SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2023; **00**:1–39
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

On Transforming Model-based Tests into Code: A Systematic Literature Review

Fabiano C. Ferrari,[1] Vinicius H. S. Durelli,[2] Sten F. Andler,[3]
Jeff Offutt,[4] Mehrdad Saadatmand,[5] Nils Müllner[6]

[1] *Computing Department – Federal University of São Carlos – Brazil*
[2] *Computer Science Department – Federal University of São João del-Rei– Brazil*
[3] *School of Informatics, University of Skövde – Sweden*
[4] *Department of Computer Science, University at Albany – United States*
[5] *RISE Research Institutes of Sweden, Västerås – Sweden*
[6] *DLR (Deutsche Luft- und Raumfahrt, German Aerospace Center) – Germany*

SUMMARY

Model-based test design is increasingly being applied in practice and studied in research. Model-Based Testing (MBT) exploits abstract models of the software behavior to generate abstract tests, which are then transformed into concrete tests ready to run on the code. Given that abstract tests are designed to cover models but are run on code (after transformation), the effectiveness of MBT is dependent on whether model coverage also ensures coverage of key functional code. In this article, we investigate how MBT approaches generate tests from model specifications and how the coverage of tests designed strictly based on the model translates to code coverage. We used snowballing to conduct a systematic literature review. We started with three primary studies, which we refer to as the initial seeds. At the end of our search iterations, we analyzed 30 studies that helped answer our research questions. More specifically, this article characterizes how test sets generated at the model level are mapped and applied to the source code level, discusses how tests are generated from the model specifications, analyzes how the test coverage of models relates to the test coverage of the code when the same test set is executed, and identifies the technologies and software development tasks that are on focus in the selected studies. Finally, we identify common characteristics and limitations that impact the research and practice of MBT: *(i)* some studies did not fully describe how tools transform abstract tests into concrete tests; *(ii)* some studies overlooked the computational cost of model-based approaches; and (iii) some studies found evidence that bears out a robust correlation between decision coverage at the model level and branch coverage at the code level. We also noted that most primary studies omitted essential details about the experiments. Copyright © 2023 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: model-based testing, test coverage criteria, test case generation, test case transformation, systematic literature review

1. INTRODUCTION

Software engineers apply Model-Driven Development (MDD) [48] and Model-Driven Engineering (MDE) [57] to achieve quality in the design of software products at an abstract level before mixing details of implementation and the complexities of a programming language. The key idea in both MDD and MDE is that the model should define the behavior of software, allowing engineers to abstract away from implementation details. Some researchers [57] suggest that MDD can be seen as a subset of MDE whose main focus is on generating implementations from models. In contrast,

[1]Correspondence to: Rodovia Washington Luis, Km 235, São Carlos, São Paulo - Brazil. E-mail: fcferrari@ufscar.br

MDE employs more elaborate models to support the evaluation of quality attributes such as reliability and security during development and model-driven evolution. A hallmark of MDD and MDE is that models are kept at a level where it is still relatively easy to make large-scale changes without getting bogged down in implementation details [6].

This ability to separate abstraction layers has many benefits, including speeding up the overall development, reducing the effort of making changes, and producing more reliable software. Partly because of the expense of changing software after deployment, MDD tends to be applied more commonly in embedded and real-time systems such as transportation systems and electronic appliances.

On the one hand, if a model is executable, such as models written in executable UML [37] or Simulink[2] [17], then by running the model it is possible to test some of its aspects. Software engineers may also exploit sophisticated model-to-code transformation tools to automatically generate code from the model. On the other hand, some model languages, such as UML statecharts, are not executable and do not have sufficiently defined semantics to support automatic model-to-code transformation. Thus, these models are often transformed into code by hand.

A common application of models is to design test cases. An early example derived test cases from (non-executable) UML statecharts [41]. The test cases covered specific elements of the statechart at the model-level, then were run on the code-level implementation. Subsequent papers referred to test cases defined at the model-level as *abstract tests*, while their corresponding code-level implementations are called *concrete tests*. This concept, called *model-based testing* (MBT) [54], is now used widely throughout the software industry and has led to hundreds of papers exploring various aspects of MBT. A key question is related to coverage. If test cases are designed to cover specific aspects of the model (nodes, edges, logic predicates, etc.), what is the relationship between model coverage and code coverage? Does the code include decisions that were **not** in the model? Are covered elements of the model dispersed into different places in the code? As the code changes over time, how can the tests be kept up to date?

One of the most significant challenges arises due to standards. For example, the US Federal Aviation Administration, the European Union Aviation Safety Agency, and the Transport Canada department require that safety-critical software on commercial airplanes and air-traffic control systems to be tested to a stringent standard [46]. The same standard is often looked to as a goal in other transportation industries, such as trains and automobiles. The coverage requirements in the standard are defined on the code level, not the model. Thus, compliance cannot be based on test cases derived from models. Software companies must show that test cases that run on the code will fill in any "gaps" in coverage introduced by the model-to-code transformation.

These gaps between model coverage and code coverage also make traceability crucial. To measure and ensure code coverage for model-based tests, engineers must be able to trace from model-level element to code-level element, and from model-level coverage to code-level coverage. Research into these crucial questions has been going on since 1990, and this article is the first attempt to catalog and categorize relevant papers. A particular challenge is that these papers have been published in many different conferences and journals, and employed diverging sets of terms. This makes it hard for researchers and practitioners to get a clear idea of the current state of the art.

We have carried out a systematic literature review (SLR), which is a study to identify, select, and critically appraise research to answer clearly formulated questions [26]. Our SLR focuses on scenarios in which abstract models are defined prior to testing and investigates how source code coverage can be gauged from test sets generated based on model-based testing approaches. More specifically, our SLR makes the following contributions:

- it characterizes how test sets generated at the model level are mapped and applied to the source-code level;
- it discusses how tests are generated from the model specifications when MBT is applied;

[2] http://www.mathworks.com/products/simulink.html – accessed in June, 2023.

- it analyzes the relationship between the test coverage of models and the test coverage of the code when the same test set is executed; and

- it provides an overview of all selected studies, including varied classifications applied to them, which is made available online as complementary material.[3]

Our SLR applies a *snowballing* process to search for papers of interest. Snowballing recursively analyzes references cited in related papers, and citations to those papers [58]. In the SLR domain, such papers are termed *primary studies* [26] and typically describe research results from well-founded experimental procedures or from early research approaches. We started our snowballing process with three core primary studies. During three recursions, we analyzed 498 non-duplicate primary studies. after the study selection phases, 33 peer-reviewed primary studies (including the 3 seeds) passed our study criteria, from which 30 were analyzed in this SLR given that they present either original or updated contributions. We categorized the selected studies into several groups. Given our focus on test coverage at the model and code levels, our key categorizations concern whether and how the study addresses the transformation of abstract tests to concrete tests, and the level of traceability of software elements, and the coverage of such elements, across the abstraction levels. Finally, we also categorized the selected studies based on the adopted technologies and the level of automation for test generation.

The remainder of this article is organized as follows. Section 2 summarizes concepts related to software testing, MDE, and model-based testing, and brings a brief discussion regarding test coverage at the model and code levels. Section 3 provides details of our SLR protocol, and the criteria and procedures we adopted to select and analyze the selected studies. Section 4 summarizes the results from our search. Section 5 addresses our research questions. Section 6 discusses threats to the validity of our work, and Section 7 presents prior papers that summarized related literature reviews. Finally, Section 8 presents our conclusions as well as implications and recommendations for future research.

2. BACKGROUND

This section introduces concepts related to model-based testing. We first discuss software testing in general, independent of whether the testing is applied to models or code. Then we discuss concepts related to utilizing models to develop software, and then focus on model-based testing. Finally, we introduce the key issues for testing when transforming abstract, model-based tests to code.

2.1. General Software Testing

We start with general concepts and terms related to software testing [4]. Generally, we view testing as an act of executing some software artifact on inputs designed to assess whether the behavior is as intended. Note that the term *artifact* is intended to include anything that can be executed, including but not limited to code, models, and requirements. The term *system under test* (*SUT*) refers to the artifact being tested. Researchers also specialize this term to particular artifacts such as *module under test*, *method under test*, *predicate under test*, and *clause under test*.

Test inputs are the key input values used to satisfy the requirements for testing. Test inputs are sometimes called *test vectors*. To be able to run the tests, the inputs are usually embedded in automated scripts or methods (such as JUnit[4] methods). Automated tests include additional elements beyond test inputs, including *test oracles* that decide whether the software behaves as intended. Test oracles can be implemented as assertions in JUnit.

[3] https://doi.org/10.5281/zenodo.8113394
[4] http://junit.org – accessed in June, 2023.

DOI: 10.1002/stvr

2.2. Testing Coverage

A common technique is to design test cases that ensure some sort of coverage, on the theory that if some element of the software artifact is not covered, then we do not know whether its behavior is acceptable [4]. The simplest coverage criterion is *node coverage*, which requires that each node in a graph is covered. This equates *e.g.* to *statement overage* if the graph represents code, *state coverage* if the graph represents state machines. Thus, node coverage, state coverage, and statement coverage amount to the same thing. A slightly more strenuous criterion is *edge coverage*, which requires that each edge in a graph is covered. This equates to decision coverage and branch coverage, depending on the type of artifact. Node and edge coverage treat predicates as simple black boxes. Thus, a predicate with three clauses (A && B || C) is only evaluated to true and false, without considering the different clauses. *Modified Condition and Decision Coverage* (MCDC) [11] requires that each individual clause evaluates to both true and false, with the other clauses being such that the clause under test determines the final value of the predicate. Thus, the example predicate p = (A && B || C) can be MCDC-covered with the test set $\{TTF,\ FTF,\ TFF,\ FFT,\ FFF\}$. A final structural coverage criterion is *data-flow coverage*, which requires that definitions of variables (*defs*) reach specific *uses* of those values on at least one path.

Test cases are sometimes derived from requirements, where for each requirement, at least one test case has to ensure that the requirement is satisfied (or, covered). When requirements are used, testers usually refer to behavioral or functional requirements that describe how the software should behave. But they can also refer to *non-functional requirements* such as performance, timeliness, liveness, stability, smoothness, and responsiveness, among others.

2.3. Model-Driven Engineering

Model-driven engineering (*MDE*) [57] is an approach to software development that starts with an abstract design model that ignores concerns regarding the implementation language, operating system, and target hardware. An *executable model* is written in a language with enough semantics so they can be simulated directly. Models without such semantics are called *non-executable models*. Executable models are sometimes called *formal models*. Models are transformed to code either automatically by special-purpose compilers or by hand. When transformed by hand, the process is often called *model-based design* (*MBD*).

The studies we summarize in this article do not always apply the terminology consistently, so we introduce several terms here so we can emphasize their overlap and differences. A *platform-independent model* (*PIM*) [40] describes the behavior of a system in an abstract modeling language; this is also called the *model level*. The complementary *platform-specific model* (*PSM*) [40] is the *code level*, that is, the system implemented in a programming language such as C or Java. Some studies also adopt the terms *computation independent models* (*CIM*) [40] for models that do not depend on a computation model. We also find the terms *model-in-the-loop* with the aim of describing software development processes that include abstract models and *processor-in-the-loop* to describe implementations at the code level.

2.4. Model-Based Testing

Models are defined at an abstract, high level, making them convenient artifacts for designing test cases [41]. *Model-based testing* (*MBT*) designs test cases from an abstract model (*model-level* or *abstract* tests), and transforms them into test cases that can be run on the code (*code-level* or *concrete* tests). The term *computation-independent tests* (*CIT*) is sometimes used for model-level tests and *computation-dependent tests* for code-level tests. When test cases are designed from informal models or code-based models, such as control-flow graphs, we sometimes use the term *model-driven testing* (*MDT*).

2.5. Issues of Transforming Models to Code

When models are transformed to code, whether automatically or by hand, the structure of the code might differ from the structure in the design model [7, 39]. This brings up a serious issue: the code-level test cases may not achieve the same level of coverage as the model-level test cases. This is serious for aeronautics software in particular, and transportation software in general. For example, the US Federal Aviation Administration (FAA) requires full code coverage to certify safety critical avionics software [46], and requires that each test must be derived from the requirements. This makes it imperative that when model-based test cases are transformed to the concrete level, testers are able to ensure *traceability* from abstract tests to concrete tests [2]. When code is automatically derived from models, potential problems with the transformation m motivate the application of the so-called *witness functions* (in the form of traceability [3]) that allow differences to be discovered.

3. STUDY SETUP

This section provides key information about the protocol we defined for our SLR. We follow the guidelines for conducting secondary studies proposed by Kitchenham et al. [26] and Wohlin [58]. Our full protocol is available online.[5]

3.1. Goals and Research Questions

The general goal of this SLR is *to analyze the state of the art in model-based testing with respect to how source code coverage can be measured from test sets generated using model-based testing approaches.* This goal is achieved by answering the following research questions (RQs):

- RQ1: *How are test suites that are developed at the model level mapped to the code level; code which may or may not be created by automatic transformation?*
- RQ1.1: *What is required of the model-to-code transformation to support the transition from model level tests to code level tests?*
- RQ2: *How are tests generated from the model specifications (e.g. UML or Simulink)?*
- RQ3: *How does the coverage of the model produced by abstract tests relate to the coverage of the code for the corresponding concrete tests?*
- RQ4: *Which are the applied technologies and which are the software development tasks focused by studies that address mapping of tests across model and code levels?*

Our RQs emphasize transformation details because we believe that by having a more complete understanding of "under the hood" transformation details testers can have a better idea of how to improve test cases at both model and code levels. As a result, testing and language design principles can be brought to bear on the model-to-code transformation problem. Specifically, by being more knowledgeable about details of the modeling language, testers can help the language evolve by making certain constructs/elements more explicit (*i.e.* targeted by the transformation).

It is also worth mentioning that, as stated by Stürmer et al. [50], rendering high-level models into code poses a set of challenges that in a way differ from the challenges inherent to traditional compiler design. Most notably, the semantics of the modeling language often is not explicit, and may depend on layout information (*e.g.* position of the states). Consequently, code generation entails more than simply performing stepwise transformations from the model representation into code: in effect, a series of computation must be derived from the analysis of data dependencies. Therefore, understanding the model-to-code transformer backend and how it turns models into code can help testers arrive at a better operational understanding of the transformation and allow them to focus on corner cases (boundary values and code elements that are seldom covered during MBT).

[5] `https://doi.org/10.5281/zenodo.8113394`

3.2. Inclusion and Exclusion Criteria

The inclusion criteria (Section 3.2.1) and exclusion criteria (Section 3.2.2) for study selection are presented in this section. A study was selected if it passed $((I1 \wedge I2) \vee I3) \wedge I4)$. Studies that fulfilled at least one of the exclusion criteria were not selected.

The protocol available online provides more details about how these criteria were applied. Regarding E2, although secondary studies were not included in our final set of selected studies, some may be of interest as a source for new studies; therefore, they were analyzed in an additional study selection round (details in Section 4).

3.2.1. Inclusion (I) Criteria

I1: *The study proposes/applies model-based testing for/to models.*

I2: *The study addresses automatic model-to-code (or model-to-text) transformation.*

I3: *The study addresses the mapping from test suites developed at model level to source code level.*

I4: *The study must have undergone peer-review.*

We highlight that this literature review particularly focuses on research that addresses model-to-model or model-to-code transformations, with an emphasis on the automatic transformation of models to code. Note that our RQs and inclusion criteria (particularly, I2 and I3) reflect this intent. This is not strictly the case of MBT in general, which would be the case of I1 if applied individually. While the combination of I1 with I2 allows for the selection of studies that explore MBT and forward engineering of the models to lower levels of abstraction, I3 solely leads to the selection of studies that establish relationships between tests that evolve from the model level to the code level. The three rightmost columns of Tables II and III show the criteria each selected study fulfilled.

