**Noname manuscript No.**
(will be inserted by the editor)

# Model Execution Tracing: A Systematic Mapping Study

**Fazilat Hojaji · Tanja Mayerhofer · Bahman Zamani · Abdelwahab Hamou-Lhadj · Erwan Bousse**

Received: date / Revised version: date

**Abstract** Model Driven Engineering is a development paradigm that uses models instead of code as primary development artifacts. In this paper, we focus on *executable models*, which are used to abstract the behavior of systems for the purpose of verifying and validating (V&V) a system's properties. Model execution tracing (i.e., obtaining and analyzing traces of model executions) is an important enabler for many V&V techniques including testing, model checking, and system comprehension. This may explain the increase in the number of proposed approaches on tracing model executions in the last years. Despite the increased attention, there is currently no clear understanding of the state of the art in this research field, making it difficult to identify research gaps and opportunities. The goal of this paper is to survey and classify existing work on model execution tracing, and identify promising future research directions. To achieve this, we conducted a systematic mapping study where we examined 64 primary studies out of 645 found publications. We found that the majority of model execution tracing approaches has been developed for the purpose of testing and dynamic analysis. Furthermore, most approaches target specific modeling languages and rely on custom trace representation formats, hindering the synergy among tools and exchange of data. This study also revealed that most existing approaches were not validated empirically, raising doubts as to their effectiveness in practice. Our results suggest that future research should focus on developing a common trace exchange format for traces, designing scalable trace representations, as well as conducting empirical studies to assess the effectiveness of proposed approaches.

**Keywords** Model Driven Engineering, Executable Models, Model Execution Tracing, Dynamic Analysis of Model Driven Systems, Systematic Mapping Study

Fazilat Hojaji
E-mail: f.hojaji@eng.ui.ac.ir

Tanja Mayerhofer
E-mail: mayerhofer@big.tuwien.ac.at
https://big.tuwien.ac.at/people/tmayerhofer/

Bahman Zamani
E-mail: zamani@eng.ui.ac.ir

Abdelwahab Hamou-Lhadj
E-mail: wahab.hamou-lhadj@concordia.ca

Erwan Bousse
E-mail: bousse@big.tuwien.ac.at
https://big.tuwien.ac.at/people/ebousse/

## 1 Introduction

Model Driven Engineering (MDE) is a software development paradigm that aims to decrease the complexity of software systems development, by raising the level of abstraction through the use of models and well-defined modeling languages [98]. Two main types of modeling languages are used: General-Purpose Modeling Languages (GPMLs), such as UML, that can be used for modeling systems regardless of the domain, and Domain-Specific Modeling Languages (DSMLs) that are each designed specifically for a given domain [15]. Models are very useful for analyzing a system's quality properties to explore design alternatives or identify potential improvements, which requires checking both functional and non-functional properties, by examining the structural and behavioral aspects of a system. In the case of behavioral aspects, the focus of this paper, *dynamic* V&V techniques are typically used. This family of techniques necessitate the ability to *execute* models. To this

end, many efforts have been made to support the execution of models, such as methods that ease the development of executable DSMLs (xDSMLs) [11, 21, 56, 84, 102], or to support the execution of UML models [19]. This endeavor includes both facilitating the definition of the execution semantics of modeling languages, and the development of dynamic V&V methods that are tailored to these executable languages.

Yet, a very important prerequisite for most dynamic V&V tools is the ability to *trace* the execution of executable models. For example, model checking techniques as proposed by Meyers et al. [86], Jhala et al. [65], Hilken et al. [60] and Hegedus et al. [57] check whether execution traces satisfy a system's temporal properties and rely on execution traces for representing counter examples. Omniscient debugging as, for instance, proposed by Barr et al. [8] and Bousse et al. [10, 12] utilizes execution traces to go back in the execution and revisit previous states. Semantic model differencing techniques, such as the ones proposed by Langer et al. [73] and Maoz et al. [80], compare execution traces of two models to identify semantic differences between them.

In the last decade, there has been a noticeable increase in the number of papers on tracing techniques for model executions (a detailed discussion of this trend will follow in Section 3.2.1.2). However, the direction in which the field is heading is not clear at the moment, as are the advantages and limitations of existing techniques. We believe this is due to the lack of a survey of the state of the art and of a classification of existing work in the area of model execution tracing. While there are a few surveys available in literature that cover related topics, such as techniques and tools for the execution of UML models surveyed by Ciccozzi et al. [19] and approaches for using models at runtime investigated by Szvetits and Zdun [101], none of these studies focuses on model execution tracing. We discuss the related surveys and their relationship to our work in more detail in Section 7.

