Evaluating coverage and fault detection capability of scenarios for the validation of Asmeta specifications

Andrea Bombarda[1][0000−0003−4244−9319], Silvia Bonfanti[1][0000−0001−9679−4551], Cesar Cornejo[1][0000−0003−3716−3607], Angelo Gargantini[1][0000−0002−4035−0131], and Nico Pellegrinelli[1][0009−0000−4944−6845]

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

Abstract. Scenario-based validation is a widely used technique for assessing the correctness of executable formal specifications, yet its effectiveness strongly depends on the adequacy of the scenarios used. Coverage measures have been commonly adopted to evaluate scenario adequacy, but it remains unclear whether higher coverage correlates with improved fault detection, especially at the level of formal specifications. While this relationship has been extensively studied for source code, it has received little attention in the context of executable formal models. In this paper, we investigate the relationship between coverage and fault detection capability for scenario-based validation of Asmeta specifications. We extend the AsmetaV tool with an extensive set of coverage criteria that go beyond existing *macro rule* coverage. To assess the effectiveness of these coverage criteria, we introduce a set of mutation operators for Asmeta specifications and conduct a large-scale mutation-based experimental study, based on scenarios generated through model-checking-based, random, and evolutionary techniques. Our results show that higher coverage is generally correlated with increased fault detection capability, but *macro rule* coverage alone is insufficient to capture scenario effectiveness and more fine-grained coverage criteria provide stronger correlation with mutation scores and lead to improved fault detection.

Keywords: Abstract State Machine · Asmeta· Validation · Specification coverage · Scenario-Based Testing

1 Introduction

The successful development of complex systems relies on employing an appropriate method to identify the requirements of the target system and ensure that the final product meets these requirements. Validation refers to the process of analyzing a model (which can be a formal specification) in relation to user expectations [31], verifying that it accurately represents user needs and application requirements while identifying specification faults early with minimal effort. This

process is essential to determine whether the implemented specification behaves as expected and also to verify whether it meets the intended safety requirements, particularly in the context of safety-critical systems [8, 11].

The Abstract State Machines (ASMs) method enables the writing of formal and executable specifications and is supported by the `Asmeta` framework [5, 12], which provides tools for simulation, validation, and verification. In particular, `AsmetaV` [15] allows for writing scenarios in the `Avalla` language and to validate `Asmeta` specifications through scenario execution.

Evaluating only the correctness of the model's observed behavior does not ensure that the entire model has been fully validated, as some behaviors may go undetected. A common measure to evaluate how thoroughly test cases validate software is code coverage [43]. However, covering a specific object (e.g., statements, methods, or branches) does not necessarily imply coverage of all other related or contained objects [27]. Similarly, we argue that model coverage can be used to assess the extent to which a model has been tested during validation. In scenario-based validation, users can leverage `AsmetaV` to evaluate coverage and determine whether the defined scenarios cover all *macro rules* of interest. Nevertheless, writing test scenarios that cover all *macro rules* does not guarantee that all underlying transition *rules* are exercised. Consequently, if a tester only focuses on coverage measures currently provided by `AsmetaV` when executing scenarios, it is possible that, despite achieving full *macro rule* coverage, some transition *rules* remain uncovered and certain faults go undetected.

In this paper, we present an extension of `AsmetaV` where the coverage is not limited only to *macro rules*, but it is extended to a subset of *rules* supported by `Asmeta`: *par*, *seq*, *choose*, *conditional*, *extend*, *forall*, *let*, *macro call*, *skip*, and *update rules*. We describe how we implemented this extension and illustrate its usage on a straightforward `Asmeta` model representing a simple scheduler to manage a set of jobs.

To evaluate the impact of these new criteria, and to assess the correlation between coverage and fault detection capability in the context of scenarios for `Asmeta` specifications, we adopt a mutation-based evaluation. We introduce a set of 8 mutations and we artificially generate mutated `Asmeta` specifications. Then, we use these mutants to automatically evaluate the mutation score of `Avalla` scenarios generated for a collection of `Asmeta` models. We use such a mutation score as a proxy measure for the evaluation of the fault detection capability.

Our empirical study confirms that coverage is a meaningful indicator of scenario effectiveness for `Asmeta` specifications. Across 272 generated test suites, mean coverage shows a moderate positive correlation with mutation score, suggesting that test suites achieving higher coverage tend to detect more faults. Moreover, when analyzing individual criteria, we observe that *macro rule* coverage alone is not sufficient to capture scenario adequacy, while more fine-grained measures, such as *rule* coverage, provide stronger explanatory power and fault detection capability.

The remainder of the paper is structured as follows. Section 2 describes the `Asmeta` framework, details the `Avalla` language used for writing scenarios, and

```
1  asm Scheduler
2  import StandardLibrary
3  signature:
4   enum domain Job = {JOB1|JOB2|JOB3}
5   enum domain Status = {RDY|RUN|FIN}
6   controlled st: Job -> Status
7   controlled idle: Boolean
8   monitored fin: Job -> Boolean
9  definitions:
10  macro rule r_SetRunning =
11   choose $j1 in Job with st($j1) = RDY do
13    st($j1) := RUN
14   ifnone idle:= true
15  macro rule r_SetFinished =
16   forall $j2 in Job with st($j2) = RUN do
17    if fin($j2) then
18     st($j2) := FIN endif
19  main rule r_Main =
20   par r_SetRunning[] r_SetFinished[] endpar
21 default init s0:
22  function idle = false
23  function st($j in Job) = RDY
```

Fig. 1: The Asmeta model of a simple scheduler

outlines the characteristics of our running example of the simple scheduler. Section 3 formally defines coverage criteria introduced in this work and discusses subsumption relations among them. Section 4 presents the mutation operators we implemented for Asmeta specifications. In Section 5, we empirically evaluate the coverage measures and their effectiveness in detecting faults by performing a large-scale mutation-based evaluation. Section 6 discusses possible threats to the validity of our findings. Finally, Section 7 presents related works, and Section 8 concludes the paper.