Regarding I4, we only selected studies published in scholarly venues (which are well-established types within the research community), namely, conference proceedings, symposium proceedings, workshop proceedings, and scientific journals.

3.2.2. Exclusion (E) Criteria

E1: The study emphasizes hardware testing.

E2: The study is a secondary study.

E3: The study is a peer-reviewed study that has not been published in journals, conferences, symposia, or workshop proceedings (*e.g.* Ph.D. theses and technical reports).

E4: The study is not written in English.

3.3. Search Strategy

The first focus of our work is on a literature study. As we found it very hard to find search strings to match a manageable number of primary studies of interest to this study, we employed a *snowballing* process based on three initial studies. Snowballing, also referred to as *citation analysis*, is a literature search method that can take one of two forms: backward snowballing or forward snowballing [26, 58]. Backward snowballing starts the search from a set of studies that are known to be relevant (either a start set, or the current set of selected studies). It involves searching the references sections of the studies. Forward snowballing entails finding all studies that cite a study from either the start set or the current set of selected studies. Both search methods update the set of selected studies in an iterative fashion; only the studies included in the previous step are considered in each search iteration, and both backward and forward snowballing end when no new primary studies are found in the search iterations.

Three reviewers were in charge of running the search process. During backward snowballing, they extracted references from the background, related work, and experimental setup sections of the study

Prepared using stvrauth.cls
DOI: 10.1002/stvr

under analysis. For studies that did not include these sections, they considered other sections such as the introduction. Details of our analysis procedures can also be found in the spreadsheet that is available online.[6]

Our initial set of primary studies, the seeds, includes the three studies listed in the sequence. In order to achieve a comprehensive understanding of the research landscape in this field, we selected studies based on three criteria: age, prominence, and relevance. Our goal was to identify a *visionary* paper that identified the problem and a *mature* paper that represented the current state-of-the-art. Along with these two papers, we included a paper from the research group that inspired this work. While we acknowledge the limitations of age and prominence as selection criteria, we believe that taking these criteria into consideration was necessary to identify key contributions to the field. Older studies tend to have more citations since these studies have had more time to accumulate citations, while recent studies may not have had the opportunity to accumulate as many citations. However, prominent studies may have gained attention more quickly, hence these studies may have been cited more frequently in a shorter amount of time. In hindsight, our selection criteria led to the identification of seeds that have proven to be valuable for the snowballing process, leading to the identification of additional key contributions in the field. Thus, we believe that our approach was a useful starting point for our study.

1. *Testing the Untestable: Model Testing of Complex Software-intensive Systems*, by Briand et al. [9];
2. *Data Flow Model Coverage Analysis: Principles and Practice*, by Camus et al. [10]; and
3. *UML Associations: Reducing the Gap in Test Coverage Between Model and Code*, by Eriksson and Lindström [18].

Search Stopping Criterion: To keep the review feasible, for the study selection phase we executed three snowballing iterations, called *rounds*, after which we started the data extraction and synthesis. We analyze this stopping criterion in Section 6 (Threats to Validity).

3.4. Procedures for Data Extraction and Analysis

To answer the RQs described in Section 3.1, we extracted from primary studies the information outlined in a data extraction form. Before starting the review, the data extraction form was revised by all involved reviewers. Beyond data extraction fields intended to gather general information about the primary studies (*e.g.* title, authors, year, and publication venue), the form includes the following fields:

(1) The general goal of the study;

(2) A description of the study from the perspective of each research question;

(3) The main results of the study;

(4) The conclusion of the study, *cf.* the original authors;

(5) The conclusion of the study, *cf.* the reviewers;

(6) The target specification language (at model level);

(7) The target programming language (at code level);

(8) The tool used for model-to-text transformation (for the main software artifacts);

(9) The tool used for automatic test case generation (at the model level);

[6] https://doi.org/10.5281/zenodo.8113394

Softw. Test. Verif. Reliab. (2023)
DOI: 10.1002/stvr

(10) The tool used for test set transformation (from model to source code);

(11) The obtained code coverage obtained with model-based test set;

(12) Level of automation for model-based test generation;

(13) Level of automation for test re-execution (model $\rightarrow$ code);

(14) Level of traceability of model elements $\rightarrow$ code elements; and

(15) Level of automation for traceability of model elements $\rightarrow$ code elements.

After extracting data from all selected studies, the three reviewers in charge of the primary study selection checked all extracted data to make sure the data is accurate and ready for further analysis.

We intentionally structured the data extraction form in terms of the RQs to facilitate the identification of pieces of information that would help us develop the discussion as well as outline the conclusions with respect to each RQ. In particular: fields (1) to (5) supported the discussions regarding RQ1, RQ1.1, RQ2, and RQ3; fields (6) to (10) supported the discussions regarding RQ4; field (11) supported the discussions regarding RQ3; and fields (12) to (15) added details to enrich the descriptions and discussions presented in this article. All studies that provided relevant information with respect to a given RQ are listed in the beginning of the sections that discuss the RQs (namely, Sections 5.1 to 5.5).

4. SEARCH ITERATIONS AND RESULTS

Table I summarizes the search rounds. It shows the number of backward references and forward citations analyzed in each study selection round, and shows which studies we have selected. For the sake of completeness, the table includes the initial seeds in Round 0. The analysis of forward citations was updated in February, 2020. Columns 4, 5, and 8 show two values for each entry regarding forward snowballing: the left-hand values refer to the first analysis of forward citations, and the right-hand values refer to the most recent analysis. As an example, for study P0003 (column 2), we analyzed 12 studies retrieved in March 3, 2018, and an additional 12 studies retrieved in February 18, 2020. From these, none were included in our dataset (column 8). The table provides the following details:

- The study IDs[7] and references (column 2). The study IDs are composed by a prefix P followed by a sequential number assigned to each study we retrieved through either backward or forward snowballing.
- The number of analyzed backward references and forward citations with respect to each seed (columns 3 and 4, respectively). The numbers of backward references and forward citations listed in the table refer only to non-duplicate entries (*i.e.* entries that did not appear in a previously analyzed study).
- The date on which forward citations were retrieved with the Google Scholar[8] search engine (column 5).
- The number of selected studies through backward snowballing and which studies these are (columns 6 and 7).
- The number of selected studies through forward snowballing and which studies these are (columns 8 and 9). In column 9, studies with a * prefix were selected in the forward snowballing update.

[7]Key primary studies in this literature review are cited in the references, and also indexed by "P"-numbers for brevity. The P-numbers are used in figures and in the online material, which has complete bibtex-formatted references and much more information about each primary study and how we categorized it.

[8]http://scholar.google.com/ – accessed in June, 2023.

DOI: 10.1002/stvr

Note that two additional rounds (*Additional Round* in Table I) were performed. The first concerns the analysis of a secondary study (ID P0237) found in Round 2, from which we retrieved and analyzed the references. The round named *Additional Round (from experts)* refers to the analysis of studies suggested by experts,[9] which was done in February, 2022.

In total, we analyzed 180 backward references and 318 forward citations. From the backward references, 17 studies were selected, whereas 16 studies were selected from forward citations. From the 33 selected studies, P0064 [52] was subsumed[10] by P0253 [51]; furthermore, P0498 [15] and P0499 [35] were subsumed by P00487 [14]. Therefore, we ended up with a set of 30 studies that we analyze in the next sections of this article.

To illustrate the process of analyzing a particular study, from the start set, let us consider study P0003, by Briand et al. [9], titled *Testing the Untestable: Model Testing of Complex Software-intensive Systems*. We have observations from both backward and forward snowballing.

- Backward snowballing: We analyzed the “Background & State of the Art” section of the study, since the study is not a conventional paper (it was published in the *Visions of 2025 and Beyond* Track of ICSE 2016). There are eight backward references. From these, two were selected: P0010 [49], and P0011 [36].

- Forward snowballing: We analyzed 12 forward citations to this study in March 2018, and another 12 in February 2020. None were selected.

Tables II and III list the 30 studies we analyze in this article. They show the study ID (column *ID*), the snowballing iteration round (column *R*) that reflects the first detection of a study, the snowballing technique (column *B/F* for ’B’ackward or ’F’orward), the reference entry (column *Ref.*), the list of authors (column *Author(s)*), the study title (column *Title*), the venue in which the study was published or presented (column *Venue*), and the results of the application of the inclusion criteria (columns *I1*, columns *I2* and columns *I3*). In the column that indicates the round, “4” represents the forward snowballing update, “a1” represents *Additional Round (from SLR)*, and “e1” represents *Additional Round (from experts)*.

Figure 1 depicts the distribution of selected studies per publisher. IEEE Xplore[11] includes the most studies in our SRL (twelve studies), followed by ACM Digital Library[12] (five studies) and Springer SpringerLink[13] (four studies).

Figure 2 shows the citation map between the selected studies. Continuous edges indicate studies retrieved via backward snowballing; in these cases, a study in a destination node was cited by the study in the origin node (*e.g.* P0003 cited P0010 and P0011). Dashed edges indicate studies retrieved via forward snowballing; in these cases, a study in an origin node cited the study in the destination node (*e.g.* P0010 is cited by P0071, P0086, P0443, P0448, and P0451). Studies with no incoming and outgoing edges were included based on experts’ suggestions (namely, P0496, P0497, P0498, and P0499). In the citation map, the set of 30 studies we analyze in this article is composed of the 3 studies shown in white background (original seeds) and the 27 studies shown in light gray background (selected studies).

The top of Figure 2 has a timeline for study publication. Starting from the left-hand side, the graph shows that the most recent selected studies were published in 2019. Figure 2 also provides a transitive trace between studies selected in our SLR. The start set (initial seeds) is composed of P0003 [9], P0004 [10], and P0005 [18]. By taking P0005 as an example, we see that it was influenced, among others, by P0045; then also, P0045 influenced P0234, which in turn influenced P0374, P0375 and P0463.

[9]The experts were reviewers of prior versions of this article. In the reviews, they suggested a set of studies that we analyzed according to our study selection criteria. The studies that passed our inclusion criteria were added to our final set.
[10]A study subsumes another study when it updates a technique previously published, or extends a prior publication.
[11]http://ieeexplore.ieee.org/Xplore/home.jsp – accessed in June, 2023.
[12]http://dl.acm.org/ – accessed in June, 2023.
[13]http://link.springer.com/ – accessed in June, 2023.

DOI: 10.1002/stvr

Table I. Summary of search rounds (in Round 0, studies P0003, P0004 and P0005 are listed in *Selected Backward* column just for convenience; these were the original seeds upon which we started the snowballing process).

1	2 Seed	3 # Analyzed Backward	4 # Analyzed Forward	5 Date Forward	6 # Selected Backward	7 Selected Backward	8 # Selected Forward	9 Selected Forward
Round 0	n/a n/a	n/a	n/a	n/a	1	P0003 Briand et al. [9]	n/a	
	n/a n/a	n/a	n/a	n/a	1	P0004 Camus et al. [10]	n/a	
	n/a n/a	n/a	n/a	n/a	1	P0005 Eriksson and Lindström [18]	n/a	
Subtotal		n/a	n/a		3		n/a	
Round 1	P0003 Briand et al. [9]	8	12 + 12	3/15/2018 – 2/18/2020	2	P0010 Shokry and Hinchey [49] P0011 Matinnejad et al. [36]	0 + 0	
	P0004 Camus et al. [10]	6	0 + 1	3/22/2018 – 2/18/2020	0		0 + 1	*P04111 Aniculaesei et al. [5]
	P0005 Eriksson and Lindström [18]	30	0 + 1	4/6/2018 – 2/18/2020	3	P0056 Kirner [25] P0045 Eriksson et al. [20] P0059 Eriksson et al. [19]	0 + 0	
Subtotal		44	26		5		1	
Round 2	P0010 Shokry and Hinchey [49]	4	58 + 21	5/1/2018 – 2/18/2020	1	P0064 Stürmer et al. [52]	2 + 3	P0071 Tekcan et al. [53] P0086 Li et al. [29] *P0443 Durak et al. [16] *P0448 Koch et al. [27] *P0451 Amalfitano et al. [3]
	P0011 Matinnejad et al. [36]	35	23 + 20	6/17/2018 – 2/18/2020	1	P0158 Mohalik et al. [38]	0 + 0	
	P0056 Kirner [25]	22	21 + 1	6/17/2018 – 2/18/2020	1	P0223 Baresel et al. [7]	0 + 0	
	P0045 Eriksson et al. [20]	7	4 + 0	6/5/2018 – 2/18/2020	0		1 + 0	P0234 Li and Offutt [31] ***P0237 Abade et al. [1]
	P0059 Eriksson et al. [19]	1	2 + 0	6/14/2018 – 2/18/2020	0		0 + 0	
Subtotal		69	150		3		6	
Round 3	P0064 Stürmer et al. [52]	11	43 + 6	7/19/2018 – 2/18/2020	0		2 + 0	P0253 Stürmer et al. [51] P0259 Conrad [12]
	P0071 Tekcan et al. [53]	7	11 + 0	8/8/2018 – 2/18/2020	0		0 + 0	
	P0086 Li et al. [29]	15	2 + 0	9/18/2018 – 2/18/2020	0		0 + 0	
	P0234 Li and Offutt [31]	5	6 + 3	11/1/2018 – 2/18/2020	0		2 + 1	P0374 Li and Offutt [32] P0375 Li et al. [30] *P0463 Vanhecke et al. [55]
	P0223 Baresel et al. [7]	4	31 + 8	8/19/2018 – 2/18/2020	0		3 + 0	P0313 Pretschner et al. [45] P0321 Conrad et al. [13] P0362 Amalfitano et al. [2]
	P0158 Mohalik et al. [38]	7	14 + 18	9/20/2018 – 2/18/2020	0		0 + 1	*P0474 Kalaee and Rafe [23]
Subtotal		49	142		0		9	
Additional Round (from SLR)	P0237 Abade et al. [1]	8	n/a	n/a	2	P0381 Lamancha et al. [28] P0383 Fraternali and Tisi [21]	n/a	n/a
Subtotal		8			2			
Additional Round (from experts) (February, 2022)		10	n/a	n/a	4	P0496 Veanes et al. [56] P0497 Drave et al. [14] P0498 Drave et al. [15] P0499 Markthaler et al. [35]	n/a	n/a
Subtotal		10			4			
Total		180	318		17		16	

* selected in the forward snowballing update.
*** not selected, but used as source of references in the additional round.
In Round 0, studies P0003, P0004 and P0005 are listed in *Selected Backward* column just for convenience; these were the original seeds upon which we started the snowballing process.
In *Additional Round (from experts)*, studies are listed in columns related to backward snowballing for convenience.

Table II. Selected studies (part 1/2) (R = round; B/F = (B)ackward snowballing, (F)orward snowballing; I1/I2/I3 = inclusion criterion I_i) (I4 is omitted because all studies are peer-reviewed).

