: Translation of `pick` in `Asmeta` for a simple scenario of `ACVMRef` specification

(A) Scenarios with no `pick` statements should be treated as they were treated before;
(B) `pick` statements could be present only in some steps, and when in a step there is no pick, the update set is computed as before with a nondeterministic behavior;
(C) Only valid values can be selected with a `pick`, i.e., it should not be possible to select a value outside the original domain of the `choose` rule, or a value clashing with the condition in the `choose`.

In the following, we better detail how we exploited these principles for the translation of the `pick` statement from `Avalla` to its corresponding `Asmeta` code.

Signature - New controlled functions. For each logical variable `$v` used in a `choose` rule that has at least one variable picked during the scenario, we introduce a controlled function `v_macroSignature__actual_value` of the same type as `$v`, where `macroSignature` denotes the signature of the macro rule in which the `choose` rule is defined. This function is used to store the actual value chosen by the user during the scenario execution when the variable is picked. Otherwise, it holds a randomly chosen value, in accordance with principle B. For instance, the command **pick** $p := COFFEE requires the insertion of a corresponding controlled function (see (1) in Figure 2).

Translation of `choose` rules. The translation of a `choose` rule takes into account the previously mentioned principle A: If none of the logical variables defined by the `choose` rule are ever picked in the scenario, the choose rule is left as in the original `Asmeta` specification. Instead, if at least a logical variable is picked at least once in the scenario, the original `choose` rule is translated into a `let` rule, as shown in step (3) of Figure 2. The resulting `let` rule declares the same variables as the original `choose` rule and assigns to each of them its corresponding controlled function. The `do` rule of the original `choose` rule becomes the `in` rule of the newly introduced `let` rule. Since the guard of the original `choose` rule is incorporated into the translation of `pick` statements, the `ifnone` rule is omitted in this part of the translation.

Translation of `pick` statements. For every `choose` rule in which at least one variable is picked at least once in the scenario, two situations can occur at each step of the scenario: (i) all the variables in the `choose` rule are picked for that step, (ii) some or all variables are not picked. In both cases, for each variable `$v` that is picked in the step, the picked value is assigned to the corresponding controlled function `v_macroSignature__actual_value`. If the assigned value falls outside the original domain specified by the `choose` rule, an error is raised, in accordance with principle C. After that, in the former case, shown in step (2) in Figure 2, a `conditional` rule checks whether the picked values satisfy the condition of the original `choose` rule: If the condition is not satisfied, an error is raised (as per principle C) and the controlled functions are set to `undef`. In the latter case, following principle B, a single `choose` rule is used to assign random values to all variables `$v` that are not picked in the step and to update the corresponding `v_macroSignature__actual_value` controlled functions. The condition of this `choose` rule is derived from the condition of the original `choose` rule by substituting picked variables with the corresponding controlled function. In the event that, given the values already assigned for the picked variables, there is no possible assignment for the remaining (not-picked) variables that satisfies the condition, an error is raised (principle C) and the controlled functions are set to `undef`.

6 Qualitative evaluation

In this section, we preliminary assess the proposed technique by qualitatively evaluating its advantages and disadvantages. We summarize the pros and cons of each approach that could be used when writing `Avalla` scenarios for nondeterministic `Asmeta` specifications in Table 1, and we detail them in the following. In general, when writing `Avalla` scenarios, we can follow three different approaches:

Approach	Pro	Cons
CLASSIC	No need to control nondeterminism No new commands to learn	Reduced fault detection capability It may be infeasible
FLAKY	No new commands to learn	Failure causes unclear
PICK	Clear cause for failure Fault detection capability	Need to learn a new command Need to control nondetermism Need to write multiple scenarios

Table 1: Preliminary Evaluation

CLASSIC: Users write the scenarios without the approach proposed in this paper, and they require that they never fail. The `check` commands will be written in a way that they won't fail (e.g., using weak conditions over the state when the exact value is unknown). An example is the scenario reported in Listing 3 where the `check` states that a product is dispensed, but it accepts any possible value.