2 The Asmeta framework

In this section, we introduce the Asmeta framework [5, 12]. It 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 presented in [5], the Asmeta framework includes various tools designed to assist developers in different stages of the software life-cycle: design, development, and operation. In this paper, we focus only on the *design* part, and, in particular, on the modeling and validation activities, beginning with the system requirements. In the modeling phase, the user implements the system models using the AsmetaL language and the editor AsmetaXt, which provides editing support. The model simulator AsmetaS, which enables users to execute their models, supports the validation process, while AsmetaV executes scenarios written using the Avalla language [15]. Each scenario comprises the anticipated system behavior, and the tool meticulously checks whether the machine operates correctly.

Figure 1 reports the example of an Asmeta specification modeling a simple scheduler that manages a set of jobs. It defines three *macro rules*, namely r_SetRunning, r_SetFinished, and the *main rule* r_Main, which is the entry point of the execution and calls the other two rules in parallel. The controlled[1] function st represent the current status of each job (ready, running, or finished), while

[1] The value of a *controlled* function is set by the machine and read, i.e., used, by the environment.

the monitored[2] function fin models the environment signal indicating whether a job has completed. In the initial state (line 21), idle is set to false (line 22) and the status of all jobs is set to ready (line 23). At each execution step, the *choose rule* at line 11 non-deterministically select one job whose status is ready and sets its status to running. If no job is in the ready status, the scheduler is set to idle. The *forall rule* at line 16 and the *conditional rule* at line 17 evaluate all jobs whose status is running and set their status to finished if the monitored function fin indicates completion. Note that, since the *rules* executed within a *par* block are evaluated on the same previous state, r_SetFinished cannot finish a job that is set to running by r_SetRunning in the same step. Each update to the controlled functions idle and st is performed when an *update rule*, such as the one at line 13 in Figure 1, is fired.

Given a specification such as the scheduler, users can validate its expected behavior through `Avalla` scenarios. An example of `Avalla` scenario is shown in Figure 2. It loads the specification against which it is executed, then it contains a set of check, set, pick, and step commands. With the check statement, users can verify whether the values of controlled functions are as expected, with the set command, the value of monitored functions is set, and with the pick command it is possible to force the non-deterministic choice of a *choose rule* to a fixed value, solving the issue of flaky scenarios [10] (i.e., scenarios testing non-deterministic `Asmeta` specifications, including the choose statement). Additionally, the scenario may force the `Asmeta` specification to proceed one step with the step command.

```
scenario scenario1
load Scheduler.asm
check idle = false;
check st(JOB1) = RDY and st(JOB2) = RDY
      and st(JOB3) = RDY;
pick $j1 := JOB3;
step
check st(JOB1) = RDY and st(JOB2) = RDY
      and st(JOB3) = RUN;
set fin(JOB3) := true;
pick $j1 := JOB2;
step
check st(JOB1) = RDY and st(JOB2) = RUN
      and st(JOB3) = FIN;
```

Fig. 2: `Avalla` scenario for the specification in Figure 1

The scenario in Figure 2 first checks that the model starts in the expected initial configuration (see Figure 1, line 21). It then fixes the scheduler's non-deterministic choice of which job to start, executes one step, and verifies that the chosen job becomes running while the others remain ready. Next, it models completion of the running job, fixes the next scheduling choice, and executes a further step. Finally, it checks that the completed job is finished, the newly scheduled job is running, and the remaining job is still ready.

3 Coverage Criteria

This section formally defines the new coverage criteria we introduce in this work. It then describes their implementation in `AsmetaV` and illustrates their computation through an example. Finally, the section discusses criteria infeasibility and presents subsumption relations among the proposed criteria.

[2] The value of a *monitored* function is set by the environment and read by the machine.

In Definition 1, we introduce the notation required to describe the execution of a scenario. Building on this definition, we formally define the coverage of *rules* (Definition 2), *update rules* (Definition 3), *forall rules* (Definition 4), and *branches* (Definition 5). In the following definitions, the transition *rules* under consideration are limited to *par*, *seq*, *choose*, *conditional*, *extend*, *forall*, *let*, *macro call*, *skip*, and *update rules*. Among these, *conditional*, *choose*, and *forall rules* introduce guarded branches. For both *choose* and *forall rules*, the guard is considered satisfied if there exists at least one combination of elements from the considered domains for which the condition evaluates to *true*.

Definition 1 (Scenario execution). *The execution of a scenario S over an Asmeta specification A is the finite sequence of states $s_0, s_1, ..., s_n$ of A produced during the validation.*

Definition 2 (Rule coverage). *Given a macro rule R defined in the Asmeta specification A, let $\#r_{fired}(R)$ be the number of rules in R that fired at least once during the execution of S, and $\#r_{total}(R)$ the number of rules in R. The coverage of the rules in R is defined as :*

$$cov_{rule}(R) = \frac{\#r_{fired}(R)}{\#r_{total}(R)}$$

Definition 3 (Update rule coverage). *Given a macro rule R defined in the Asmeta specification A, let $\#u_{nontrivial}(R)$ be the number of update rules in R that fired at least once with a non-trivial update set[3] during the execution of S, and $\#u_{total}(R)$ the number of update rules in R. The coverage of the update rules in R is defined as:*

$$cov_{update}(R) = \frac{\#u_{nontrivial}(R)}{\#u_{total}(R)}$$

Definition 4 (Forall rule coverage). *Given a macro rule R defined in the Asmeta specification A, let $\#f_{zero}(R)$, $\#f_{one}(R)$, and $\#f_{mult}(R)$ be the number of forall rules in R that fired with zero, exactly one, and more than one combination of elements satisfying its condition during the execution of S, respectively. Let $\#f_{total}(R)$ be the number of forall rules in R. The coverage of the forall rules in R is defined as:*

$$cov_{forall}(R) = \frac{\#f_{zero}(R) + \#f_{one}(R) + \#f_{mult}(R)}{3 \cdot \#f_{total}(R)}$$

Definition 5 (Branch coverage). *Given a macro rule R defined in the Asmeta specification A, let $\#b_{true}(R)$ and $\#b_{false}(R)$ be the number of rules introducing branches in R whose guard is satisfied or not satisfied during the execution of S, respectively. Let $\#b_{total}(R)$ be the number of rules introducing branches in R. The coverage of the branches in R is defined as:*

[3] We say that an update is trivial in Asmeta if the location is updated to the value that it already has.

$$cov_{branch}(R) = \frac{\#b_{true}(R) + \#b_{false}(R)}{2 \cdot \#b_{total}(R)}$$

These coverage definitions naturally extend from a single *macro rule* R to the entire `Asmeta` specification A, by considering all transition *rules* occurring in all *macro rules* defined in A. The coverage of *macro rules*, which was the only criterion available prior to this work, is defined at the specification level as the ratio between the number of *macro rules* defined in A that are called during the execution of S and the total number of *macro rules* defined in A.