ID	R	B/F	Ref.	Author(s)	Year	Title	Venue	I1	I2	I3
P0003	0		[9]	Briand et al.	2016	Testing the Untestable: Model Testing of Complex Software-intensive Systems	International Conference on Software Engineering (ICSE) - Visions of 2025 and Beyond Track	✓	✓	
P0004	0		[10]	Camus et al.	2016	Data Flow Model Coverage Analysis: Principles and Practice	European Congress on Embedded Real Time Software and Systems (ERTS)	✓		✓
P0005	0		[18]	Eriksson and Lindström	2016	UML Associations: Reducing the Gap in Test Coverage Between Model and Code	International Conference on Model-Driven Engineering and Software Development (MODELSWARD)	✓	✓	✓
P0010	1	B	[49]	Shokry and Hinchey	2009	Model-Based Verification of Embedded Software	IEEE Computer	✓	✓	✓
P0011	1	B	[36]	Matinnejad et al.	2015	Search-based Automated Testing of Continuous Controllers: Framework, Tool Support, and Case Studies	Information and Software Technology	✓		✓
P0045	1	B	[20]	Eriksson et al.	2013	Transformation Rules for Platform Independent Testing: An Empirical Study	International Conference on Software Testing, Verification and Validation (ICST)	✓	✓	✓
P0056	1	B	[25]	Kirner	2009	Towards Preserving Model Coverage and Structural Code Coverage	EURASIP Journal on Embedded Systems	✓	✓	✓
P0059	1	B	[19]	Eriksson et al.	2012	Model Transformation Impact on Test Artifacts: An Empirical Study	Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)	✓	✓	
P0071	2	F	[53]	Tekcan et al.	2012	User-driven Automatic Test-case Generation for DTV/STB Reliable Functional Verification	IEEE Transactions on Consumer Electronics	✓	✓	
P0086	2	F	[29]	Li et al.	2011	A Case Study on SDF-based Code Generation for ECU Software Development	International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS)	✓	✓	
P0234	2	F	[31]	Li and Offutt	2015	A Test Automation Language Framework for Behavioral Models	Workshop on Advances in Model Based Testing (A-MOST)	✓	✓	✓
P0223	2	B	[7]	Baresel et al.	2003	The Interplay between Model Coverage and Code Coverage	EuroSTAR Software Testing Conference	✓	✓	
P0158	2	B	[38]	Mohalik et al.	2014	Automatic Test Case Generation from Simulink/Stateflow Models using Model Checking	Software Testing, Verification and Reliability	✓	✓	✓
P0253	3	F	[51]	Stürmer et al.	2007	Systematic Testing of Model-Based Code Generators	IEEE Transactions on Software Engineering	✓	✓	✓
P0259	3	F	[12]	Conrad	2009	Testing-based Translation Validation of Generated Code in the Context of IEC 61508	Formal Methods in System Design	✓	✓	
P0313	3	F	[45]	Pretschner et al.	2005	One Evaluation of Model-based Testing and Its Automation	International Conference on Software Engineering (ICSE)	✓	✓	✓
P0321	3	F	[13]	Conrad et al.	2005	Automatic Evaluation of ECU Software Tests	SAE Transactions	✓	✓	
P0362	3	F	[2]	Amalfitano et al.	2015	Comparing Model Coverage and Code Coverage in Model Driven Testing: An Exploratory Study	International Workshop on Testing Techniques for Event BasED Software (TESTBEDS)	✓	✓	✓
P0374	3	F	[32]	Li and Offutt	2016	Test Oracle Strategies for Model-Based Testing	IEEE Transactions on Software Engineering	✓	✓	✓
P0375	3	F	[30]	Li et al.	2016	Skyfire: Model-Based Testing with Cucumber	International Conference on Software Testing, Verification and Validation (ICST) - Testing Tool Papers	✓	✓	✓
P0381	a1	B	[28]	Lamancha et al.	2011	Model-driven Testing - Transformations from Test Models to Test Code	International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)	✓	✓	✓
P0383	a1	B	[21]	Fraternali and Tisi	2010	Multi-level Tests for Model Driven Web Applications	International Conference on Web Engineering (ICWE)	✓	✓	✓
P0411	4	F	[5]	Aniculaesei et al.	2019	Using the SCADE Toolchain to Generate Requirements-Based Test Cases for an Adaptive Cruise Control System	Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)	✓	✓	✓

Table III. Selected studies (part 2/2) (R = round; B/F = (B)ackward snowballing, (F)orward snowballing; I1/I2/I3 = inclusion criterion I_i) (I4 is omitted because all studies are peer-reviewed).

ID	R	B/F	Ref.	Author(s)	Year	Title	Venue	I1	I2	I3
P0443	4	F	[16]	Durak et al.	2018	Modeling and Simulation based Development of an Enhanced Ground Proximity Warning System for Multicore Targets	International Symposium on Model-driven Approaches for Simulation Engineering (Mod4Sim)	✓	✓	✓
P0448	4	F	[27]	Koch et al.	2018	Simulation-based Verification for Parallelization of Model-based Applications	Computer Simulation Conference (SummerSim)	✓	✓	✓
P0451	4	F	[3]	Amalfitano et al.	2019	Using Tool Integration for Improving Traceability Management Testing Processes: An Automotive Industrial Experience	Software: Evolution and Process	✓	✓	✓
P0463	4	F	[55]	Vanhecke et al.	2019	AbsCon: A Test Concretizer for Model-Based Testing	Workshop on Advances in Model Based Testing (A-MOST)		✓	✓
P0474	4	F	[23]	Kalaee and Rafe	2019	Model-based Test Suite Generation for Graph Transformation System Using Model Simulation and Search-based Techniques	Information and Software Technology	✓	✓	
P0496	e1		[56]	Veanes et al.	2008	Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer	Formal Methods and Testing Workshop (FORTEST)	✓		✓
P0497	e1		[14]	Drave et al.	2019	SMArDT modeling for automotive software testing	Software: Practice and Experience	✓		✓

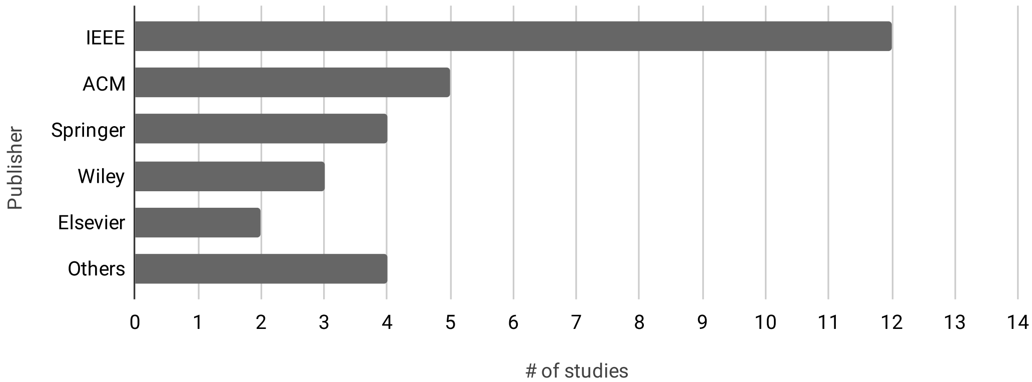

Figure 1. Number of studies per publisher 30 studies, in total).

5. ANALYSIS BASED ON THE RESEARCH QUESTIONS

This section provides answers to the RQs that were defined in Section 3. Table IV classifies the studies based on the research questions RQ1 to RQ3 (separate tables are shown in Section 5.5 to support the discussion regarding RQ4). We discuss each RQ in turn.

In the first paragraphs of Sections 5.1 to 5.4 we present the characteristics that we considered to group the studies that helped us draw answers to the RQs. Beyond this, we discuss the studies in ascending chronological order, with a few exceptional cases which involve studies that are closely related (*e.g.* pieces of research that were evolved by the same research group) or studies that to a limited extent contributed to the RQ answers.

5.1. Discussion Regarding RQ1: How are test suites that are developed at the model level mapped to the code level; code which may or may not be created by automatic transformation?

For discussing RQ1, we grouped the 23 studies listed in the first line of Table IV as follows: studies that directly provided information regarding the transformation of test cases across the abstraction

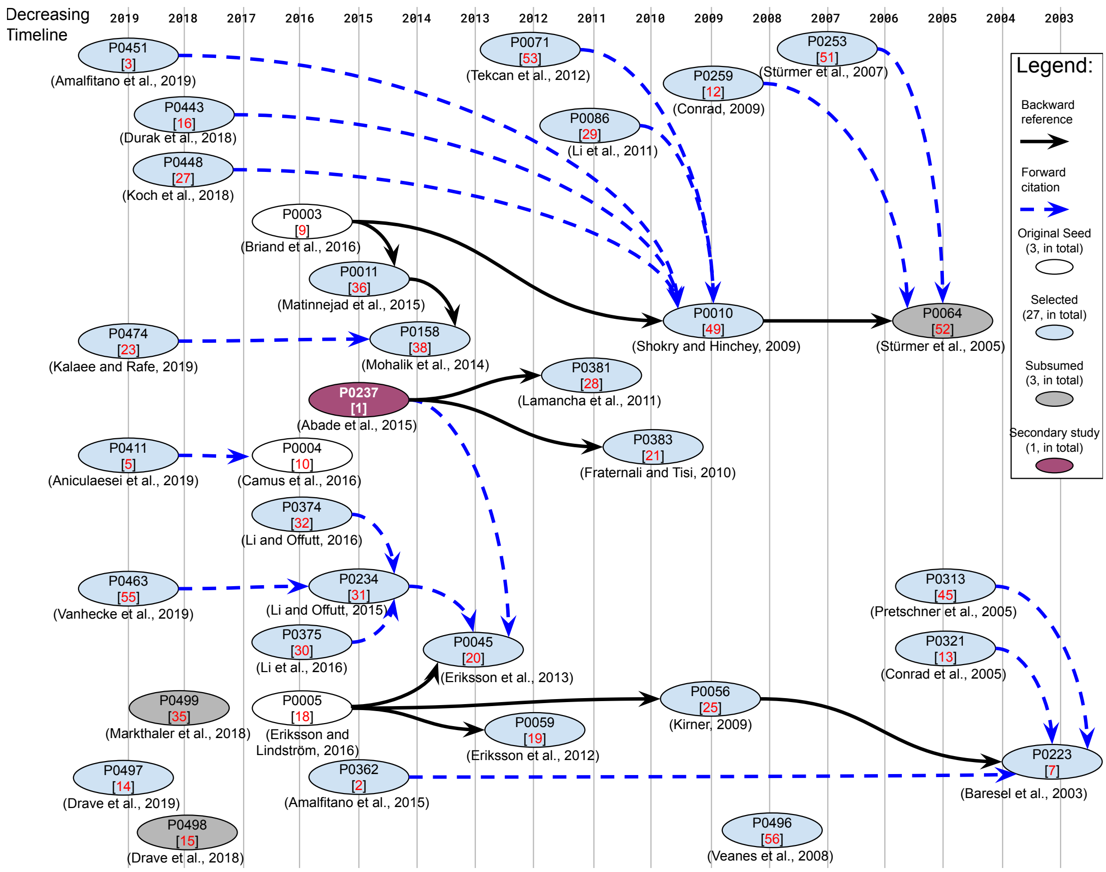

Figure 2. Citation map for studies that passed the inclusion criteria with decreasing timeline (from left to right).

Table IV. Classification of studies with respect to our research questions.

RQ	# of studies	References
RQ1	23	[2, 3, 5, 10, 14, 16, 18, 20, 21, 23, 25, 27, 28, 30, 31, 32, 36, 38, 45, 49, 51, 55, 56]
RQ1.1	17	[2, 3, 5, 10, 16, 18, 21, 25, 27, 28, 30, 31, 32, 45, 51, 55, 56]
RQ2	27	[2, 5, 7, 10, 12, 13, 14, 16, 18, 20, 21, 23, 25, 27, 28, 29, 30, 31, 32, 36, 38, 45, 49, 51, 53, 55, 56]
RQ3	13	[2, 5, 7, 14, 18, 20, 23, 32, 38, 45, 49, 53, 56]

levels by describing the tool that supports the transformation [2, 3, 5, 10, 16, 27, 38, 51, 56]; studies that described procedures for transforming test cases from models to code [14, 21, 25, 28, 30, 31, 32, 45, 55]; and, studies that just reported that test cases developed at the model level are then applied to test the code [18, 20, 23, 36, 49]. Studies from the three groups are discussed in the sequence.

With respect to *studies that described tools*, Stürmer et al. [51] developed tools to automatically transform test cases based on executable models. The study reported on test vectors generated for Simulink and Stateflow[14] models that can be automatically executed on auto-generated C code with support of a tool called *Mtest*. Their approach allows the model elements to be traced to code, including changes performed by a model-to-code transformation optimizer. However, the authors did not give technical details on how Simulink and Stateflow models are turned into code, or how test vectors are transformed into code.

[14] http://www.mathworks.com/products/stateflow.html – accessed in June, 2023.

DOI: 10.1002/stvr

Veanes et al. [56] presented details of the Spec Explorer[15] tool that uses a state model (specified in a language named Spec#) to derive abstract test cases. Spec Explorer employs algorithms similar to those of explicit state model checkers to explore the machine's states and transitions; the automatically generated abstract tests are further converted into concrete tests. The authors defined a set of rules to map what they call *action methods* (at the model level) to concrete methods present in the actual system under test.

Mohalik et al. [38], also in the context of Simulink and Stateflow models, developed the AutoMOTGen test generation tool. AutoMOTGen transforms Stateflow models into code written in the SAL language, which underlies the generation of test cases. Test coverage requirements are encoded as goals in SAL to establish traceability, and a model checking engine is utilized to generate test cases from counter-example traces. The tool generates test cases to satisfy block coverage, condition coverage, decision coverage, and MCDC. The generated test cases are directly used to test the code produced from the models.

Camus et al. [10] employed the SCADE tool suite[16] to automatically transform model-based test cases to be directly applied to source code. The Model Test Coverage[17] (MTC) tool was employed to run tests and collect model coverage data. They also applied structural code coverage analysis on the code. When applying the resulting test cases to code, the code coverage can be used as a measure of conformance to standards such as DO-178C/DO-331.

Amalfitano et al. [2] studied test cases that were automatically generated to provide "full coverage" (such as the coverage of all states, all transitions, and all paths) of UML state machines and then run on automatically generated Java code. They employed the Conformiq Designer tool[18] to generate test cases at the model level, and then automatically transform model-level test cases into code. In another research initiative, Amalfitano et al. [3]

reported on a relatively simple experiment to probe into the difference between model and code coverage for four different state machine models and eight test sets. They specified test cases in an ordinary spreadsheet that is automatically processed by a legacy, homemade, unnamed testing environment; the same test cases are executed at both model and code levels.

Koch et al. [27] presented the Scilab/Xcos XTG[19] tool. It supports the Durak et al.'s [16] X-in-the-loop testing pipeline for model-based development of parallel real-time software that runs on multicore processor architectures tailored to the avionics industry. Scilab/Xcos XTG enables back-to-back testing by injecting automatically generated code into the model elements, thus allowing enhanced simulations to be carried out at the model level. It also generates input test data and expected output that can be used to exercise the model and the code at various phases of the model-based testing workflow. In both studies, a single example was outlined, without any further empirical assessment.

Aniculaesei et al. [5] compared the fault revealing capability of test sets automatically generated with a commercial tool (the SCADE tool suite[16] and an academic, open-source tool (NuSMV) that applies the model checking approach. Both tools turn models and test cases generated at the model level into C code, and the study assessed the effectiveness of the test sets based on their mutation scores.

Regarding *studies that described procedures for transforming test sets from models to code*, Pretschner et al. [45] provided clear information about model-to-code transformation of test sets. The study describes a compiler that transforms abstract, model-level test cases to concrete, code-level test cases. Model-level test cases are automatically generated based on program specifications written in a

[15] https://marketplace.visualstudio.com/items?itemName=SpecExplorerTeam.SpecExplorer2010VisualStudioPowerTool-5089 – accessed in June, 2023.
[16] http://www.ansys.com/products/embedded-software/ansys-scade-suite – accessed in June, 2023.
[17] https://www.ansys.com/training-center/course-catalog/embedded-software/introduction-to-ansys-scade-test-model-coverage-for-scade-suite – accessed in June, 2023.
[18] https://tinyurl.com/mr3bx8sv – accessed in June, 2023.
[19] https://www.scilab.org/software/xcos – accessed in June, 2023.

constraint logic programming language. Some of their test cases were generated automatically, some from models and some from code, some randomly, some with functional testing criteria, and some by hand. They found that test cases generated from models found more faults, especially faults that resulted in changes to requirements.