FLAKY: Users create scenarios as they were traditionally written before the approach proposed in this paper, acknowledging that some tests may be *flaky*. They may occasionally fail when executed repeatedly, but one may require that each scenario does not fail at least once when executed a reasonable number of times. Otherwise, we deem the scenario wrong. An example is the scenario reported in Listing 2, which can either pass or fail (as illustrated in Listing 4 and Listing 5) due to nondeterministic factors.

PICK: Users adopt the `pick` commands as proposed in this paper, so no correct scenario is expected to ever fail when the specification is not faulty. An example is the scenario reported in Listing 10, where we force nondeterministic choices to a deterministic value.

The main advantage of the CLASSIC approach is that it does not require the tester to exactly *drive* the machine when nondeterministic choices are made. The behavior remains nondeterministic as originally specified by the modeler. No extra commands (like `pick`) are used. However, this makes the scenarios more tolerant and they may not be effective in revealing potential faults. If, for instance, the machine always chooses a behavior, the CLASSIC approach won't allow testers to notice it, assuming that the specific behavior is captured by a series of OR in the `check` condition, as in Listing 3. If the original `choose`, which is:

```
choose $p in Product do dispensed := $p
```

is erroneously implemented, by simply removing `MILK` from the possible choices, as in the following excerpt

```
choose $p in {TEA, COFFEE} do dispensed := $p
```

such fault can not be discovered, and the scenario would not fail, because the `check` is capturing an OR of all the elements in the `Product` domain. Indeed, with this approach, the conditions have to be general enough to be satisfied by *any* possible behavior, and this reduces the scenario fault detection capability. Note that writing a scenario that can pass for every behavior may be infeasible, as the conditions and paths may exponentially grow in number.

With the FLAKY approach, the tester can write precise checks, however, they can fail sometimes, leaving unclear if the failure is a real fault or it is because of the nondeterminism. Of course, one could accept flaky scenarios and require that they sometimes pass, but this increases the time required for running the tests and leaves uncertainty about the correctness of the specification, since some faults may be undetected. For instance, if during the simulation, a specification selects a behavior different from that specified in a scenario s, the failure of s could be due to a different nondeterministic choice or an actual fault that manifests only in specific cases. For example, if the `choose` rule in the specification in Listing 1 is implemented as in the following

```
choose $p in Product do
    if $p = MILK then dispensed := COFFEE
    else if $p = TEA then dispensed := TEA
    else if $p = COFFEE then dispensed := COFFEE
    endif
```

a scenario like the one implemented in Listing 2 would not fail, if the random choice selects `TEA`, while it would fail in all other cases. In the case the machine chooses `COFFEE`, the scenario would fail because of the nondeterminism, while in the case `MILK` is selected, it would fail and reveal an implementation fault. In this setting, a scenario implemented with the CLASSIC approach would not fail in any case, and no faults would be revealed.

Finally, with the PICK approach, the tester can write multiple scenarios, each targeting one specific behavior of the nondeterministic ASM. This is made possible by the use of the `pick` statement, introduced in this paper, which allows for forcing the value of nondeterministic choices to a known value. In this way, the scenario will be executed as being deterministic and its failure will either correspond to an error in the specification or in the scenario itself. Suppose a correct scenario written with this approach fails. In that case, testers can be sure that there is a fault in the `Asmeta` specification, allowing for a higher fault detection rate. However, this approach comes with a cost. First, testers need to explicitly control nondeterminism. Thus, this approach is unsuitable when performing black-box testing. Moreover, since with the PICK approach, one has to write scenarios to test specific system behaviors, there is the need to write multiple (or longer) scenarios while, with the CLASSIC and FLAKY approaches, a single scenario could cover more behaviors.