The proposed coverage metrics are conceptually inspired by common code coverage criteria. In particular, *rule* coverage corresponds to statement coverage, *forall rule* coverage corresponds to loop coverage, and *branch* coverage mirrors traditional branch coverage. In addition, we introduce *update rule* coverage, which is specific to `Asmeta`, as *update rules* are the constructs responsible for modifying the values of locations and thus for producing state changes during scenario execution.

As described by [15], `AsmetaV` validates an `Asmeta` specification by translating an `Avalla` scenario S into a temporary model that executes according to S, replacing the original main rule with a wrapper that steps through the original execution. The resulting specification is then simulated, and `AsmetaV` tracks the macro rules executed. To introduce the coverage criteria mentioned above, we extend this process by additionally recording executed *rules*, *update rules*, *forall rules* and *branches*. This enables the computation of the coverage metrics defined above. From an implementation perspective, simulation relies on the reflective visitor pattern. While previous versions only overrode the visit of MacroCallRule, this work extends the visitor to all other considered *rules* and *branches*, providing a basis for future coverage metrics over additional rule types.

In the following, we provide an example of how all coverage measures listed above are computed with reference to the example in Figure 1.

Example We illustrate how the proposed criteria are computed at the level of an `Asmeta` specification using the scenario in Figure 2, starting from *macro rule* coverage. Since the *main rule* r_Main calls both r_SetRunning and r_SetFinished, all *macro rules* are fired, yielding full *macro rule* coverage. The specification in Figure 1 contains a total of 9 transition *rules*: 1 *par*, 2 *macro calls*, 1 *forall*, 1 *conditional*, 1 *choose*, and 3 *updates*. During the scenario execution, all *rules* fire at least once except the update at line 14. Consequently, *rule* coverage is $8/9 = 0.89$. Among the three *update rules*, only two fire at least once with a non-trivial update set; therefore, the *update rule* coverage is $2/3 = 0.67$. The specification includes one *forall rule*, which is observed with zero iterations in the first step (as no job is in the RUN state) and with exactly one iteration in the second step (corresponding to a single running job). As a result, the *forall rule* coverage is $(1+1+0)/(3 \cdot 1) = 0.67$. Finally, three *rules* introduce branches: the *choose*, the *forall*, and the *conditional rule*. During the execution of the scenario, the guard of the *choose rule* is always satisfied (there is always at least one

ready job), the *conditional rule* is observed only in the true case (after setting fin(JOB3):= true), and the forall rule is fired both with and without iterations. Therefore, the resulting *branch* coverage is $(3 + 1)/(2 \cdot 3) = 0.67$.

Infeasibility In practice, given a *macro rule* R in an Asmeta specification A, it may be infeasible to achieve full coverage according to a given coverage criterion. This may occur for two reasons. First, some transition *rules* in R may never fire due to semantic constraints in A, resulting in dead *rules*. Second, *branch* coverage may be infeasible because guards may always be satisfied or unsatisfied; similarly, *forall rule* coverage may be infeasible if there are always either zero, exactly one, or multiple combinations of elements in a *forall rule* for which the condition is *true*. In the proposed coverage criteria, coverage is computed with respect to all transition *rules* and *branches* syntactically defined in the specification. As a result, uncovered elements may represent either untested behavior or behavior that is infeasible to exercise, and their interpretation requires manual analysis or further analysis with the model reviewer AsmetaMA [6].

Subsumptions There exist subsumption relations among the coverage criteria defined in this section and the *macro rule* coverage. We adopt the standard notion of subsumption: a coverage criterion $C1$ is subsumed by another criterion $C2$ if achieving full coverage according to $C2$ implies achieving full coverage according to $C1$. We denote this relation by $C1 \subseteq C2$.

Considering only transition *rules* of the types considered in the definition of the criteria, exercising all guarded branches implies exercising all *rules*. Therefore, we have:

$$\textit{rule} \text{ coverage} \subseteq \textit{branch} \text{ coverage}$$

Moreover, if for each *macro* rule defined in an Asmeta specification A there exists at least one *macro call* rule invoking it, we also have:

$$\textit{macro rule} \text{ coverage} \subseteq \textit{rule} \text{ coverage}$$

Note that *update rule* coverage is not subsumed by either *rule* coverage or *branch* coverage, since an *update rule* may fire but produce a trivial update set.

4 Mutation Operators

Mutation testing is a software testing technique in which small and systematic changes are introduced into the program's source code to create mutants that simulate potential faults. A test suite is considered effective at detecting such faults if it causes at least one mutant to fail, indicating it would likely detect a similar fault in the original program [36]. In this case, we say that the mutant is *killed*. To assess the effectiveness of the code coverage extension presented in this paper, we adopt the *mutation score*, defined as the ratio between the number of killed mutants and the total number of mutants generated for each Asmeta specification.

As commonly done in the literature [4,29], we employ mutation analysis both to assess the coverage measures proposed in this paper and to gain further insight into the effectiveness of `Avalla` scenarios [29]. More specifically, we apply eight mutation operators to the original `Asmeta` specifications, where each application of a mutation operator produces at least one distinct mutated model. These mutants are then used to assess the fault detection capability of different `Avalla` scenarios and explore the correlation between coverage and mutation score. The mutation operators adopted in this work are inspired by a previous study on mutation testing for `Asmeta` models [3]. Unlike that work, which focuses on defining mutation operators without providing tool support or source code, we implement and apply these operators within our framework. The set of mutation operators considered in this paper is described below.