Kirner [25] theoretically addressed the problem of preserving structural code coverage after transformations that are applied by automatic code generators and compilers. The key idea is that program properties must be maintained when program P_1 is transformed into program P_2, so that the structural coverage on P_1 is preserved in P_2 with the same test data. The author defined formal rules based on coverage criteria (statement coverage, decision coverage, and MCDC), a set of coverage preservation rules, and a set of code optimizations. Kirner's study presents examples on Simulink models.

Fraternali and Tisi [21] developed an MDE approach that addresses a series of model-to-model and model-to-text transformations to automatically generate test cases at the model level, then transform them to the code level. At the highest abstraction level, the Computation Independent Model (BPMN[20]), models are handled in two transformation streams, system and test model. At the lowest abstraction level, the Platform Specific Model, their tool produces Java code and web test scripts for a tool called WebTest.[21] The test scripts are updated by mappings that can be applied after changes take place in the system models.

Similarly to Fraternali and Tisi's MDE approach [21], Lamancha et al. [28] extended a previously implemented framework to automatically derive code-level test cases from model-level test cases. The framework first does a model-to-model transformation from UML to UML Testing Profile models, then uses the MofSCript[22] tool to transform the abstract tests to JUnit[23] or NUnit[24] test cases.

Li and Offutt [31] introduced the STALE[25] framework to automatically transform test cases from the model level to the code level. Unlike approaches such as the one by Camus et al. [10], STALE handles non-executable models (statecharts) that are typically transformed into source code by hand. Li and Offutt created a language named STAL (Structured Test Automation Language), based on which testers created model-code-transformations for piecewise test components. These components were then assembled automatically to create JUnit[23] test scripts. In a subsequent study, Li and Offutt [32] employed the STALE framework to investigate test oracle strategies, empirically evaluating how much of the program state should be evaluated in automated tests, and when the evaluation should be done. Li et al. [30] later presented the *skyfire*[26] MBT tool to support automatic generation of Cucumber[27] test scenarios. This approach included manual effort to define the Cucumber steps that are further automatically handled by *skyfire* to produce Cucumber test scenarios based on the abstract tests produced by STALE.

Vanhecke et al. [55] described an approach that is embedded in the AbsCon (Abstract test case Concretizer) tool. The approach consists in generating executable test cases from abstract definitions. Abstract tests are initially defined in an XML file in which each test case is described as a sequence of actions and assertions regarding the system under test. Concrete tests are generated as Python scripts that execute the verification steps and sequences of assertions.

Drave et al. [14] presented an approach to manage requirements, design, and test. The approach emphasizes the technical aspects of the models that appear in the different layers of the V-Model. According to the authors, by ensuring consistency among the models in these different layers it is possible to turn high-level test representations into lower level representations automatically.

[20] `http://www.bpmn.org/` – accessed in June, 2023.
[21] `https://daveparillo.github.io/webtest/manual/WebTestHome.html` – accessed in June, 2023.
[22] `https://marketplace.eclipse.org/content/mofscript-model-transformation-tool` – accessed in February, 2022.
[23] `http://junit.org` – accessed in June, 2023.
[24] `http://nunit.org/` – accessed in June, 2023.
[25] `http://cs.gmu.edu/~nli1/stale/` – accessed in June, 2023.
[26] `http://github.com/mdsol/skyfire` – accessed in June, 2023.
[27] `http://cucumber.io/` – accessed in June, 2023.

As a proof-of-concept, the authors described how their approach can be implemented in a modeling environment agnostic fashion through a configurable tool chain that can render functional requirements modeled using activity diagrams, state charts, and sequence diagrams into executable test cases for various outputs. Therefore, although Drave et al. emphasized the description of the proposed approach, they also provided some insights into how their approach can be realized.

Regarding *studies in which the authors stated that test cases developed at the model level are also applied to test the code, however without providing details of the test transformation,* Shokry and Hinchey [49] concluded that test cases randomly generated at the model level provide low code coverage, but did not provide details. Matinnejad et al. [36] applied nine test cases generated at the model level in practice at the HIL stage, but did not provide details either. Eriksson and Lindström [18] proposed new model-based coverage criteria that are computed from executable xtUML[28] models. They continued the work of Eriksson et al. [19] by generating logic-based test cases at the platform-independent level using xtUML. Eriksson and Lindström's approach comprises measuring coverage at the model level by first creating model-level predicates that capture the predicates that would appear during model transformation to code. This allows test coverage to be measured at the model level. In that study, for a single subject application from the avionics domain, the authors reported on the coverage achieved by a test set generated at the model level and re-executed at the code level. However, neither that study nor a prior study on the same project [20] provided details of how the test set is mapped across the abstraction levels. Finally, Kalaee and Rafe [23] mentioned that test cases generated at the model level, based on graphs and transformation rules, can be transformed into sequences of method invocations, but the authors did not elaborate on it.

To summarize the RQ1-relevant studies, we identified the following two perspectives regarding the transformation, or reuse, of test cases generated at the model level to test, or evaluate, the code derived from models: fully-automated transformation and execution of test cases, and partially-automated or manual transformation of test cases with subsequent automated execution. Both perspectives are following summarized.

- Test cases are fully automatically transformed from model into code by using specific industrial or tailor-made tools. Software specifications are automatically transformed into code, and test cases generated to cover the specification are automatically applied to code without manual intervention [2, 3, 5, 10, 16, 27, 38, 51, 56].

- Abstract, model-level test cases are transformed into concrete, code-level test cases after stepwise model-to-model and model-to-code transformations that are performed either automatically or by hand. The concrete tests are then run directly on the code [14, 21, 25, 28, 30, 31, 32, 45, 55].

5.2. Discussion Regarding RQ1.1: What is required of the model-to-code transformation to support the transition from model level tests to code level tests?

Transforming abstract tests that were created from the model into concrete tests on the code can be complicated and challenging. The extent to which the rules and process of turning models to code support the transformation of abstract tests to concrete tests varies. RQ1.1 asks what is required from these transformations. This was discussed in 17 studies that supported our answer to RQ1. While four studies [2, 21, 28, 45] followed the MDE approach, the other 13 studies [3, 5, 10, 16, 18, 25, 27, 30, 31, 32, 51, 55, 56] used various approaches to transformations. These two groups of studies are next described and discussed.

Regarding *studies that addressed MDE*, they all require model-to-model transformations to create test models that form the basis for test generation and transformation down to the code level. For example, Pretschner et al.'s approach [45] transforms extended finite state machines into specifications written

[28] http://xtuml.org/ – accessed in June, 2023.

in a constraint logic programming language from which the test cases are generated. Then, a compiler transforms abstract, model-level, test cases into concrete, code-level, test cases, which are executed on the software under test (written in C).

The approach by Amalfitano et al. [2] requires executable system models, which allow testing models to be automatically generated. The testing models then underlie the generation of test cases at both the model and code levels. The authors work in a Model Driven Architecture[29] development context and employ the Conformiq Designer tool[30] to automatically generate and transform model-level test cases down to code.

In the approach of Fraternali and Tisi [21], a series of model-to-model and model-to-text transformations is applied to automatically generate test cases for models and code. At the Computation Independent Model (CIM), or model level, they transform BPMN[31] models into BPMN-Test metamodels. They utilized WebML[32] at the Platform Independent Model (PIM) level and the WebML-Test metamodel for test cases. Finally, test cases are represented as scripts at the Platform Specific Model (PSM) or code level. Vertical transformations of test cases between the levels (CIM to PIM to PSM) are synchronized with the corresponding model transformations using horizontal mappings.

Similarly to Fraternali and Tisi [21], Lamancha et al. [28] exploited a model-to-model transformation of UML models to UML Testing Profile models, from which test cases for the model level are generated. Subsequently, model-to-text transformations automatically produces test cases in a variety of languages; examples are test scripts that follow the JUnit[33] style.

Regarding *studies that used various approaches to transformations*, Stürmer et al. [51], for instance, addressed the issue of reusing test sets across abstraction levels, suggesting that the specifications of model-to-code optimizations should be available. This allows model elements to be traced to auto-generated code elements, including elements omitted from or inserted into the code through optimizations.

The approach proposed by Veanes et al. [56] needs human intervention for transforming (*i.e.* binding) model elements (*i.e.* action methods in the model) into code elements (methods with matching signatures in the SUT).

In a theoretical study, similarly to what was proposed by Stürmer et al. [51], Kirner [25] also considered optimizations, suggesting that the code generator must conform to a set of rules that are derived from a coverage profile. For that, the author initially defined formal rules based on some coverage criteria (statement coverage, decision coverage, and MCDC), a set of coverage preservation rules, and a set of code optimizations. Based on the formal rules, a coverage profile is created and integrated into a code transformer.

Li and Offutt [31] assumed non-executable behavioral models such as UML state machines, which do not contain details such as objects, parameters, actions, and constraints. They employed the STALE[34] framework to manually write code in the STAL language to define mappings between abstract (model-level) and concrete (code-level) elements, so that abstract and concrete execution paths can be automatically generated by STALE. Example mappings are a UML action mapped to a Java method call, and an initialization of a UML object mapped to a Java object creation. The authors extended that work to use the *skyfire*[35] tool to generate Cucumber test scenarios for different types of applications [32] [30].

In the context of code written in imperative languages (for instance, C) automatically generated from data-flow models (such as in SCADE[36]), while considering data-flow coverage at the model

[29] http://www.omg.org/mda/ – accessed in June, 2023.
[30] https://tinyurl.com/mr3bx8sv – accessed in June, 2023.
[31] http://www.bpmn.org/ – accessed in June, 2023.
[32] https://www.ra.ethz.ch/cdstore/www9/177/177.html – accessed in June, 2023.
[33] http://junit.org – accessed in June, 2023.
[34] http://cs.gmu.edu/~nli1/stale/ – accessed in June, 2023.
[35] http://github.com/mdsol/skyfire – accessed in June, 2023.
[36] http://www.ansys.com/products/embedded-software/ansys-scade-suite – accessed in June, 2023.

DOI: 10.1002/stvr

level, Camus et al. [10] stated that "*it has been verified in practice for complex models that tests covering the model also cover the code generated from that model, except few [sic] systematic cases which are predictable and justifiable*." These *systematic cases* include refinements of the model coverage criteria such as addressing numeric aspects (which would allow performing analysis of singular points) and handling delays (which would allow assessment of sequential logic). With these refinements implemented, the authors state that one could provide formal evidence of model-to-code coverage being preserved in conformance with DO-331 FAQ#11, hence eliminating the need to double-check structural coverage at the code level.

Eriksson and Lindström [18] found that software engineers need explicit model-to-model transformation rules that turn implicit predicates in the model into explicit predicates. Such rules ensure that structural coverage at the model level is preserved during transformation down to the code level. One example is an implicit loop structure at the model level, which is transformed into an explicit loop in the code, with a predicate being introduced. The new code level predicate must be covered, even though it did not exist at the model level.

Durak et al.'s and Koch et al.'s approach [16, 27] relies on the called Scilab/Xcos[37] tool chain to generate test cases for models and re-executing them to test the code. In their approach, the code must be automatically generated from the models by the Scilab/Xcos tool. Amalfitano et al. [3] utilized ordinary spreadsheets to specify test cases that can be executed at both levels. The spreadsheet is automatically processed by a legacy, homemade testing environment. To allow it, the code must be automatically generated from MATLAB/Simulink[38] models, but the authors did not provide further details about how tests are handled in the legacy testing environment.

In Aniculaesei et al.'s approach [5], system requirements must be formalized in the Linear Temporal Logic (LTL) language, which then underlies the generation of test cases. As long as the same set of requirements are used as a basis for modeling the system with the Scade[39] language, both models are assumed to be consistent, and automatic system and test code generation allows for the execution of the test cases at the code level.

Similarly to the approach proposed by Veanes et al. [56], For Vanhecke et al. [55], transforming test cases from model into code initially requires the definition of abstract test cases in XML by utilizing mappings for the interface, actions, and assertions of the system under test. The abstract tests are later transformed into concrete tests that encompass verification steps and sequences of operations that interact with the system under test.

Drave et al. [14] proposed an approach that is modeling environment agnostic in the sense that the approach does not prescribe a modeling environment. To provide the software tooling that supports such approach, the authors used the MontiCore language workbench to develop a domain specific language tool, termed activity diagram (AD) for SMArDT[40] (AD4S). Additionally, the authors developed a parser that can transform ADs in extensible markup language (XML) into AD4S. In this context, the output of a given modeling tool has to be transformed to XML before being parsed into AD4S. AD4S turns the XML representation of models into another textual representation (*i.e.* AD4S-representation), which in turn can be used to derive test cases that can be stored in a format that is executable by functional test execution tools.

In summary, the following sources of information are required to map the test cases across the abstraction levels:

- Formal model-to-model transformations are needed to produce executable test models, typically in the context of MDE development approaches. Such test models are aligned with the system models and underlie the generation of test cases that can be either executed on models as well as code, or exclusively on the code. When tests are executed on the code, the model-level test cases are abstract.

[37] https://www.scilab.org/software/xcos – accessed in June, 2023.
[38] http://www.mathworks.com/products/simulink.html – accessed in June, 2023.
[39] http://www.ansys.com/products/embedded-software/ansys-scade-suite – accessed in June, 2023.
[40] A more in-depth discussion of SMArDT is presented in Subsection 5.3.

DOI: 10.1002/stvr

- The transformation rules performed by model-to-code generators must be explicit to clarify the correspondence between model and code elements. Such transformations include optimizations performed by compilers while transforming model elements into code elements, and the generation of code from model elements that have implicit predicates. The rules can be created either automatically or by hand.

5.3. Discussion Regarding RQ2: How are tests generated from the model specifications (e.g. UML or Simulink)?

Through our investigation of RQ2, we delved into the implications of transforming high-level test models into lower-level test code. We contend that the challenges of model-to-code transformation differ from conventional compiler design, as mentioned in Subsection 3.1. Unlike for traditional compilers for imperative programming languages, there are no established approaches to evaluating the correctness of artifacts generated by model-to-code transformers: transforming models into code requires a more nuanced approach than a straightforward, stepwise transformation from the model representation into code. To gain a better understanding of this transformation process, it is key to understand approaches to developing model-to-code transformers. We surmise that understanding how model-to-code transformers turn models into code can help testers focus on edge cases that are often neglected during model-based testing. With those concerns in mind, we hereafter discuss representative studies[41] that helped us answer RQ2 in four groups, as follows: studies that explored stepwise model transformation but still require human intervention in the last transformation steps [30, 31, 32, 45, 55]; studies that automated test case generation all the way to code generation [2, 5, 7, 12, 13, 14, 18, 20, 21, 25, 28, 29, 38, 45, 49, 53]; studies that relied on executable model and code [10, 18, 28, 51]; and, finally, studies that dealt with test case generation from models in ways that differ from the others discussed in this section [16, 23, 27, 36, 56].

Regarding *studies that explored stepwise model transformation but still require human intervention in the last transformation steps*, these approaches are semiautomatic given that human intervention is required in the final stage of transforming a lower-level model representation into code. For instance, Li and Offutt [32] generated test cases that cover all transitions (edge coverage) and all 2-transition sequences (edge-pair coverage [4]) on UML state machine diagrams. The STALE[42] framework first turns UML state machines into general graphs (model to model). Abstract tests are generated to cover the graphs. The abstract tests include transitions and constraints (based on state invariants). Testers provide mapping rules, which are sequences of method calls to represent transitions in the statechart, which are assembled to transform abstract tests into concrete tests.