7 Related work

To the best of our knowledge, no existing work in the literature addresses the problem of removing flakiness from test scenarios executed against `Asmeta` nondeterministic specifications, highlighting the novelty of our proposed approach. However, flaky tests are a well-explored issue in software engineering.

Flaky tests can originate due to different reasons [24], with nondeterminism being the most common one. Nondeterminism in formal models is frequently adopted to account for the environment or uncertain events [8], such as timed behavior or stochastic decisions [29], or to keep the model more abstract [7]. This inherent nondeterminism, while useful for abstraction and modeling uncertainty, can also introduce challenges, such as Heisenbugs, which are addressed in [32]. Those are bugs that disappear or change their behavior under different observation because of the presence of nondeterminism in modeled systems. Tests can be flaky when they reveal a Heisenbug. In [9], the authors report the need to establish a well-defined testing process, considering flaky tests, for safety assurance of safety-critical systems, e.g., in the avionics or medical domains. This applies not only to validation, as we discuss in this paper, but also to verification [31], in systems where nondeterminism is present.

`Asmeta` is not the only formal approach supporting nondeterminism. Event-B has its own implementation of probabilistic behaviors [2,1]. However, validating models written in this formalism necessitates either modifying them or providing an ad-hoc implementation of nondeterministic aspects [25]. This is in contrast with what we propose in this paper, where no modification to the `Asmeta` specification is needed. Similarly, UML-B allows for expressing probabilistic behavior [27], as well as Simulink [35]. We believe that an approach similar to what we propose in this paper would be beneficial also for the validation of specifications written with other formalisms.

8 Conclusion

Nondeterminism plays a crucial role in formal specifications by enabling the modeling of uncertainty, environmental factors, and abstract behaviors; Yet its presence during validation can lead to flaky tests that undermine the reliability and reproducibility of validation efforts.

To solve this limitation, in this paper, we addressed the issue of *flaky scenarios* in the validation of nondeterministic formal models within the `Asmeta` framework. By extending the `Avalla` language, we introduced a novel mechanism to control internal nondeterminism, ensuring deterministic behavior during scenario execution. This approach eliminates flakiness, allowing developers to write deterministic scenarios even for models with nondeterministic features. Through a running example of an automatic coffee vending machine, we demonstrated how deterministic scenarios can be defined and executed without encountering flakiness, validating the effectiveness of our extension. Our contribution enhances the usability of scenario-based validation in systems with nondeterministic specifications, where previously it was not possible to write scenarios with known outcomes. As future work, we plan to extend the set of specifications on which it is possible to apply the approach proposed in this paper, e.g., by allowing the use of the `pick` statement with `choose` rules involving variables not declared within the rule itself, either in their guard or in the domains, and to test the quantitative impact of deterministic control over nondeterminism for more complex specifications. Moreover, we plan to extend the use of `pick` to automatic scenario generation.

Acknowledgements. The work of Andrea Bombarda was supported by PNRR - ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 – CUP: B53C22006700001 - Spoke 1 - Pilot 1.4. The work of Silvia Bonfanti, Angelo Gargantini, and Nico Pellegrinelli was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. Mohamed Amine, Benoît Delahaye, and Arnaud Lanoix. Moving from event-b to probabilistic event-b. In *Proceedings of the Symposium on Applied Computing*, SAC 2017, page 1348–1355. ACM, April 2017.
2. Mohamed Amine Aouadhi, Benoît Delahaye, and Arnaud Lanoix. Introducing probabilistic reasoning within event-b. *Software & Systems Modeling*, 18(3):1953–1984, October 2017.
3. Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. *The ASMETA Approach to Safety Assurance of Software Systems*, page 215–238. Springer International Publishing, 2021.
4. Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor, and Elvinia Riccobene. Integrating formal methods into medical software development: The asm approach. *Science of Computer Programming*, 158:148–167, June 2018.
5. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. *Modeling and Analyzing Using ASMs: The Landing Gear System Case Study*, page 36–51. Springer International Publishing, 2014.
6. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. *SMT-Based Automatic Proof of ASM Model Refinement*, page 253–269. Springer International Publishing, 2016.
7. Pieter Bekaert, Eric Steegmans, K Baclawski, and H Kilov. Non-determinism in conceptual models, 2001.
8. Andrea Bianco and Luca Alfaro. *Model checking of probabilistic and nondeterministic systems*, page 499–513. Springer Berlin Heidelberg, 1995.
9. Marcel Bohme. Assurances in software testing: A roadmap. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, page 5–8. IEEE, May 2019.
10. Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. *Developing Medical Devices from Abstract State Machines to Embedded Systems: A Smart Pill Box Case Study*, page 89–103. Springer International Publishing, 2019.
11. Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. *formal MVC: A Pattern for the Integration of ASM Specifications in UI Development*, page 340–357. Springer Nature Switzerland, 2023.

12. Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, and Elvinia Riccobene. Developing a prototype of a mechanical ventilator controller from requirements to code with asmeta. 2021.
13. Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. *ASMETA Tool Set for Rigorous System Design*, page 492–517. Springer Nature Switzerland, September 2024.
14. Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Generation of c++ unit tests from abstract state machines specifications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 185–193. IEEE, April 2018.
15. Egon Börger and Alexander Raschke. *Modeling Companion for Software Practitioners*. Springer Berlin Heidelberg, 2018.
16. Egon Börger and Robert Stärk. *Abstract State Machines*. Springer Berlin Heidelberg, 2003.
17. Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189, 1993.
18. Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A scenario-based validation language for asms. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, pages 71–84, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
19. Ewen Denney and Ganesh Pai. Evidence arguments for using formal methods in software certification. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, page 375–380. IEEE, November 2013.
20. Paolo Gaspari, Elvinia Riccobene, and Angelo Gargantini. A formal design of the hybrid european rail traffic management system. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA, page 156–162. ACM, September 2019.
21. Gabriella Gigante and Domenico Pascarella. *Formal Methods in Avionic Software Certification: The DO-178C Perspective*, page 205–215. Springer Berlin Heidelberg, 2012.
22. Patrick Heymans and Eric Dubois. Scenario-based techniques for supporting the elaboration and the validation of formal requirements. *Requirements Engineering*, 3(3–4):202–218, March 1998.
23. Hardi Hungar. *Scenario-Based Validation of Automated Driving Systems*, page 449–460. Springer International Publishing, 2018.
24. Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT/FSE'14. ACM, November 2014.
25. Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot. Refinement-based validation of event-b specifications. *Software & Systems Modeling*, 16(3):789–808, February 2016.
26. Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D. Ames, Jessy W. Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. Correct-by-construction adaptive cruise control: Two approaches. *IEEE Transactions on Control Systems Technology*, 24(4):1294–1307, July 2016.
27. Mohammad Nosrati and Hassan Haghighi. A probabilistic extension of uml-b. *Computing and Informatics*, 38(1):85–114, 2019.
28. Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology*, 31(1):1–74, October 2021.

29. Andrea Pferscher and Bernhard K. Aichernig. *Learning Abstracted Nondeterministic Finite State Machines*, page 52–69. Springer International Publishing, 2020.
30. Md Tajmilur Rahman and Peter C. Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '18. ACM, October 2018.
31. Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are, 2020.
32. Sarah Sallinger, Georg Weissenbacher, and Florian Zuleger. *A Formalization of Heisenbugs and Their Causes*, page 282–300. Springer Nature Switzerland, 2023.
33. Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. Fixing nondeterminism. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, page 1–12. ACM, September 2015.
34. W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent catastrophic accidents: Investigating how software was responsible. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, page 14–22. IEEE, 2010.
35. Chikatoshi Yamada and D. Michael Miller. Using spin to check nondeterministic simulink stateflow models. In *2015 IEEE International Symposium on Multiple-Valued Logic*, page 145–151. IEEE, May 2015.