- *Rule remover*: It mutates the model by replacing a rule with a skip statement.
- *ChooseRule remover*: It mutates a ChooseRule into a LetRule, where instead of randomly selecting the variable during the simulation, it is deterministically assigned in the model, removing that source of non determinism from the specification [10].
- *Condition remover*: It alternately removes the condition of a ConditionalRule, generating two distinct models: one in which only the rule in the then branch is kept, and another in which only the rule in the else branch is kept.
- *Condition negator*: It swaps the block of the then rule in the ConditionalRule with the block of the else rule. In case the else rule is empty, a skip rule is used.
- *Forall rule mutator:* It modifies the guard of the ForallRule, generating three new cases: a case with the negation of the guard; another where the guard changes to true; and the last one with the guard changed to false.
- *Parallel to Sequential:* It replaces a `par` block with a `seq` block.
- *Sequential to Parallel:* It replaces `seq` block with a `par` block.
- *Case mutator:* It creates new models where, in each one, a case block defined in a CaseRule (and the associated rules) is removed. If the `otherwise` statement is used, this mutation operator produces a model without it.

Figure 3 provides an illustrative example of the *Forall rule mutator* applied to the simple scheduler specification introduced in Section 2. The mutation of the *forall rule* generates three additional models as follows: a) the guard negation, b) the guard is set to true, c) the guard is set to false.

Let us consider the `Avalla` scenario shown in Figure 2. Restricting the scenario to the first step and the subsequent check is sufficient to achieve full *macro rule* coverage. However, under this restriction, the third mutant presented in Figure 3, in which the guard is set to false, is not killed. This illustrates that macro rule coverage alone, for the specification shown in Figure 1, may be insufficient to reveal certain faults. In contrast, combining different coverage criteria, such as achieving also the *forall rule* coverage, would guide the tester toward writing a more comprehensive scenario with higher fault detection capability. Specifically, when only the first step in considered, the *forall* is executed with zero iterations

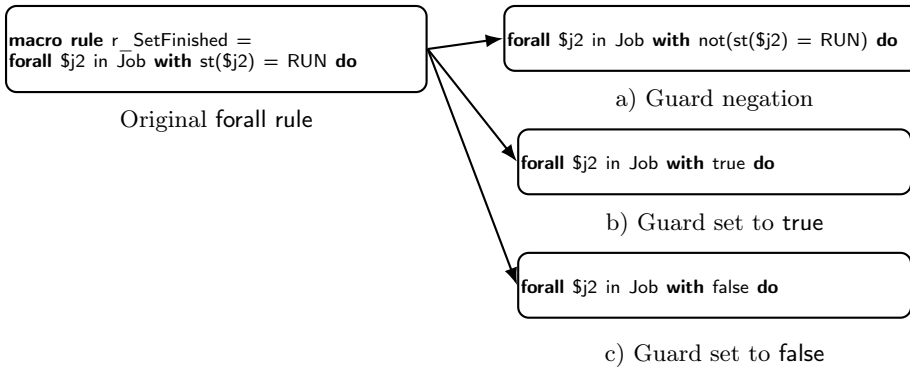


Fig. 3: Application of the *Forall rule mutator* within the r_SetFinished rule in the scheduler example

and the *forall rule* coverage is $1/3 = 0.33$. When the full scenario is considered, *forall rule* coverage increase to $2/3 = 0.67$, as the forall is executed with exactly one iteration. In that case, all three mutants are killed. Note that, although the presented scenario is sufficient to expose the fault, achieving full *forall rule* coverage would require writing an additional scenario with at least three steps.

5 Evaluation

In this section, we evaluate Avalla scenarios in terms of different coverage metrics and we relate them with fault detection capability. We first introduce the methodology we followed in our experiments (Section 5.1); then, we present our results (Section 5.2).

5.1 Methodology

This section describes our experimental methodology. Our aim is to evaluate Avalla scenarios in terms of their coverage, by using the different coverage criteria we presented in Section 3 and to investigate the correlation between coverage and fault detection capability. To perform such an evaluation, we identified the following research questions (RQs):

RQ1 Do higher coverage test suites lead to higher fault detection capability?
RQ2 Do finer-grained coverage criteria lead to test suites with higher fault-detection capability than coarser criteria?

While we consider these two research questions to be appropriate for assessing the relationship between coverage measures and fault detection capability, we acknowledge that additional experiments are required to account for the effect of test suites size in terms of number of scenarios and scenario length (i.e., number

of executed steps). In particular, it can be argued that a substantial portion of the impact on fault detection capability may stem not from the coverage achieved by an `Avalla` scenario over an `Asmeta` specification, but rather from the test suite size itself. For this reason, in order to rule out the possibility that improved fault detection capability is primarily due to increased test suite size rather than enhanced coverage, we devised the following research question:

RQ3 Do higher coverage scenarios, with the same test suite size, lead to higher fault detection capability?

We performed our experiments on a PC with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (16 physical cores, 32 logical cores) with 256 GB RAM, running Ubuntu 20. The results we obtained and the scripts for the replication of our experiments are available online [7].

In our evaluation, we selected all `Asmeta` specifications available in the official `Asmeta` GitHub repository[4], excluding duplicates and specifications marked as "old" or written solely for testing purposes. The resulting benchmark is composed of 297 specifications. For RQ1 and RQ2, we generate `Avalla` scenarios by using three different approaches: `ATGT` [24], random exploration, and EvoAvalla. `ATGT` is based on the use of the NuSMV model checker [18]. It supports forward symbolic model checking that explores a system's state space forward (from initial states towards future states) using symbolic representations in order to reach the desired coverage [23], with all the limits inherited from this technique, and generates `Avalla` scenarios. Random exploration exploits the random simulation offered by `AsmetaS` and records monitored and controlled function values, which are saved as `Avalla` scenarios. This process requires user to specify the number of steps in each scenario, and the number of tests. In our experiments, we generate random scenarios iteratively: starting from 10 scenarios with 10 steps each, we increase the number of steps and tests iteratively by step 10. With this process, we stop when coverages and mutation scores do not improve between two subsequent iterations or the 100 seconds time budget elapsed. Finally, EvoAvalla is a prototypical test generation strategy[5]. By starting from `Avalla` specifications, it translates them into Java code [9], it exploits EvoSuite [20] to generate JUnit test cases, which are abstracted back as `Avalla` scenarios. Note that, for each test generation strategy, there exists a subset of specifications for which it is not possible to generate a valid and non-empty test suite.

For RQ3, to ensure fair comparison with the same test length, we only compare `ATGT` with random scenario generation and we set random scenarios with the same size and numbers as those generated by `ATGT`.