Li et al. [30] improved on STALE by further automating the test case generation step. The resulting framework, named *skyfire,*[43] is built on STALE, but generates concrete tests directly from the graphs in the form of Cucumber test scenarios. Skyfire generates test cases that satisfy graph coverage criteria and transforms test cases into Cucumber scenarios. Nevertheless, similarly to AbsCon [55], this approach is semiautomatic given that testers have to write the Cucumber mappings for the generated scenarios.

In another research initiative, the AbsCon (Abstract test cases Concretizer), by Vanhecke et al. [55], was designed to turn abstract tests into concrete ones. The tool's test case concretization process maps assertions and actions in abstract tests to verifications and sequences of operations (*i.e.* concrete tests), respectively, that exercise the SUT through the test API. However, the process of turning abstract tests into concrete test scripts is not fully automated, it requires tester intervention. Specifically, before turning assertions and actions, which are defined in XML, into concrete tests in Python, testers must provide the following additional information: the test API model for executing the SUT, the path to the Python files that implement the SUT model and the mapper for the chosen API, and a CSV file with input values (*i.e.* test case values).

[41] We did not describe all studies to avoid too much overlap with the descriptions we did for the other RQs.
[42] `http://cs.gmu.edu/~nli1/stale/` – accessed in June, 2023.
[43] `http://github.com/mdsol/skyfire` – accessed in June, 2023.

Softw. Test. Verif. Reliab. (2023)
DOI: 10.1002/stvr

Regarding *studies that automated test case generation all the way to code generation*, some approaches, such as the one by Conrad [12], ensure that models are transformed into *functionally equivalent* code. Conrad, for example, exploited a testing-based approach to gauging the functional equivalence of the model and the resulting code. In a previous work, Conrad et al. [13] emphasized test case generation in the context of back-to-back testing of electronic control unit (ECU) software. This type of test emphasizes the equivalence between the test object (*i.e.* model) and its reference (*i.e.* generated code).

Fraternali and Tisi [21] developed a multi-level test generation approach, and a transformation framework to align two streams of transformation, from computation independent models to code, and from computation independent test specifications to executable test scripts. The test scripts are updated by mappings that can be applied when model changes take place.

The approach proposed by Lamancha et al. [28] is stepwise in the sense that it applies model-to-model transformations and then model-to-text transformations. The approach turns high-level UML 2.0 representations (*i.e.* sequence diagrams) into test case scenarios that conform to the UML Testing Profile 2 (UTP2), and then model-to-text transformations are applied to the UTP2 models to render these models into text (*i.e.* code). According to the authors, the model-to-text transformation step allows for the generation of test cases in a variety of programming languages owing to the fact that it is implemented with MOFScript, which is an OMG standard.

Tekcan et al. [53] also devised a twofold approach to turning a high-level representation into executable test code. Specifically, in the proposed user-driven test case generation approach test cases are first represented as states and state transitions in XML files, and then these XML files are transformed into Python scripts.

Drave et al. [14] developed a method to manage requirements, design, and test in automotive industry. The specification method SMArDT leverages model-based software engineering techniques with the aim of mitigating the deficiencies of the established V-Model. The method is based on the premise that consistency checking between layers and test case generation (and regeneration) helps developers and testers cope with the bureaucracy imposed by the classical V-Model. The authors posited that consistency among specification artifacts between layers enables automatic transformation of test cases to lower levels. To realize the method in a modeling environment agnostic fashion, the authors put together a configurable tool chain that can turn functional requirements modeled using activity diagrams, state charts, sequence diagrams, and internal block diagrams from various formats into executable test cases for various output formats.

Some researchers have also turned their attention to the formal verification technique of model checking to derive test cases automatically. Essentially, model checking hinges on the capability of model checkers to exhaustively probe into the state space of the SUT and generate test cases that are based on traces or counter-examples of properties specified by the SUT's model. Therefore, model checking based test generation is built on the assumption that by thoroughly exploring the state space it is possible to achieve complete coverage and determine unreachability of model elements. AutoMOTGen, by Mohalik et al. [38], is an example of tool that has been developed to automatically generate tests from Simulink/Stateflow models using model checking. Aniculaesei et al. [5] sought to explore model checking for the automatic generation of test cases based on requirements for test cruise control systems for the automotive industry. Essentially, the authors devised an approach in which system requirements are formalized into Linear Temporal Logic (LTL) language requirements, which is then used to generate test cases. Additionally, if the same set of requirements underlies the modeling of the system with the Scade language, both models are assumed to be consistent, hence automatic system and test code generation allows for the execution of the test cases at the code level.

Some *studies rely on executable models and code*; these studies assume that test cases can be generated and run on the models, and that the resulting test cases can be transformed into code or directly executed, depending on the syntax. The approaches investigated in such studies include test code generators whose inputs and outputs are executable models. Stürmer et al. [51], for instance, devised an approach in which test cases comprise a test model in Simulink/Stateflow and the input values are called test vectors. Input values are used to check the functional equivalence between the model under test and the auto-generated C code. As mentioned, the authors built a tool (*i.e.* Mtest) to

DOI: 10.1002/stvr

map model elements to code so test cases can be executed in both artifacts, allowing for optimizations during code generation, as long as the optimizations are clearly specified. This allows the model elements to be traced to code, including changes by the optimizer. The model and the code generated from it can be considered *functionally equivalent* if they both lead to *compatible* output data when executed with the same input data [51].

Lamancha et al. [28], as another example (and previously described in this section), devised a framework that, after transforming UML models into UML Testing Profile[44] models, can derive the source code of the test cases from the testing profile models.

With respect to *other studies* that addressed research on test case generation from models, it is worth noting that some model representations used internally may not be readily executable. Nonetheless, integrated tool environments for model exploration and validation, test case generation, and test execution against an auto-generated implementation of the system under test can be developed. An example of such an integrated tool environment is Spec Explorer, by Veanes et al. [56], which is a tool for testing reactive, object-oriented software systems. In the context of Spec Explorer, the system's behavior is described by models written in the language Spec# (an extension of C#) or AsmL. Fundamentally, a model in Spec# defines the state variables and update rules of an abstract state machine. Spec Explorer employs algorithms similar to those of explicit state model checkers to explore the machine's states and transitions, which results in a finite graph containing a subset of model states and transitions. This graph-based representation is then used for test case generation. Spec Explorer allows for two test case execution modes: offline (*i.e.* when test generation and execution are seen as two independent phases) and online (*i.e.* which integrates test generation and test execution into a single phase). Online execution incorporates a sort of feedback loop in which immediate results from test execution are used to further guide the test generation process. Thus, as pointed out by the authors, executable models are not crucial to developing tools that can further refine test case generation.

Search-based testing has also been explored in the context of MBT [23, 36]. Matinnejad et al. [36] investigated how a search-based technique based on random search, adaptive random search, hill climbing and simulated annealing algorithms can be used to identify worst-case test scenarios which are utilized to generate test cases for requirements that characterize the behavior of continuous controllers. Similarly to Matinnejad et al. [36], Kalaee and Rafe [23] examined how search algorithms can be applied to generate test sets from graphs. The proposed approach is tailored to systems that are specified as graph transformations.

Koch et al. [27] and Durak et al. [16] designed tools to support the X-in-the-loop testing pipeline, and both tools generate test cases from Scilab/Xcos models. At the model level, test cases are automatically generated for individual and integrated components. The authors refer to test generation for integrated components as model-in-the-loop (MIL). These test cases hinge on what the authors termed "a number of plausible scenarios" which are derived from decision trees that formally represent the integrated models. The results of the tests performed at the model level are subsequently used as "reference" for software-in-the-loop (SIL) testing of auto-generated code.

To summarize, we found that most of the selected studies deal with test case generation from models. However, there are important differences in the way high-level models are turned into lower-level test cases and how the resulting test cases are used:

- Some test case generation approaches emphasize model-to-model transformations, thus the last step to transform to code has to be semiautomatic. Testers have to bridge the gap between the lowest model level and code by specifying how certain model elements should be transformed into code, for example, by mapping a graph to a sequence of method calls.

- Some approaches automate test case generation all the way to code generation by performing stepwise model refinements until they reach a low-level model representation that is suitable for code generation.

[44] https://www.omg.org/spec/UTP/1.2/About-UTP/ – accessed in June, 2023.

DOI: 10.1002/stvr

- As long as both model and code are executable, another common approach entails deriving test cases from models and then applying these test cases to the auto-generated code. Some approaches utilize the model-level test cases to create low-level test cases that can be executed to test the auto-generated code.

5.4. Discussion Regarding RQ3: How does the coverage of the model produced by abstract tests relate to the coverage of the code for the corresponding concrete tests?

When models are transformed to code, whether automatically or by hand, it is imperative that the behavior defined in the model is preserved in the code. Likewise, even though computing the coverage of artifact-specific constructs with particular tools may result in diverging coverage results, it is important that we maintain high degree of coverage when transforming test cases from the model to the code level.

Models and code use structural elements to represent the underlying logic, albeit at different levels of abstraction. Models, for example, use high-level structures such as activity diagrams to represent the steps and branching logic (*i.e.* decisions) involved in a specific behavior. Code represents the procedural logic that manipulates data and implements specific behavior using lower-level constructs. As a result, the similarity between models and code is found in their use of structures to represent the logic of the system. We believe that to answer RQ3, it is necessary to consider the following points. Firstly, can model-level decision coverage results be extrapolated to branch coverage at the code level? Secondly, are there any high-level constructs that represent behavior in an implicit fashion? Implicit behavior at the model level can interfere with model-to-code transformations, and as a result, implicit behavior at model level may not be included in the resulting code representation. This can impact coverage when models are transformed into code. Finally, while there is some overlap in how models and code represent decisions, is this overlap sufficient to result in the same number of test requirements? In order to answer RQ3, we have examined these subquestions in the context of the empirical research presented in the selected studies. We framed the aforementioned subquestions as follows:

1. Given the structural similarities between code and models, can we expect correlation between model and code coverage?

2. Models tend to have some implicit behaviors, for example, conditional behavior that does not appear as predicates. What are the implications with respect to coverage when we transform the models into code?

3. What happens to the number of test requirements when we transform models to code? Does the code have more, fewer, or the same number of test requirements?

We found that 13 of the selected studies address RQ3 [2, 5, 7, 14, 18, 20, 23, 32, 38, 45, 49, 53, 56]. In what follows, we describe and discuss studies that elaborated on different aspects related to the implications of applying model-based tests to code automatically generated from models [2, 5, 7, 18, 20, 32, 38, 45], and studies that only briefly mentioned coverage at both abstraction levels (namely, model and code) [14, 23, 49, 53, 56]. Then we draw answers to the three aforementioned subquestions.

Regarding *studies that elaborated on different aspects related to the implications of applying model-based tests to code automatically generated from models*, Baresel et al. [7] studied the relation between requirements and structural coverage at both the model and the code levels. The authors report on empirical coverage results for model (Simulink/Stateflow) and code (C) of three functional modules of an automotive system. They found a strong correlation between model and code coverage in terms of achieved percentages of coverage. Other studies we describe in the sequence, however, found a substantial difference in the number of test requirements when performing model-to-code transformations, particularly when models have implicit behaviors that are transformed into decisions and loops in code that have explicit predicates. The introduced predicates create new code-level test requirements.

DOI: 10.1002/stvr

Pretschner et al. [45] argued that it is key to make behavior explicit at model level (*i.e.* akin to the introduction of model-level predicates to make implicit semantic assumptions explicit). The results of their experiment would seem to suggest that, when behavior is made explicit at model level, automatically generated test sets are able to uncover as many faults as handcrafted model-based test sets with the same amount of test cases. In terms of test requirements, Pretschner et al. noted that the implementation (C code) contained 47% less decision/condition (C/D) required transitions when compared to the modeled system (System Structure Diagram and Extended Finite State Machines). Furthermore, the results show that there is a moderate positive correlation between model and implementation C/D coverage, a moderate positive correlation between C/D implementation coverage and fault detection, and a strong positive correlation between C/D model coverage and failure detection.

Eriksson et al. [20] addressed the issue of implicit conditional behaviors at the model level. They devised model-to-model transformation rules that turn implicit predicates into explicit predicates at the model level, thereby ensuring that structural coverage achieved at model level is preserved at the code level. These transformation rules resulted in near 100% code-level MCDC coverage. Thus, although model-level test cases generated from the original model may not be enough to guarantee code-level coverage, they can be augmented in clearly defined ways to achieve coverage. Regarding the number of test requirements, for the original artefacts (*i.e.* without applying the devised rules) they found 67% additional logic-based test requirements from the code compared to the design model, whereas this percentage dropped to near 0% when the rules were applied.

By building on previous work [19], Eriksson and Lindström [18] devised an approach that addresses test generation for executable UML (xtUML) models. It includes two new logic-based testing coverage criteria for models (namely, *all navigation* and *all iteration*). The new criteria aim at covering the implicit predicates that logic-based criteria miss. For example, by using only predicate-based criterion in one of the six applications addressed in their previous study [19], the number of test requirements increased 51% when xtUML models are transformed to code. In Eriksson and Lindström's approach, coverage measurement at the model level is enabled by introducing model-level predicates that capture predicates that would appear during model-to-code transformations. The results from a single application demonstrated that coverage measured at the model level can accurately predict coverage at the code level. This is particularly important for logic-based testing, since coverage at the code level is often required.

Mohalik et al. [38] shed some light on how AutoMOTGen compares to Reactis (which is a commercial tool that implements a combination of random input-based and guided simulation-based techniques for test case generation) in terms of test coverage. According to the results of industrial case studies, the test case generation techniques employed by both tools can be seen as complementary. Specifically, AutoMOTGen performs better (*i.e.* achieves higher coverage) for about one third of the cases, while Reactis shows higher coverage for about other third of the cases. As for the rest of the cases, the coverage obtained by both tools seems to be roughly equal. A closer inspection of the results indicates that when models have more logic (*i.e.* switches and delay types of blocks) AutoMOTGen performs better than Reactis. As for models with more blocks of mathematical operations, Reactis seems to perform better in comparison to AutoMOTGen. This indicates that the technique implemented by AutoMOTGen is more suitable for covering paths with logical constraints. Additionally, when approximations have to be applied in order to handle complex mathematical operators, the coverage achieved by the test cases generated by AutoMOTGen suffers. Therefore, the authors postulate that AutoMOTGen and Reactis should be used together to achieve better coverage and unreachability guarantees.

Amalfitano et al. [2] compared the model coverage achieved by the test cases at the model level with the coverage obtained by the test cases when run against the generated code. They found differences between model coverage on state machines and code coverage. They ran two test sets on four state machine models and their code. The test sets reached 100% coverage on states and transitions, but statement coverage varied from 48% to 75% and branch coverage from 25% to 52% on the code. Amalfitano et al. gave three main reasons for these differences: *(i)* the code generator added extra code for exception handling and debugging; *(ii)* model coverage was not enough to

DOI: 10.1002/stvr

guarantee code coverage; and *(iii)* the design of the models play a major role in the quality of the generated test cases. Their results indicate the major source of differences was code that added behavior that was not included in the model, even without explicitly showing the absolute number of test requirements at both abstraction levels.

The approach proposed by Li and Offutt [32] renders state machine diagrams into general graphs, which are then used to generate abstract tests. The abstract tests are generated so as to satisfy graph coverage criteria: edge and edge-pair coverage. According to the experiment results, edge-pair coverage tests were not significantly stronger than the edge coverage tests. The authors believe that this is the case because edge-pair coverage did not entail many more mappings (*i.e.* test inputs) than edge coverage.

Aniculaesei et al. [5] evaluated the coverage in terms of a mutation score. In their experiment, which used a single subject, the test set generated by the SCADE toolchain was able to kill roughly 21% of the mutants (a very low mutation score of 0.21). After analyzing the causes that might have contributed to this low mutation score, the authors concluded that the system targeted in the experiment was only partially represented through LTL specifications.