For each test execution, we recorded the name of the `Asmeta` specification, as well as its characteristics (the total number of *rules*, the number of *macros* and *update rules*, the number of *branches*, and the number of *choose* and *forall rules*), the execution time, and the number of generated scenarios. We also recorded

[4] `https://github.com/asmeta/asmeta/tree/master/asm_examples`

[5] `https://github.com/asmeta/asmeta/tree/master/code/experimental/asmeta.evotest/asmeta.evotest.evoavalla`

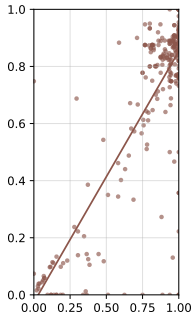


Fig. 4: Mean Coverage vs. Mutation Score for Avalla scenarios on Asmeta specifications

Coverage Criterion	# Test Suites	Regression Slope β	R^2	p-value (β)	Spearman ρ	p-value (ρ)
Mean coverage	272	0.88	0.63	3.78e-60	0.60	3.17e-28
Macro rule coverage	272	0.83	0.53	8.40e-46	0.56	5.84e-24
Rule coverage	272	0.85	0.66	1.43e-64	0.64	2.90e-33
Update rule coverage	272	0.75	0.67	1.38e-66	0.66	1.08e-35
Forall rule coverage	45	1.12	0.27	2.75e-04	0.47	1.19e-03
Branch coverage	269	0.79	0.55	5.22e-48	0.55	6.18e-23

Table 1: Linear regression and Spearman correlation between coverage criteria and mean mutation score

the characteristics of the generated test suite (total number of sets, steps, and checks), the achieved coverage (for all criteria reported in Section 3), and the mutation score (both per mutation operator and as an overall aggregated score across all operators reported in Section 4).

5.2 Results

In the following, we describe the results of our experiments and answer the RQs.

RQ1: Coverage value and fault detection capability In this section, we investigate whether there is a correlation between the coverage of Asmeta entities and the fault detection capability. The rationale behind this RQ is that this correlation is well-studied and, in general, confirmed for code [26, 40], but there has been little focus on formal models.

To answer this RQ, as previously mentioned, we considered all Asmeta models in the official Asmeta GitHub repository, and we generated test suites (i.e., sets of Avalla scenarios) with three test generation approaches. Figure 4 shows a scatter plot relating the mean coverage (including all coverage criteria) of the test suites generated by each test generation strategy for each specification (x-axis) to their fault detection capability (y-axis), measured in terms of mutation

score across all operators. Thus, each point in Figure 4 represent a test suite $ts_{A,S}$ generated by the approach A for the `Asmeta` specification S. Additionally, we report the regression line obtained from a linear regression fit. The scatter plot highlights a clear positive correlation between mean coverage and fault detection capability: test suites achieving higher coverage generally exhibit higher fault detection rates. In particular, the points are concentrated in the upper-right region of the plot, indicating that high coverage is often associated with strong fault detection performance. Table 1 reports both the linear and monotonic correlations between individual coverage criteria and mean coverage with the mutation score. Specifically, we report the number of test suites for which the coverage criterion and at least one mutation operator are defined, the slope of the regression line, the coefficient of determination (R^2) and the p-value of the linear regression. In addition, we report Spearman's rank correlation coefficient (ρ) and it's associated p-value. This analysis is motivated by the fact that several data points saturate at full coverage, which may introduce heteroskedasticity and reduce the reliability of linear regression alone. The qualitative correlation observed in Figure 4 is quantitatively confirmed by the first row of Table 1, where an R^2 of 0.63 indicates a moderate correlation between mean coverage and mutation score.

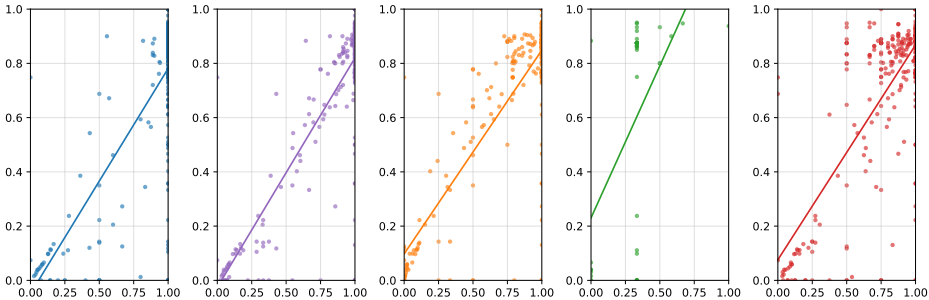
Similar observations hold when considering each coverage criteria separately. Figure 5 shows such an analysis. Except for the *forall rule* coverage, that is defined for substantially fewer test suites compared to other criteria, similar trends are confirmed. The results in the R^2 and Spearman ρ columns in Table 1 confirm moderate correlation between coverage and fault detection capability.

RQ1 - Answer

Our results show that there exist a moderate correlation between coverage and mutation score: testers should aim for higher coverage on `Asmeta` entities, which should lead to improved fault detection capability.

RQ2: Coverage criteria and fault detection capability In this section, we investigate whether there is a correlation between using a more fine-grained coverage criterion and the fault detection capability, as it has been shown for code [2, 34].

As for the previous RQ, we consider all `Asmeta` models in the official `Asmeta` GitHub repository and use the same test suites (i.e., sets of `Avalla` scenarios) we considered for RQ1. The comparison between different criteria is reported in Figure 5 and Table 1. To quantitatively compare the different criteria, we consider the *Slope* of the regression line, which provides an intuitive estimate of how much mutation score increases as coverage increases. The slope is significantly different from zero for all criteria reported in Table 1 (p-value of $\beta < 0.05$), indicating a positive association. The *forall rule* coverage is defined for substantially fewer test suites than the other criteria (45 vs ≈270) and shows the weakest relationship with mutation score ($R^2 = 0.27$, $\rho = 0.47$). Therefore, conclusions about



(a) *Macro rule* Coverage (b) *Rule* Coverage (c) *Update rule* Coverage (d) *Forall rule* Coverage (e) *Branch* Coverage

Fig. 5: Coverage Criteria vs. Mutation Score for Avalla scenarios on Asmeta specifications

this criterion should be interpreted with caution due to limited applicability and lower explanatory power.