Regarding *studies that only briefly mentioned coverage at both the model and the code levels*, Spec Explorer, by Veanes et al. [56], derives test cases from graph-based models (dubbed *model automata*). The resulting test cases are generated in hopes of either providing some sort of coverage of the state space, reaching a state (*i.e.* node) satisfying some property, or traversing the state space randomly, likewise the coverage of the corresponding implementation under test which may be a distributed system consisting of subsystems, a (multithreaded) API, a graphical user interface, etc.

Shokry and Hinchey [49] simply reported that randomly generated model-level tests provide low code coverage (around 32%). These findings were based on their own experience with the X-in-the-loop testing process. In a similar level of details, in a study in which test cases were first represented as states and state transitions in XML files, and then transformed into Python scripts, Tekcan et al. [53] mentioned coverage-related results without defining what they mean by "coverage".

Drave et al. [14] reported on the results of an experiment in terms of the fault-finding effectiveness of the proposed MBT as opposed to structural coverage-related results. More specifically, the authors carried out a case study to compare model-based test cases derived in the context of the tool chain environment that realizes SMArDT and manually created test cases. According to their results, the MBT approach generated test cases had a higher fault coverage (*i.e.* detected more faults) than the traditional hand-crafted test cases. The MBT approach was especially effective at generating test cases that uncover faults caused by inconsistent requirements. Nevertheless, neither the traditional nor the model-based test cases uncovered all faults. The authors do not elaborate on the structural coverage achieved by neither test set.

Kalaee and Rafe [23] proposed a test case generation approach for graph transformation systems (GTS) that utilizes model simulation and search-based techniques. In this context, coverage is analyzed in terms of the all def-use criterion: specifically, data flow coverage criteria is determined by data dependencies between nodes in the graph. Initially, the approach creates a model of the GTS using graph transformation rules. The model is then simulated to generate the first test cases. Following that, the initial test suite is optimized through search-based techniques. The authors conducted an experiment to evaluate the effectiveness of their test case generation approach (using different meta-heuristic algorithms). According to the results of the experiment, the generated test sets can cover a significant portion of the GTS while keeping test generation cost low: on average, the best algorithm achieved 98.25% coverage, and the second best achieved 96.50% coverage.

By revisiting RQ3, we draw the following answers to its three subquestions, respectively:

- *Empirical studies have shown a strong correlation between decision coverage at the model level and branch coverage at the code level.* This answer relies on the few studies that have shed light on the implications that arise from the similarities between models and code. Often, these implications are discussed either in the light of the problem of preserving structural code coverage when transforming model into code, or in terms of the correlation between a high-level (*i.e.* model-based) testing criteria and a lower-level criteria (*i.e.* based on notions

DOI: 10.1002/stvr

of structural code coverage). From a model-to-code transformation viewpoint, when looking at the effect of model to code transformations on the test artifacts and the number of test requirements, it would seem that as the number of test artifacts increases, the test requirements for logic-based coverage criteria also increase accordingly [19]. From a criteria comparison standpoint, a study suggests that there is a need to combine high-level testing criteria (which are based on test requirements) with logic-based criteria [18]; in that study the overlap between these criteria is not straightforward since the test requirements come from different sources. Another study that empirically investigates the relationship between structural model and code coverage [7] showed that there is a strong correlation between decision coverage on model level and branch coverage on code level.

- *We found that models can have implicit test requirements represented by implicit predicates at the model level and these predicates are not affected by logic-based criteria applied at the model level. Nevertheless, studies suggest that deterministic rules applied during model-to-code transformation can make these predicates explicit, resulting in better structural coverage at the model level with relatively low additional testing effort.* More specifically, some studies posit that models tend to have implicit test requirements [19, 20]. Specifically, implicit predicates at model level represent hidden controls and loops, which account for most of the implicit behavior in models. Therefore, the main implication with respect to model-to-code transformations and test coverage is that these implicit predicates are not affected by logic-based criteria, so they do not contribute any test requirements when such criteria are applied at model level. However, studies show that the hidden behavior in models can be made explicit by deterministic rules that can be applied by a model-to-code compiler during transformation. The results of these studies suggest that by making implicit behavior explicit it is possible to achieve structural coverage at model level that is closer to the coverage obtained at code level. Additionally, most implicit behavior when turned explicit tend to result in single-clause predicates, thus few extra test cases are needed and the ensuing test design activity is cheap.

- *Few studies reported on the increase in test requirements when implicit behavior in modeling structures is made explicit during model-to-code transformation.* In particular, only three studies [18, 20, 45] provided details about the number of test requirements when models are transformed to code. The three studies addressed turning implicit behavior present in modeling structures (*e.g.* predicates) into explicit behavior at the code level, and how this leads to an increase in the number of test requirements in the code when compared to the corresponding model. Overall, these studies introduced approaches for making behavior explicit either through transformation rules to be applied to the model before it is transformed to code [20], coverage criteria for models [18], or by forgoing modeling structures that omit logic at the model level [45].

5.5. Discussion Regarding RQ4: Which are the applied technologies and which are the software development tasks focused by studies that address mapping of tests across model and code levels?

The discussion and conclusion for RQ4 are based on study classifications that rely on the MBT technology (*e.g.* , modeling language and tools) and software development tasks (*e.g.* , modeling and test coverage calculation). Note that even though the elements of the taxonomy (*e.g.* , the input and output languages) were defined in advance (as detailed in Section 3.4), the list of elements inside each category was constructed during the analysis of the selected studies. In other words, the list of elements grew over the course of our systematic review of the literature. At the end of the study analysis and data extraction, we revised the resulting categories to avoid ambiguity and remove duplicates.

The results discussed in this section encompass (i) the modeling language, herein referred to as *input* language (Figure 3 and Table V), (ii) the source code or test specification language – *i.e.* the *output* language – used to encode artifacts that are generated with either automatic or manual

Softw. Test. Verif. Reliab. (2023)
DOI: 10.1002/stvr

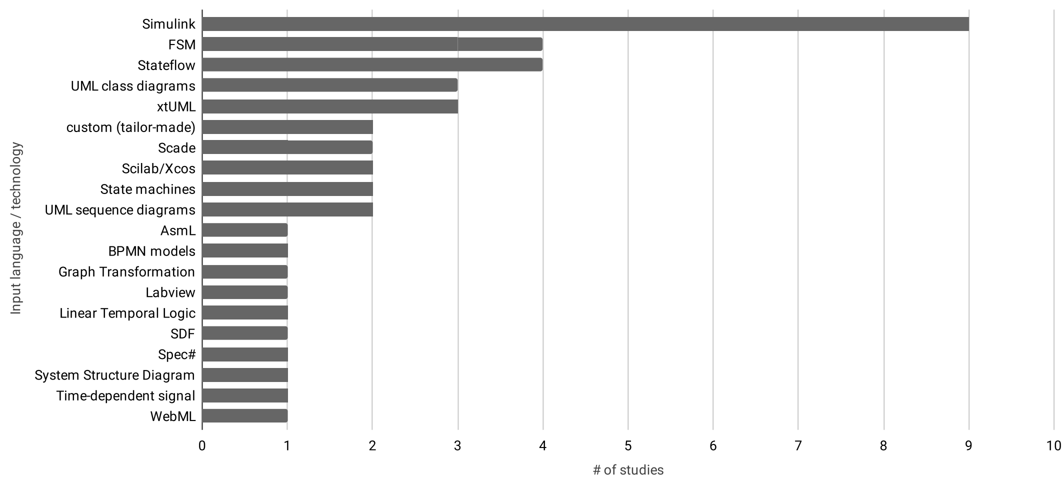

Figure 3. Input languages (for modeling).

model-to-code or model-to-model transformations (Figure 4 and Table VI), and (iii) the goal of the tools and frameworks used in the studies (Figure 5 and Table VII). Note that there is some overlapping in the results shown in the charts and tables, given that some studies used combined technologies and used tools for varying purposes. For instance, Baresel et al. [7] and Mohalik et al. [38] adopted Simulink and Stateflow as languages for creating models. As another example, Aniculaesei et al. [5] employed tools for modeling, test generation at the model level, and computing test coverage at the code level.

Figure 3 shows the number of studies in which a given language was used for modeling purposes. Table V lists the respective studies. Simulink[45] was the most used (nine studies), followed by Finite State Machines (FSMs) and Stateflow[46] (four studies each). These three languages are more mature and have more automated support, so we were not surprised that they are widely addressed.

Figure 4 summarizes the classification of studies with respect to the output language. The respective studies are listed in Table VI. The results for this study classification reflect the numbers presented in Figure 3. For instance, the toolkits that support Simulink- and Stateflow-based modeling usually support automatic generation of C code, which was the case of seven studies. FSMs are commonly employed to represent states of objects in object-oriented (OO) systems that are further implemented in C++ (three studies), Java (two studies), and Python (two studies) languages.

Figure 5 displays the number of studies that utilized tools and frameworks for specific tasks in the MBT process. Table VII lists the respective studies. Examples are modeling (with 13 occurrences in our selected studies), test generation at the model level (twelve occurrences), and test coverage calculation at the code level (five occurrences).

In summary, for studies that address, to varying degrees, the mapping of abstract tests to concrete tests:

- Simulink and Stateflow, either individually or in combination, are by far the most commonly used input languages for system modeling.
- C and C++ are the most explored output languages for model-to-code transformation, thus corroborating the findings regarding the input language.
- Tools are mostly used for the modeling activity, generation of abstract tests, and test coverage computation (either at code or model level).

[45] http://www.mathworks.com/products/simulink.html – accessed in June, 2023.
[46] http://www.mathworks.com/products/stateflow.html – accessed in June, 2023.

DOI: 10.1002/stvr

Table V. List of studies with respect to the input languages.

Input Language	# of studies	References
Simulink	9	[3, 7, 12, 25, 29, 36, 38, 49, 51]
FSM	4	[2, 30, 32, 45]
Stateflow	4	[7, 12, 38, 51]
UML class diagrams	3	[14, 28, 55]
xtUML	3	[18, 19, 20]
custom (tailor-made)	2	[9, 55]
Scade	2	[5, 10]
Scilab/Xcos	2	[16, 27]
State machines	2	[31, 53]
UML sequence diagrams	2	[14, 28]
AsmL	1	[56]
BPMN models	1	[21]
Graph Transformation Specification	1	[23]
Labview	1	[49]
Linear Temporal Logic	1	[5]
SDF	1	[29]
Spec#	1	[56]
System Structure Diagram	1	[45]
Time-dependent signal	1	[13]
WebML	1	[21]

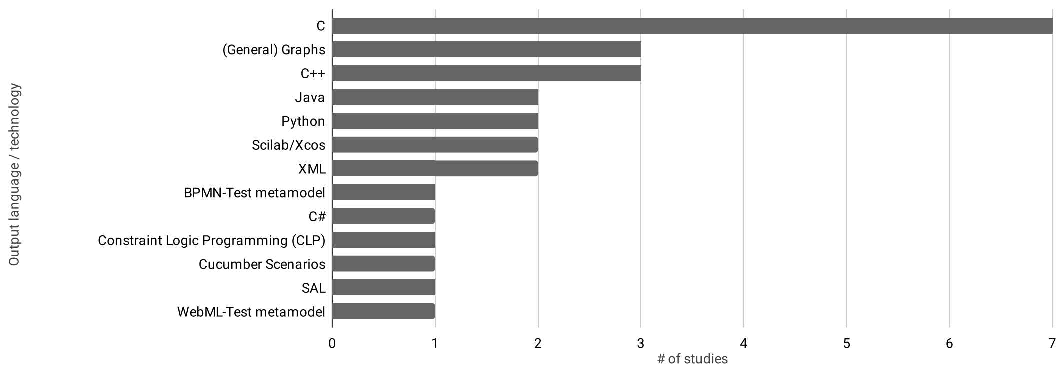

Figure 4. Output languages (for source code).

Table VI. List of studies with respect to output languages.

Output Language	# of studies	References
C	11	[3, 5, 7, 10, 12, 16, 25, 27, 29, 36, 51]
(General) Graphs	3	[30, 32, 56]
C++	3	[18, 19, 20]
Java	2	[28, 31]
Python	2	[53, 55]
Scilab/Xcos	2	[16, 27]
XML	2	[14, 55]
BPMN-Test metamodel	1	[21]
C#	1	[56]
Constraint Logic Programming (CLP) language	1	[45]
Cucumber Scenarios	1	[30]
SAL	1	[38]
WebML-Test metamodel	1	[21]

DOI: 10.1002/stvr

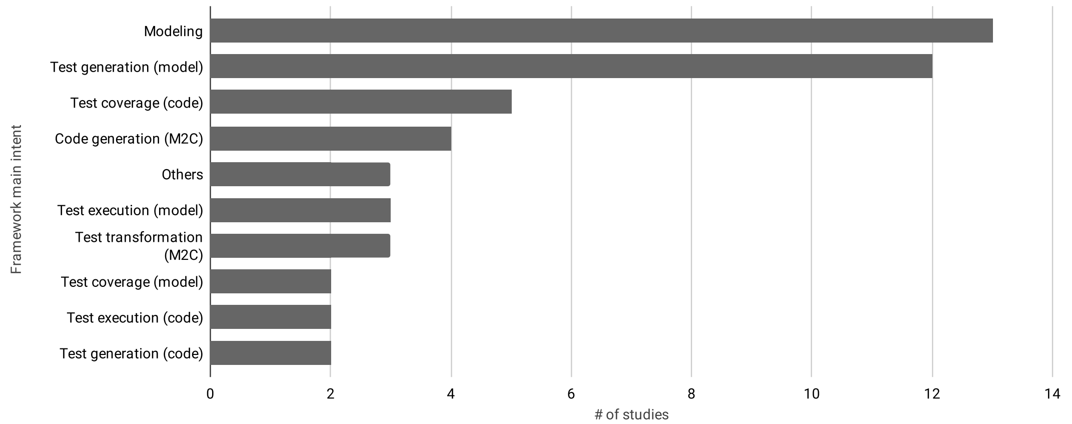

Figure 5. Goal of used tools and frameworks.

Table VII. List of studies with respect to the goal of used tools and frameworks.

Goal of used tools and frameworks	# of studies	References
Modeling	13	[5, 10, 16, 18, 20, 21, 23, 27, 28, 28, 29, 45, 56]
Test generation (model)	12	[5, 7, 12, 14, 30, 31, 32, 38, 51, 53, 55, 56]
Test coverage (code)	5	[2, 5, 7, 18, 20]
Code generation (M2C)	4	[2, 12, 29, 51]
Others	3	[12, 13, 51]
Test execution (model)	3	[3, 12, 36]
Test transformation (M2C)	3	[2, 28, 51]
Test coverage (model)	2	[10, 12]
Test execution (code)	2	[53, 55]
Test generation (code)	2	[7, 51]

5.6. Summary of Findings

A summary of the main findings of our study is provided in Table VIII. Regarding the established RQs, they complement each other given that the focus of our research is on investigating the consequences of transforming higher-level test models into lower-level test code. The RQs emphasize transformation details because we believe that by having a more complete understanding of associated nuances testers can have a better idea of how to improve test cases at both model and code levels. On the one hand, RQ1 and RQ1.1 are concerned with shedding some light on how high level test cases are rendered into lower-level test cases (*i.e.* code level). On the other hand, given that it is important to understand current approaches for developing model-to-code transformers and how the approaches turn models into code, RQ2 helped us summarize current knowledge regarding how model-level tests are derived from models. Furthermore, when models are transformed to code, whether automatically or by hand, it is important to maintain a high degree of coverage across the software abstraction levels, and this is addressed in our analysis concerning RQ3. Finally, the results for RQ4 establish a connection between the studies that corroborated the discussion and conclusions regarding the other RQs and the technologies employed in those studies.