Surprisingly, the most fine-grained criterion, namely *branch* coverage, does not exhibit the highest slope. Although branch coverage is theoretically more fine-grained, in practice many branches are guarded by conditions that are rare or infeasible to realize in generated scenarios. As a consequence, many test suites achieve high mutation scores while still failing to reach high (or full) *branch* coverage. This decoupling reduces the degree to which mutation score co-varies with *branch* coverage, leading to a smaller fitted slope and weaker association. The results reported in Table 1 confirm that *rule* coverage, i.e., the second most fine-grained coverage criterion, yields the highest slope and, consequently, the greatest fault detection capability. This is followed by *macro rule* coverage and *branch* coverage. While *macro rule* coverage may appear less fine-grained, a *macro rule* typically encapsulates multiple transition rules; thus, covering a *macro* often implies exercising a non-trivial portion of the specification rather than a single atomic element. Finally, the weakest criterion, *update rule* coverage, which targets only a specific type of atomic rule, exhibits the lowest slope.

RQ2 - Answer

Our experiments indicate that finer-grained coverage criteria generally lead to higher fault detection capability. However, some of these criteria are not consistently achieved across all specifications, which may introduce bias into the results and affects the comparative ranking of the criteria. The best coverage criterion is the *rule* coverage, while the worst one is the *update rule* coverage.

RQ3: Coverage value and fault detection capability with same test suite size. In this section, we investigate whether correlation exists between the coverage of Asmeta entities and fault detection capability when the test

Coverage Criterion	# Test Suites	W	p-value (W)
Mean coverage	39	632	7.46e-05
Macro rule coverage	18	170	7.63e-06
Rule coverage	34	530	3.52e-05
Update rule coverage	36	573	8.14e-05
Forall rule coverage	0	-	-
Branch coverage	34	792	7.87e-05

Table 2: Wilcoxon signed-rank test on mutation score comparing ATGT and random test suites of equal size

suite size is kept equal across different test generation strategies. The rationale behind this research question is to eliminate the influence of larger test suites on the observed fault detection capability.

We considered all 51 Asmeta specifications that ATGT can handle and generated test suites (i.e., sets of Avalla scenarios) using both ATGT and random exploration, while keeping the test suite size equal across different test generation strategies. This resulted in two sets of test suites: one composed of scenarios generated by ATGT, which consistently achieve higher or equal coverage than the other, composed of randomly generated scenarios.

Since all test suites for a given specification have the same size, differences in fault detection capability can be attributed primarily to differences in the coverage achieved by the test suites. For this reason, we compare the two sets by using the Wilcoxon Signed-Rank test [42], a general test that compares the distributions in paired samples and that does not require data to be normally distributed. The test is conducted under the null hypothesis that the median paired difference in mutation score is not positive (i.e., median ≤ 0), against the alternative hypothesis that ATGT yields higher values (median > 0). To focus on cases where the two approaches differ in achieved coverage, we restrict the analysis to the specifications for which ATGT attains strictly higher coverage than random exploration. We perform this analysis for each coverage criterion as well as for mean coverage.

The obtained results are shown in Table 2. It reports the number of specifications for which we could compute the specific coverage criterion and for which ATGT attains strictly higher coverage than random exploration, the Wilcoxon statistics W, and the p-value. All tests are significant (p-value < 0.05), and this allows us to reject the null hypothesis and confirm that the improved fault detection capability comes from higher coverage. Note that, also in this case, the *forall rule* coverage is not considered. Indeed, none of the Asmeta specifications that ATGT can handle includes *forall* rules.

RQ3 - Answer

Our experiments show that, when the effect of test suite size is ruled out, the correlation between coverage and fault detection capability is confirmed.

6 Threats to Validity

In this section, we discuss potential threats to the validity [38] of our work and the actions we have taken to mitigate them.

Internal validity concerns the varying outcomes observed with the techniques and tools under analysis. It refers to the fact that the results are genuinely due to the different approaches and the experimental methods used, rather than resulting from methodological errors. To minimize this risk, we thoroughly reviewed the code used in our experiments, looking for potential alternative factors that could have influenced the results, such as tool or experimental code errors.

A possible threat to *construct validity* stems from the assumption that the metrics we selected to evaluate the results of our experiments are adequate for proper assessment. Indeed, in our experiments, we considered test suite size, coverage, and mutation score, while we ignored other factors that could influence the outcome. For example, one may also consider the number of sets and checks in Avalla scenarios. However, given the theoretical assumptions on ASMs, the internal state of the machine is updated at every step and not at every check. So, we believe that considering just the test suite size (i.e., the number of scenarios and steps in each of them) is enough. We may better investigate the impact of alternative factors in future work. As commonly done in the literature [36], we use the mutation score, defined as the ratio of killed mutants to the total number of mutants, as a proxy for fault detection capability. In this analysis, we did not account for equivalent mutants, which may affect the reported results; we plan to investigate their impact in future work.

External validity is concerned with whether it is possible to generalize the results outside the scope of the presented benchmarks and tools. Concerning the benchmarks used for RQ1 and RQ2, we conducted our experiments on all the specifications available in the official Asmeta repository, aiming to cover as many features of the Asmeta language as possible. One could argue that, for RQ3, we considered only two test generation approaches (random-based and ATGT) and only the subset of benchmarks for which ATGT was applicable, while excluding EvoAvalla. We acknowledge this as a potential limitation; however, this choice was made to minimize the influence of different test generation strategies and to concentrate on the correlation between coverage and fault detection capability, using test suites of the same size, which was the primary objective of RQ3. We do not claim that our findings can be generalized to other state-based formal notations. However, the formal definition of coverage criteria can be used to conduct similar experiments with other notations.

In addition, our analysis considered only automatically generated scenarios, which are derived from the model rather than from requirements. As future work, we plan to include scenarios written by engineers to better assess the correlation between coverage and fault detection capability in realistic settings. Finally, our experiments do not consider *flaky* tests [35], which may also arise in Asmeta [10]. To mitigate their occurrence, we executed scenarios deterministically; however, additional experiments are needed to assess whether our conclusions extend to non-deterministic ASMs.