6. THREATS TO VALIDITY

We identify three types of threats to the validity of our study: (i) researcher bias during study selection, (ii) inaccurate data extraction, and (iii) researcher-induced bias during data synthesis. Data from a decade of SLRs in Software Engineering [59] indicates that threats to validity are usually described

DOI: 10.1002/stvr

Table VIII. Overview of the results and findings for each RQ.

RQ	Summary of Key Findings
#1	• Test cases are transformed from a model representation into code using specialized tools. Moreover, software specifications undergo transformation into code. The test cases to cover the specifications are then applied to the code without the need for any manual intervention. • Transforming model-level test cases into code-level test cases involves stepwise model-to-model and model-to-code conversions, which can be performed either automatically or manually.
#1.1	• Model-to-model transformations are employed to generate executable test models. The resulting test models are closely aligned with system models and serve as the foundation for generating test cases that can be run on models and code. • It is imperative that the transformation rules realized by model-to-code generators be explicit, allowing for the identification of the relationship between model and code elements.
#2	• Most of the studies focus on test case generation from models. There are differences in how models are transformed into lower-level test cases and their subsequent utilization: – Some approaches prioritize model-to-model transformations, requiring a semiautomatic step to convert the model into code. – Some approaches automate test case generation through stepwise model refinements, gradually achieving a representation that aligns with the code's abstraction level. – When both the model and code can be easily executed, the prevalent approach involves deriving test cases from models and subsequently executing these test cases on the auto-generated code.
#3	• Studies have provided evidence of a strong correlation between decision coverage at the model level and branch coverage at the code level. • Models contain implicit test requirements represented by implicit predicates. These predicates are not covered by logic-based criteria applied at the model level. • Few studies reported on the increase in test requirements when implicit behavior in modeling structures is made explicit during model-to-code transformation.
#4	• Simulink and Stateflow are by far the most commonly used input languages for system modeling. • C and C++ stand out as the two most extensively explored output languages for model-to-code transformation.

in four major categories: construct validity, conclusion validity, internal validity, and external validity. We organize this section according to these four categories.

DOI: 10.1002/stvr

6.1. Construct Validity

Our main concepts are model-based testing and different approaches to transforming model-level test cases into code. To determine the correct interpretation of these concepts, we checked their definitions within the context of our study and discussed them among the authors to reach a consensus. As a result, the categorization schemes we generated during data analysis stem from how we interpreted the concepts involved in our study. However, we cannot completely rule out the possibility that some primary studies might have been misclassified. To cope with this issue, the proposed categorization schemes underwent several reviews by the authors to maximize confidence. We also provide all details in our companion spreadsheet.[47]

6.2. Conclusion Validity

Conclusion validity is primarily concerned with the degree to which the conclusions we reached are reasonable. In our study, we answered our RQs and drew conclusions based mostly on information extracted from the primary studies. Thus, the conclusion validity issue lies in whether there is a relationship between the number of studies we selected and current research trends in the subject area. We cannot fully rule out this threat because the broad nature of our study makes data identification, extraction, and synthesis susceptible to bias.

Particularly regarding data identification, the snowballing process should end when no new studies are found in the search iterations [58]. For the original search, executed in 2018, we performed the search in three depth levels for both backward and forward snowballing variants. We considered this number of rounds as a stopping criterion to make the study feasible in terms of effort and number of studies needed to draw conclusions regarding our research questions. In the search update performed in 2020, we intended to identify new citations to the already selected studies. Moreover, the most recent update encompassed the analysis of studies suggested by experts. In both cases, we did not restart the snowballing process in several depth levels. These different search procedures may be seen as a possible threat to the results.

6.3. Internal Validity

The main threat to the internal validity of our study is missing relevant studies. Naturally, systematic studies of the literate can be carried out in different ways. In practice, different strategies for searching the literature achieve different coverages. We applied snowballing to mitigate this threat and achieve a good coverage. According to Wohlin [58], snowballing is an effective alternative to the utilization of database searches.

Another potential threat to the internal validity of our study is researcher bias during study selection. We took a sequence of steps to prevent research bias during data extraction. First, information extracted from the primary studies was discussed among the researchers. Second, in hopes of ensuring that the three researchers in charge of data extraction had a clear understanding of the extracted information, we pilot-tested many aspects of the data extraction spreadsheet among all the authors. The results of the pilot were then discussed to reach a consensus.

6.4. External Validity

A potential threat to the external validity of our study stems from determining whether the selected primary studies are representative of all the relevant efforts that have been carried out in the subject area. We mitigated this issue by following a rigorous search process. Despite the fact that we only selected studies written in English, we believe the set of primary studies we selected include enough valuable information to provide researchers with an extensive overview of the subject area.

It is also worth mentioning that several primary studies did not include the information we needed to fill out the extraction spreadsheet, and, consequently, we often had to infer the missing information

[47] https://doi.org/10.5281/zenodo.8113394

during data synthesis. For instance, some studies do not mention the degree of traceability from model to code provided by their proposed approaches.

Another potential external threat to the validity of this study is the time frame of the data utilized in this investigation. Specifically, the study selection process was completed (*i.e.* last updated) in 2020, thereby potentially limiting the generalizability of our findings. Since then, the field of research may have evolved slightly as a result of new studies and evolving perspectives, potentially altering the overall landscape of the research area. As a result, it is important to acknowledge that our findings may not fully capture the most current advancements in the field, thus warranting caution in interpreting them.

7. RELATED WORK

As described in Section 3, a *secondary study* is a study that surveys or otherwise aggregates results, such as a survey or SLR. We have identified several secondary studies that are related to ours, but that have a different scope or different goals. We have categorized these into four topics: (1) testing at the model level [17, 43], (2) testing of model transformations [1], (3) model-based testing [8, 22, 33, 44, 47], and (4) testing non-testable systems [42].

For topic 1, testing at the model level, Elberzhager et al. [17] focused on MATLAB[48] and Simulink[49] models, while Paul and Lau [43] investigated the MCDC coverage criterion.

Elberzhager et al. [17] reported results from studies about quality assurance, specifically, analysis and testing techniques, for MATLAB[48] and Simulink[49] models. Their research questions also addressed supporting tools and how the techniques are assessed. Elberzhager et al. retrieved their primary studies through an automatic search on two indexed databases (ACM Digital Library[50] and IEEE Xplore[51]) and one search engine (Elsevier Scopus[52]). In total, the authors selected 44 studies published starting from 1990. Their main finding was that some of the identified techniques have been applied in a combined manner, but more research is necessary to allow for a deeper integration and effective quality assurance of MATLAB and Simulink models.

Paul and Lau [43] performed an SLR to examine how the different forms of MCDC [11] have been studied in literature. MCDC is applied to certify the implementation of safety critical parts of avionics software [46], patient monitoring systems in hospitals, and power control systems for nuclear power plants. They found studies in six digital libraries and one indexing service: ACM Digital Library,[50] Citeseer,[53] Elsevier Online Library, IEEE Xplore,[51] Springer Online Library,[54] Wiley InterScience,[55] and Web of Science.[56] Among the 70 selected studies, 54 discussed a variant of MCDC, with a total of seven MCDC variants being identified. Apart from presenting a discussion of the state-of-the-art of MCDC according to previous studies, Paul and Lau also identified a new form of MCDC, which they termed *Unique-Cause* and *Restricted Masking* (UCRM) MCDC. UCRM is a formalism of Ammann and Offutt's [4] advice to strive for RACC, but settle for CACC when RACC is infeasible. They also carried out an empirical study to compare the fault detecting ability of UCRM to existing MCDC variants. Their results suggested that UCRM outperforms other MCDC variants in terms of fault detection. Neither Elberzhager et al. [17] nor Paul and Lau [43] considered issues related to test mapping and coverage across software abstraction levels like this article.

For topic 2, testing of model transformations, Abade et al. [1] presented an SLR to characterize structural testing approaches for testing model-to-text transformations. Abade et al.'s main goal was

[48] http://www.mathworks.com/products/matlab.html – accessed in June, 2023.
[49] http://www.mathworks.com/products/simulink.html – accessed in June, 2023.
[50] http://dl.acm.org/ – accessed in June, 2023.
[51] http://ieeexplore.ieee.org/Xplore/home.jsp – accessed in June, 2023.
[52] http://www.scopus.com/home.uri – accessed in June, 2023.
[53] http://citeseerx.ist.psu.edu – accessed in June, 2023.
[54] http://link.springer.com/ – accessed in June, 2023.
[55] http://onlinelibrary.wiley.com/ – accessed in June, 2023.
[56] http://www.webofknowledge.com/ – accessed in June, 2023.

DOI: 10.1002/stvr

to characterize how complex data has been defined and utilized in that context, as opposed to our goal of evaluating the implications of transforming model-level test cases into code. They selected nine primary studies selected from an automatic search performed in two indexed databases (ACM Digital Library[50] and IEEE Xplore[51]), and one search engine (Elsevier Scopus[52]). Additionally, the authors analyzed a set of journals and conference proceedings related to model-driven development, published between 2008 and 2013. Their main findings were that two behavior patterns, the Visitor Pattern and the Template Method, were the most common, and that the characterization of complex data was usually neglected.

Li et al. [33] carried out a survey of MBT tools. Differently from our study, the discussion presented in their study is centered mainly on test case generation. Specifically, the authors discuss test data and script generation, without addressing how the propagation of test generation decisions made at model level might have an impact on the resulting test code.

Four reviews addressed topic 3, model-based testing (MBT): one SLR [47] and three systematic mapping studies (SMS) [8, 22, 44]. Their goals differed from ours in that they did not examine issues related to test coverage and mapping across abstraction levels. Saeed et al. [47] published an SLR that analyzed the state-of-the-art of experimental applications of search-based techniques (SBTs) for MBT. They presented a taxonomy to classify the various techniques. The authors searched for journal and conference studies from 2001 to 2013 in six sources: IEEE XPlore,[57] Springer,[58] Google Scholar,[59] ACM Digital Library,[60] ScienceDirect,[61] and Wiley Interscience.[62] Saeed et al. selected 72 studies, finding that most applications of SBTs for MBT consider functional and structural coverage. Additionally, the authors highlight research gaps in the techniques, including multi-objective SBTs, devising hybrid techniques, and applying constraint handling.

Bernardino et al. [8] presented an SMS to summarize MBT research. Primary studies were retrieved through automatic searches on five indexed databases (ACM Digital Library,[60] IEEE Xplore,[57] Elsevier ScienceDirect,[61] Springer SpringerLink,[58] and Elsevier Engineering Village[63]) and one search engine (Elsevier Scopus[64]). They selected 87 primary studies published from 2006 to 2016, which included conference papers, journal papers, books, and PhD dissertations. The authors classified these studies based on five factors: (1) whether they employed model representations or specifications, (2) the application domains, (3) the tools, (4) whether they exploited test modeling or test case generation, and (5) by the research groups. The SMS presented four main results. First, the representations varied widely, and were grouped as UML-based models (UML[65], SysML[66], and MARTE[67]), whether the models were formal or semi-formal models (Finite State Machines, Markov Chains, Petri Nets, and Simulink), and others. Second, they identified 70 tools, which they classified as academic, commercial, or open-source. Third, they found 20 application domains, including desktop applications, critical systems, health care, and web services. Fourth, they found seven activities related to MBT, with most of the studies focusing on test case generation, test modeling, and model transformation.

In another SMS, Gurbuz and Tekinerdogan [22] examined the state-of-the-art of MBT for software safety. Specifically, they identified the domains in which MBT has been applied and the contemporary research trends within MBT as applied to software safety. Additionally, Gurbuz and Tekinerdogan explored whether the current approaches have been empirically evaluated. The authors searched for primary studies in the following sources: ACM Digital Library,[60] IEEE Xplore,[57] ISI Web of

[57] http://ieeexplore.ieee.org/Xplore/home.jsp – accessed in June, 2023.
[58] http://link.springer.com/ – accessed in June, 2023.
[59] http://scholar.google.com/ – accessed in June, 2023.
[60] http://dl.acm.org/ – accessed in June, 2023.
[61] http://www.sciencedirect.com/ – accessed in June, 2023.
[62] http://onlinelibrary.wiley.com/ – accessed in June, 2023.
[63] http://www.engineeringvillage.com/ – accessed in June, 2023.
[64] http://www.scopus.com/home.uri – accessed in June, 2023.
[65] http://www.uml.org/ – accessed in June, 2023.
[66] http://sysml.org/ – accessed in June, 2023.
[67] https://www.omg.org/omgmarte/ – accessed in June, 2023.

Knowledge,[68] Elsevier ScienceDirect,[61] Elsevier Scopus,[52] Springer SpringerLink,[58] and Wiley Interscience.[62] They selected 36 of the 751 studies found during the search. According to their results, MBT has the potential to positively impact software safety testing. However, the field needs further advances to apply MBT for software safety testing.

Petry et al. [44] also conducted an SMS on MBT, but they investigated how MBT has been applied to software product lines (SPLs). Petry et al. answered RQs about approaches, artifacts, domains, evaluation, solution types, test case automation, traceability, and variability. After searching for primary studies in seven sources (ACM Digital Library, Google Scholar, IEEE Xplore, IET Digtial Library, Science Direct, Scopus, and Springer), the authors selected 44 primary studies. They found that black-box testing has been widely adopted, most studies described fully-automated applications of MBT to SPLs, and the most widely employed model to test SPLs is state machines. Additionally, the most recurring empirical evaluation strategies are case studies and experiments, which are often performed in industrial settings. Most studies did not address traceability or variability management. Petry et al. stated that variability management was briefly mentioned in most of the selected studies, but none of the selected studies go into detail about how variability is dealt with. Traceability was not mentioned in any of the selected studies. According to Petry et al. the main implication of overlooking traceability is that it is challenging to trace defects from MBT artifacts to the corresponding models. The authors also presented a roadmap that may guide researchers and practitioners interested in applying MBT to SPLs.

We found one paper on topic 4, testing non-testable systems. Patel and Hierons [42] gave results from an SMS that identified and compared automated testing techniques that attempted to detect functional faults. This is closely related to the oracle problem [34]. The authors ran an automatic search on six repositories, Brunel University Library,[69] Elservier ScienceDirect,[70] ACM Digital Library,[71] IEEE Xplore,[72] Google,[73] and Citeseerx,[74] including studies that are either peer- or non-peer-reviewed (technical reports, book chapters, and magazine papers), upon which they performed one round of backward snowballing. They also analyzed the publications of every author of the selected studies, and double-checked the completeness of the study selection with those authors. Their final set comprised 137 studies. Their main result was a comparison, in terms of efficiency and cost, of five umbrella testing techniques that address the oracle problem.

8. CONCLUDING REMARKS AND IMPLICATIONS FOR FUTURE RESEARCH

This article reports on an SLR that characterized how source code coverage can be computed from test sets generated through the application of MBT approaches. We analyzed and drew conclusions from 30 primary studies that we selected via a snowballing process. We identified some common characteristics and limitations, termed *issues* in what follows, that may impact on research and practice of MBT. Next we list each issue and discuss implications for future research related to them.

Issue: Automatic tools obscure details of how are transformed from model down to code.
Implications: In some studies, industrial or custom-tailored tools were employed to fully transform test sets from model to code. Then the test cases were automatically applied to code without manual intervention. These studies did not provide details about *how* model-level test cases are transformed to the code level. Without details of how abstract tests are transformed to concrete tests, testers are unable to predict how changing test cases at one level would affect the other, making it all but impossible to effectively update or evolve the test cases. To bridge this gap, future work should focus

[68] http://www.webofknowledge.com/ – accessed in June, 2023.
[69] http://www.brunel.ac.uk/life/library – accessed in June, 2023.
[70] http://www.sciencedirect.com/ – accessed in June, 2023.
[71] http://dl.acm.org/ – accessed in June, 2023.
[72] http://ieeexplore.ieee.org/Xplore/home.jsp – accessed in June, 2023.
[73] http://www.google.com/ – accessed in June, 2023.
[74] http://citeseerx.ist.psu.edu – accessed in June, 2023.

DOI: 10.1002/stvr

on reporting the inner workings of the techniques and strategies employed for the transformation of abstract tests into concrete test cases, enabling testers to make informed decisions regarding test case modifications and enhancements.

Issue: Model checking-based techniques often do not present their computational cost.
Implications: Some primary studies that applied model checking-based techniques for test case generation did not characterize the computational cost of exploring the state space of non-trivial models. This cost is often high, and sometimes prohibitively so. Even medium-sized models might lead to large state spaces, thus transforming high-level models to lower-level models (including code) requires a trade-off between exploring the entire search space and the approximation of examining only the most promising parts of the search space. We suggest that future studies need to quantify computational costs and quantify what degree of precision, in terms of test case quality, is sacrificed to reduce computational cost to address that trade-off.

Issue: The relationship between model coverage by abstract tests and code coverage by concrete tests has not been sufficiently studied.
Implications: We found that few studies addressed how coverage of models by abstract tests relates to the coverage of low-level representations of the models (including code) for concrete and almost-concrete tests. Most studies that addressed this topic applied structural model coverage criteria such as node or edge coverage. These rely on some sort of graph, such as finite-state machines at the model level and control-flow graphs at the code level. We found that to properly convey model coverage information to lower level representations, some extra transformations are needed at the model level by, for example, turning behavior into explicit predicates. Some studies found that coverage is lower at the code level when the code includes statements that were not explicitly modeled. We also found that only one study employed mutation to evaluate coverage, a very rich area for future research development.

Issue: Lack of traceability throughout the testing process
Implications: Although traceability from model to code has the potential to be an added value of MBT, many primary studies do not mention how their approaches track these links. Most of the examined primary studies emphasize specific parts of the MBT process without detailing how the proposed approaches help testers track coverage information at both model and code levels. Significantly more research is needed to develop this important type of traceability.

Issue: Portability among models is an afterthought
Implications: Many modeling languages, both formal and informal, are in use and more have been developed. Although they have similarities, they are sufficiently different to complicate the application of model-to-code transformation approaches developed for one model to others. It appears that, for most researchers, portability of models between MBT tools is an afterthought. This hampers the creation of tools that build upon infrastructure provided by existing tools. Even tools that utilize similar model representations tend to employ different subsets of the modeling notations. We conjecture that this may gradually improve as notations and tools achieve wider market success, and maybe as more robust commercial tools are developed, the number of languages will go down.

Issue: Incomplete reporting
Implications: We found that most primary studies did not completely report experimental and analytical details of their evaluation methodology. This was also reported in previous studies [24]. This lack challenges the assessment of the strength and suitability of MBT techniques for industrial adoption. Although there are a few industrial-strength MBT tools, our study provides evidence that it is still a challenge for both practitioners and researchers to evaluate MBT tools and techniques in real-world, industrial settings. According to our results, empirical evidence on mainstream use, including the transformation of model-level test cases into code-level test cases, is somewhat limited. Several

DOI: 10.1002/stvr

studies [12, 16, 27] described a contrived usage example, failing to provide empirical evidence supporting the effectiveness of the proposed approach. Therefore, we argue that technology transfer has been negatively affected by a lack of data to inform the evolution of MBT tools. It is imperative that future studies improve the transparency of reporting their experimental designs to support better comprehension of the methodology and reproducibility of results. We also hope that reviewers and editors will be more diligent about noting missing information in studies, and insist that authors correct the oversights in revision.

In our study, we found increasing adoption of MBT in industry, increasing application of model-to-code transformations, and a complementary increasing need to understand how test cases designed for models achieve coverage on the code. Although these studies document significant progress on this topic, these issues document significant gaps in our intellectual knowledge on the topic. We hope that practitioners can benefit from our study to better test their software and to better understand how well their software has been tested. We also hope that researchers can use this study as a reference to learn about the current state of knowledge and to identify future research directions, both theoretical and empirical.

Finally, as with any SLR and despite our best efforts over a few years of work, it is unlikely that we found all primary studies. Although we applied several search strategies, the limitations of research repositories (and our own abilities) mean that no search can be exhaustive. Thus, we hope this SLR will be further updated in the future.

ACKNOWLEDGEMENTS

Fabiano Ferrari was partly supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) - Brasil, grant #2016/21251-0; and CNPq - Brasil, grants #306310/2016-3 and #312086/2021-0. Sten Andler was partly supported by KKS (The Knowledge Foundation), by project 20130085, Testing of Critical System Characteristics (TOCSYC). Mehrdad Saadatmand was partly funded by the SmartDelta project (more information available at https://smartdelta.org/).

REFERENCES

[1] A. Abade, F. Ferrari, and D. Lucrédio. Testing M2T Transformations: A Systematic Literature Review. In *Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS)*, pages 177–187, Barcelona, Spain, 2015. SCITEPRESS Digital Library.

[2] D. Amalfitano, V. De Simone, A. R. Fasolino, and V. Riccio. Comparing Model Coverage and Code Coverage in Model Driven Testing: An Exploratory Study. In *Proceedings of the 6th International Workshop on Testing Techniques for Event BasED Software (TESTBEDS)*, pages 70–73, Lincoln, NE, USA, 2015. IEEE.

[3] D. Amalfitano, V. De Simone, R. R. Maietta, S. Scala, and A. R. Fasolino. Using Tool Integration for Improving Traceability Management Testing Processes: An Automotive Industrial Experience. *Software: Evolution and Process*, 31(6):1–20, 2019.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2nd edition, 2017. ISBN 978-1107172012.

[5] A. Aniculaesei, A. Vorwald, and A. Rausch. Using the SCADE Toolchain to Generate Requirements-Based Test Cases for an Adaptive Cruise Control System. In *Proceedings of the 16th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, pages 503–513, Munich, Germany, 2019. IEEE.

[6] C. Atkinson and T. Kuhne. Model-driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[7] A. Baresel, M. Conrad, S. Sadeghipour, and J. Wegener. The Interplay between Model Coverage and Code Coverage. In *Proceedings of the 10th EuroSTAR Software Testing Conference*, pages 1–14, Amsterdam, The Netherlands, 2003. Qualtech Group.

[8] M. Bernardino, E. M. Rodrigues, A. F. Zorzo, and L. Marchezan. Systematic Mapping Study on MBT: Tools and Models. *IET Software*, 11(4):141–155, 2017.

[9] L. Briand, S. Nejati, M. Sabetzadeh, and D. Bianculli. Testing the Untestable: Model Testing of Complex Software-intensive Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE) - Visions of 2025 and Beyond Track*, pages 789–792, Austin, TX, USA, 2016. ACM.

[10] J.-L. Camus, C. Haudebourg, M. Schlickling, and J. Barrho. Data Flow Model Coverage Analysis: Principles and Practice. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS)*, pages 1–10, Toulouse, France, 2016. Centre pour la Communication Scientifique Directe.

[11] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[12] M. Conrad. Testing-based Translation Validation of Generated Code in the Context of IEC 61508. *Formal Methods in System Design*, 35(3):389–401, 2009.

[13] M. Conrad, S. Sadeghipour, and H.-W. Wiesbrock. Automatic Evaluation of ECU Software Tests. *SAE Transactions*, 114:583–592, 2005.

[14] I. Drave, S. Hillemacher, T. Greifenberg, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, and A. Wortmann. SMArDT Modeling for Automotive Software Testing. *Software: Practice and Experience*, 49 (2):301–328, 2019.

[15] I. Drave, S. Hillemacher, T. Greifenberg, B. Rumpe, A. Wortmann, M. Markthaler, and S. Kriebel. Model-Based Testing of Software-Based System Functions. In *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 146–153. IEEE, 2018.

[16] U. Durak, D. Müller, F. Möcke, and C. B. Koch. Modeling and Simulation based Development of an Enhanced Ground Proximity Warning System for Multicore Targets. In *Proceedings of the 2018 International Symposium on Model-driven Approaches for Simulation Engineering (Mod4Sim)*, pages 1–12, Baltimore, MD, USA, 2018. ACM.

[17] F. Elberzhager, A. Rosbach, and T. Bauer. Analysis and Testing of Matlab Simulink Models: A Systematic Mapping Study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation (JAMAICA)*, pages 29–34, Lugano, Switzerland, 2013. ACM.

[18] A. Eriksson and B. Lindström. UML Associations: Reducing the Gap in Test Coverage Between Model and Code. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 589–599, Rome, Italy, 2016. IEEE.

[19] A. Eriksson, B. Lindström, S. Andler, and J. Offutt. Model Transformation Impact on Test Artifacts: An Empirical Study. In *Proceedings of the 9th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, pages 5–10, Innsbruck, Austria, 2012. ACM.

[20] A. Eriksson, B. Lindström, and J. Offutt. Transformation Rules for Platform Independent Testing: An Empirical Study. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)*, pages 202–211, Luxembourg City, Luxembourg, 2013. IEEE.

[21] P. Fraternali and M. Tisi. Multi-level Tests for Model Driven Web Applications. In *Proceedings of the 10th International Conference on Web Engineering (ICWE)*, pages 158–172, Vienna, Austria, 2010. Springer.

[22] H. G. Gurbuz and B. Tekinerdogan. Model-based Testing for Software Safety: A Systematic Mapping Study. *Software Quality Journal*, 2018. ISSN 1573-1367.

[23] A. Kalaee and V. Rafe. Model-based Test Suite Generation for Graph Transformation System Using Model Simulation and Search-based Techniques. *Information and Software Technology*, 108:1–29, 2019.

[24] M. U. Khan, S. Iftikhar, M. Z. Iqbal, and S. Sherin. Empirical Studies Omit Reporting Necessary Details: A Systematic Literature Review of Reporting Quality in Model Based Testing. *Computer Standards & Interfaces*, 55:156–170, 2018.

[25] R. Kirner. Towards Preserving Model Coverage and Structural Code Coverage. *EURASIP Journal on Embedded Systems*, 2009:1–16, 2009.

[26] B. A. Kitchenham, D. Budgen, and P. Brereton. *Evidence-Based Software Engineering and Systematic Reviews*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, 2015.

[27] C. B. Koch, U. Durak, and D. Müller. Simulation-based Verification for Parallelization of Model-based Applications. In *Proceedings of the 50th Computer Simulation Conference (SummerSim)*, pages 1–10, Bordeaux, France, 2018. ACM.

[28] B. P. Lamancha, P. Reales, M. Polo, and D. Caivano. Model-driven Testing - Transformations from Test Models to Test Code. In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 121–130, Beijing, China, 2011. SCITEPRESS Digital Library.

[29] G. Li, R. Zhou, R. Li, W. He, G. Lv, and T. J. Koo. A Case Study on SDF-based Code Generation for ECU Software Development. In *Proceedings of the 3rd International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS)*, pages 211–217, Munich, Germany, 2011. IEEE.

[30] N. Li, A. Escalona, and T. Kamal. Skyfire: Model-Based Testing with Cucumber. In *Proceedings of the 9th International Conference on Software Testing, Verification and Validation (ICST) - Testing Tool Papers*, pages 393–400, Chicago, IL, USA, 2016. IEEE.

[31] N. Li and J. Offutt. A Test Automation Language Framework for Behavioral Models. In *Proceedings of the 11th Workshop on Advances in Model Based Testing (A-MOST)*, pages 1–10, Graz, Austria, 2015. IEEE.

[32] N. Li and J. Offutt. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2016.

[33] W. Li, F. Le Gall, and N. Spaseski. A Survey on Model-Based Testing Tools for Test Case Generation. In *Proceedings of the 4th Tools & Methods of Program Analysis International Conference (TMPA)*, pages 77–89, Moscow, Russia, 2017. Springer.

[34] H. Liu, F. Kuo, D. Towey, and T. Y. Chen. How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

DOI: 10.1002/stvr

[35] M. Markthaler, S. Kriebel, K. S. Salman, T. Greifenberg, S. Hillemacher, B. Rumpe, C. Schulze, A. Wortmann, P. Orth, and J. Richenhagen. Improving Model-Based Testing in Automotive Software Engineering. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 172–180. ACM, 2018.

[36] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull. Search-based Automated Testing of Continuous Controllers: Framework, Tool Support, and Case Studies. *Information and Software Technology*, 57:705–722, 2015.

[37] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.

[38] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. C. Shashidhar, and S. Ramesh. Automatic Test Case Generation from Simulink/Stateflow Models using Model Checking. *Software Testing, Verification and Reliability*, 24(2):155–180, 2014.

[39] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[40] Object Management Group. Model Driven Architecture (MDA) – MDA Guide rev. 2.0. Technical Report ormsc/2014-06-01, Object Management Group, 2014. URL `https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf`.

[41] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. Springer-Verlag.

[42] K. Patel and R. M. Hierons. A Mapping Study on Testing Non-Testable Systems. *Software Quality Journal*, 26(4):1373–1413, 2018.

[43] T. K. Paul and M. F. Lau. A Systematic Literature Review on Modified Condition and Decision Coverage. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*, pages 1301–1308, Gyeongju, Republic of Korea, 2014. ACM.

[44] K. L. Petry, E. Oliveira Jr, and A. F. Zorzo. Model-Based Testing of Software Product Lines: Mapping Study and Research Roadmap. *Journal of Systems and Software*, 167:110608, 2020.

[45] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One Evaluation of Model-based Testing and Its Automation. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 392–401, St. Louis, MO, USA, 2005. ACM.

[46] RTCA. Software Considerations in Airborne Systems and Equipment Certification DO-178C. Technical report, RTCA, Inc., December 2011.

[47] A. Saeed, S. H. A. Hamid, and M. B. Mustafa. The Experimental Applications of Search-based Techniques for Model-based Testing: Taxonomy and Systematic Literature Review. *Applied Software Computing*, 49:1094–1117, 2016.

[48] B. Selic. The Pragmatics of Model-driven Development. *IEEE Software*, 20(5):19–25, 2003.

[49] H. Shokry and M. Hinchey. Model-Based Verification of Embedded Software. *IEEE Computer*, 42(2):53–59, 2009.

[50] I. Stürmer, M. Conrad, H. Doerr, and P. Pepper. Systematic Testing of Model-Based Code Generators. *IEEE Transactions on Software Engineering*, 33(9):622–634, 2007.

[51] I. Stürmer, M. Conrad, H. Dörr, and P. Pepper. Systematic Testing of Model-Based Code Generators. *IEEE Transactions on Software Engineering*, 33(9):662–634, 2007.

[52] I. Stürmer, D. Weinberg, and M. Conrad. Overview of Existing Safeguarding Techniques for Automatically Generated Code. In *Proceedings of the 2nd International Workshop on Software Engineering for Automotive Systems*, pages 1–6, St. Louis, MO, USA, 2005. ACM.

[53] T. Tekcan, V. Zlokolica, V. Pekovic, N. Teslic, and M. Gündüzalp. User-driven Automatic Test-case Generation for DTV/STB Reliable Functional Verification. *IEEE Transactions on Consumer Electronics*, 58(2):587–595, 2012.

[54] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.

[55] J. Vanhecke, X. Devroey, and G. Perrouin. AbsCon: A Test Concretizer for Model-Based Testing. In *Proceedings of the 15 Workshop on Advances in Model Based Testing (A-MOST)*, pages 15–22, Xi'an, China, 2019. IEEE.

[56] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Proceedings of the 2008 Formal Methods and Testing Workshop (FORTEST)*, pages 39–76. Springer, 2008.

[57] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.

[58] C. Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10, London, UK, 2014. ACM.

[59] X. Zhou, Y. Jin, H. Zhang, S. Li, and X. Huang. A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 153–160, 2016.

DOI: 10.1002/stvr