7 Related Work

Coverage analysis is essential in both software testing and formal model validation to ensure the thoroughness of testing efforts and the reliability of system specifications. In traditional software development, coverage metrics such as statement, branch, and path coverage are widely adopted to assess the effectiveness of test suites and to guide test design [43]. These metrics are commonly supported by mature tooling, including JaCoCo [28] for Java and gcov [1] for C/C++, which enable the systematic measurement of test adequacy at the code level. Coverage metrics have also been studied in the context of functional programming, along with their correlation with test effectiveness [16], and supported by dedicated tools such as HPC for Haskell [25].

At the model level, several coverage criteria have been proposed to evaluate the completeness of validation and testing activities performed on formal and semi-formal specifications. Existing approaches address, among others, B, Z and UML class diagrams with OCL specifications [32], VDM specifications [19], UML Statecharts [37], and Finite State Machines [13]. These criteria mainly focus on state coverage [39], transition coverage [13, 14], boundary values [32], or trace-based adequacy notions. Scenario-based and use-case-driven validation has also been widely investigated, particularly in the context of message sequence charts and UML behavioral models, where coverage is evaluated in terms of exercised interactions or execution traces [21, 41]. While these approaches provide useful adequacy measures, they are typically defined over state-transition abstractions and do not directly account for the syntactic and semantic structure of rule-based executable specifications. In formal hardware verification, coverage has also been extensively studied. Chockler et al. [17] adapt coverage notions from simulation-based verification to the formal setting, introducing corresponding coverage metrics that are based on mutations of the model. In contrast, we derive coverage from scenario execution rather than model mutations.

In the context of Abstract State Machines, validation and testing have been extensively supported by tooling such as ASMETA and its associated model execution, simulation, and verification components [12]. Several works have investigated testing techniques for ASMs, including test generation strategies and execution-based validation [22]. However, coverage analysis for ASMs has received comparatively limited attention. Previous research has proposed *macro rule* coverage as a means of assessing whether high-level rules of an ASM specification are exercised during execution [15]. Beyond this macro-level perspective, fine-grained coverage criteria capturing the execution of internal rule constructs, control structures, and update behaviors have not been systematically explored.

Mutation testing is a well-established technique for evaluating the fault detection capability of test suites by introducing artificial faults into the system under test [30]. Although mutation testing was originally conceived for implementation-level artifacts, it has also been successfully applied in model-based development, where faults are injected into behavioral or specification models rather than source code. Mutation analysis has been used to assess the adequacy of test suites for UML state machines, formal specifications, and other model-based

representations [14, 32, 33]. More generally, mutation testing is widely regarded as a complementary adequacy criterion that enhances structural coverage measures by providing empirical evidence of fault detection capability [36].

With respect to Abstract State Machines (ASMs), mutation operators targeting rules, guards, and control structures have been proposed [3]. However, this line of work primarily focuses on the definition of mutation operators, without investigating the relationship between mutation results and coverage metrics or validation effectiveness. Moreover, the authors did not provide the source code of their tool and we could not integrate easily in our experiments.

To the best of our knowledge, the work presented in this paper is the first to introduce fine-grained, rule-level coverage criteria for the validation of Asmeta specifications and to empirically study their effectiveness in a scenario-based setting using mutation-based fault detection. By combining structural coverage analysis with mutation testing, our approach provides a more comprehensive assessment of scenario adequacy for executable ASM specifications.

8 Conclusion

In this paper, we investigated the relationship between coverage and fault detection capability in the context of scenario-based validation of Asmeta specifications. While coverage has long been used as an adequacy measure in software testing for source code, its effectiveness at the level of executable formal specifications had not been previously studied in depth. To address this gap, we extended AsmetaV with a set of coverage criteria that go beyond traditional *macro rule* coverage, and we assessed their effectiveness through a large-scale mutation-based experimental study.

Our empirical results show that higher coverage is generally associated with increased fault detection capability in scenario-based validation of Asmeta models. More fine-grained criteria, such as *rule* coverage coverage, exhibit a stronger correlation with mutation scores and lead to improved fault detection. We also analyzed the impact of coverage granularity and shown that, although more fine-grained criteria tend to be more effective, their benefits may be limited by infeasible or rarely achievable coverage targets. Among the criteria we introduced, *rule* coverage emerged as the most effective and robust indicator of fault detection capability, while *update rule* coverage showed the lowest fault detection capability. As future work, we plan to extend the set of supported coverage criteria to include domain-aware and semantically richer criteria, further investigate infeasible coverage elements, and study the impact of non-determinism and flaky scenarios on coverage-driven validation effectiveness.

Acknowledgments. The work of Andrea Bombarda has been funded by ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 - CUP: B53C22006700001. We acknowledge the financial support of the project PRIN 2022 SAFEST (Trust assurance of Digital Twins for medical cyber-physical systems), funded by the European Union - Next Generation EU, Mission 4, Component 2, Investment 1.1, CUP F53D23004230006, under the National Recovery and Resilience Plan (NRRP).

References

1. Gcov: Gnu coverage tool. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
2. Aghamohammadi, A., Mirian-Hosseinabadi, S.H., Jalali, S.: Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness. Information and Software Technology **129**, 106426 (2021). https://doi.org/https://doi.org/10.1016/j.infsof.2020.106426
3. Alkrarha, O., Hassine, J.: Muasmetal: An experimental mutation system for asmetal. In: Latifi, S. (ed.) 12th International Conference on Information Technology - New Generations, ITNG 2015, Las Vegas, NV, USA, April 13-15, 2015. pp. 421–426. IEEE Computer Society (2015). https://doi.org/10.1109/ITNG.2015.74
4. Andrews, J., Briand, L., Labiche, Y., Namin, A.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering **32**(8), 608–624 (2006). https://doi.org/10.1109/TSE.2006.83
5. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems. In: Logic, Computation and Rigorous Methods - Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 12750, pp. 215–238. Springer (2021). https://doi.org/10.1007/978-3-030-76020-5_13
6. Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of Abstract State Machines by meta property verification. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215. pp. 4–13. NASA, Langley Research Center, Hampton VA 23681-2199, USA (April 2010)
7. Bombarda, A., Bonfanti, S., Cornejo, C., Gargantini, A., Pellegrinelli, N.: Replication Package for "Evaluating coverage and fault detection capability of scenarios for the validation of Asmeta specifications" (2026). https://doi.org/10.5281/zenodo.18270328
8. Bombarda, A., Bonfanti, S., Gargantini, A.: Developing medical devices from abstract state machines to embedded systems: A smart pill box case study. In: Mazzara, M., Bruel, J.M., Meyer, B., Petrenko, A. (eds.) Software Technology: Methods and Tools. pp. 89–103. Springer International Publishing, Cham (2019)
9. Bombarda, A., Bonfanti, S., Gargantini, A.: From Concept to Code: Unveiling a Tool for Translating Abstract State Machines into Java Code, p. 160–178. Springer Nature Switzerland (2024). https://doi.org/10.1007/978-3-031-63790-2_10
10. Bombarda, A., Bonfanti, S., Gargantini, A., Pellegrinelli, N.: Eliminating flakiness: Deterministic control for validating nondeterministic asmeta specifications. In: Dutle, A., Humphrey, L., Titolo, L. (eds.) NASA Formal Methods. pp. 100–115. Springer Nature Switzerland, Cham (2025)
11. Bombarda, A., Bonfanti, S., Gargantini, A., Pellegrinelli, N., Scandurra, P.: Safety enforcement for autonomous driving on a simulated highway using asmeta models@run.time. In: Leuschel, M., Ishikawa, F. (eds.) Rigorous State-Based Methods. pp. 212–230. Springer Nature Switzerland, Cham (2026)
12. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: ASMETA tool set for rigorous system design. In: Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14934, pp. 492–517. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_28
13. Bombarda, A., Gargantini, A.: An Automata-Based Generation Method for Combinatorial Sequence Testing of Finite State Machines. In: 2020 IEEE International

Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 157–166 (2020). https://doi.org/10.1109/ICSTW50294.2020.00036
14. Briand, L., Labiche, Y., Lin, Q.: Improving the coverage criteria of UML state machines using data flow analysis. Software Testing, Verification and Reliability **20**(3), 177–207 (Apr 2009). https://doi.org/10.1002/stvr.410
15. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A Scenario-Based Validation Language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) Abstract State Machines, B and Z. pp. 71–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7
16. Cheng, Y., Wang, M., Xiong, Y., Hao, D., Zhang, L.: Empirical Evaluation of Test Coverage for Functional Programs. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 255–265 (2016). https://doi.org/10.1109/ICST.2016.8
17. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for formal verification. In: Geist, D., Tronci, E. (eds.) Correct Hardware Design and Verification Methods. pp. 111–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_11
18. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking, p. 359–364. Springer Berlin Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
19. Fitzgerald, J., G. Larsen, P., Sahara, S.: Vdmtools: advances in support for formal modeling in vdm. SIGPLAN Not. **43**(2), 3–11 (Feb 2008). https://doi.org/10.1145/1361213.1361214
20. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. p. 416–419. ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2025113.2025179
21. Fraser, G., Wotawa, F.: Using model checkers to generate test cases for dynamic systems. In: Proceedings of the International Conference on Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART). pp. 3–12. IEEE (2007)
22. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 146–162. ESEC/FSE-7, Springer-Verlag, Berlin, Heidelberg (1999)
23. Gargantini, A., Riccobene, E.: Asm-based testing: Coverage criteria and automatic test sequence. JUCS - Journal of Universal Computer Science **7**(11), 1050–1067 (2001). https://doi.org/10.3217/jucs-007-11-1050
24. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to Generate Tests from ASM Specifications, p. 263–277. Springer Berlin Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6_15
25. Gill, A., Runciman, C.: Haskell program coverage. In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. p. 1–12. Haskell '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1291201.1291203, https://doi.org/10.1145/1291201.1291203
26. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M.A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: Proceedings of the

2013 International Symposium on Software Testing and Analysis. p. 302–313. ISSTA 2013, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2483760.2483769
27. Hemmati, H.: How effective are code coverage criteria? In: 2015 IEEE International Conference on Software Quality, Reliability and Security. pp. 151–156 (2015). https://doi.org/10.1109/QRS.2015.30
28. Hoffmann, M.R., Janiczak, B., Mandrikov, E., Friedenhagen, M.: Jacoco code coverage tool (2009)
29. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering. p. 435–445. ICSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568271
30. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
31. Kemmerer, R.: Testing formal specifications to detect design errors. IEEE Transactions on Software Engineering **SE-11**(1), 32–43 (1985). https://doi.org/10.1109/TSE.1985.231535
32. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: 15th International Symposium on Software Reliability Engineering. pp. 139–150 (2004). https://doi.org/10.1109/ISSRE.2004.12
33. Li, J.h., Dai, G.x., Li, H.h.: Mutation analysis for testing finite state machines. In: 2009 Second International Symposium on Electronic Commerce and Security. vol. 1, pp. 620–624 (2009). https://doi.org/10.1109/ISECS.2009.158
34. Li, N., Praphamontripong, U., Offutt, J.: An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops. pp. 220–229 (2009). https://doi.org/10.1109/ICSTW.2009.30
35. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT/FSE'14, ACM (Nov 2014). https://doi.org/10.1145/2635868.2635920
36. Papadakis, M., Shin, D., Yoo, S., Bae, D.H.: Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: Proceedings of the 40th International Conference on Software Engineering. p. 537–548. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180183
37. Pradhan, S., Ray, M., Swain, S.K.: Transition coverage based test case generation from state chart diagram. J. King Saud Univ. Comput. Inf. Sci. **34**(3), 993–1002 (Mar 2022). https://doi.org/10.1016/j.jksuci.2019.05.005
38. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14**(2), 131–164 (dec 2008). https://doi.org/10.1007/s10664-008-9102-8
39. Salman, Y.D., Hashim, N.L., Rejab, M.M., Romli, R., Mohd, H.: Coverage criteria for test case generation using UML state chart diagram. vol. 1891, p. 020125 (10 2017). https://doi.org/10.1063/1.5005458
40. Schwartz, A., Puckett, D., Meng, Y., Gay, G.: Investigating faults missed by test suites achieving high code coverage. Journal of Systems and Software **144**, 106–120 (2018). https://doi.org/10.1016/j.jss.2018.06.024

41. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE). pp. 314–323. ACM (2000)
42. Woolson, R.F.: Wilcoxon signed-rank test (Feb 2005). https://doi.org/10.1002/0470011815.b2a15177
43. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (Dec 1997). https://doi.org/10.1145/267580.267590