To overcome the limitations of existing studies, we conducted a systematic mapping study of existing model execution tracing techniques, following the guidelines presented by Kitchenham and Charters [16, 69], and Petersen et al. [93]. We examined 64 research studies from an initial set of 645, and classified them based on the following facets: (1) the types of models that are traced, (2) the supported execution semantics definition techniques, (3) the traced data, (4) the purpose of model execution tracing, (5) the data extraction technique, (6) the trace representation format, (7) the trace representation method, (8) the language specificity of the trace structure, (9) the data carrier format used for storing traces, and (10) the maturity level of the supporting model execution tracing tools, if provided. Using this classification, we evaluated the state of the art of model execution tracing techniques, and identified promising future research directions in this area.

The main contributions of this study are: (i) a framework for classifying and comparing model execution tracing techniques, (ii) a systematic review of the current state of the art in model execution tracing, and (iii) an exploration of open research challenges in model execution tracing.

This study targets researchers and practitioners who want to gain insight into existing model execution tracing techniques, and/or contribute further to the development of this field of study.

This study focuses on model execution tracing approaches. Other tracing techniques employed in MDE, such as maintaining traceability links between source and target models of model transformations and tracing the execution of model transformations, are out of the scope of this paper. We also exclude tracing techniques for programs written in general-purpose programming languages (e.g., [53], because the requirements for tracing models differ from those that apply to tracing programs. In particular, xDSMLs usually provide concepts at a higher level of abstraction than those in programming languages. This requires trace formats that are usually quite different from trace formats for general-purpose programming languages. For instance, while for the execution of programs written in general-purpose programming languages, traces commonly capture information about threads and function calls, those concepts are not generally applicable to modeling languages. Consider, for instance, the UML state machine language where the primary concepts are states and transitions, but neither threads nor function calls are part of the language. Also, xDSMLs and general-purpose programming languages reside in different technological spaces.

The remainder of this paper is structured as follows. In Section 2, we present background information related to model execution tracing. In Section 3, we describe the research method used for conducting the mapping study and the classification scheme applied for the research. Section 4 reports the main findings. In Section 5, we present identified open challenges and present future research directions. In Section 6, we evaluate our approach and findings by discussing limitations and threats to validity. Section 7 outlines the related work. Section 8 summarizes the results and concludes the paper.

## 2 Background

In this section, we define the concepts of *executable modeling languages*, *executable models*, and *model execution traces*.

### 2.1 Executable Modeling Languages

An executable modeling languages is a specific type of modeling languages that supports the execution of models, and thus enables the application of dynamic V&V techniques.We want to note that modeling languages can be considered as programming languages that raise the level of abstraction beyond code to express information on systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure [66]. The main difference between code-based programming languages and modeling languages is the higher level of abstraction.

To support the execution of models, an executable modeling language must provide *execution semantics*. There are three different approaches for defining the execution semantics: the denotational semantics approach, the translational semantics approach, and the operational semantics approach [17]. The denotational semantics, which is also known as mathematical semantics, describes the semantics of a language by defining algebraic/mathematical terms [100]. In the *translational* approach, the model is translated into another executable language for execution. This can be done through exogenous model transformations or through code generation if the target language possesses a grammar. In the *operational* approach, the execution behavior of models conforming to an executable modeling language is defined by an interpreter (a virtual machine). The interpreter first constructs a representation of the execution state of a model and then modifies this representation by executing the model through a series of transitions from one execution state to the next one. To build the initial representation of a model's execution state, an exogenous model transformation is used. The transitions between execution states are realized as in-place model transformations.

In summary, we define the terms *executable modeling language* and *executable model* as follows. These definitions are based on the ones proposed by Bousse et al. [14].

**Definition 1** An *executable modeling language* is a modeling language with execution semantics that can be used to define and execute models. Execution semantics specifies the execution behavior of models.

**Definition 2** An *executable model* is a model conforming to an executable modeling language. It defines an aspect of the behavior of a system in sufficient detail to be executed.

In MDE, many different executable modeling languages have been developed and used to express the behavior of systems. Examples include Petri nets [94], fUML [91], BPMN [90], live sequence charts [28], and story diagrams [39].

To illustrate the concept how a model can be executed, we use the Petri net model in Figure 1. The modeling concepts provided by the Petri net language are *places* and *transitions* where places and transitions can be connected with each other. Our example model consists of four places *p1* to *p4* and two transitions *t1* and *t2*. We define the execution semantics of the Petri net language following the operational semantics approach. In particular, we define the execution state of a Petri net by a distribution of *tokens* among places (e.g., in the initial state, the place *p1* holds one token) and two transformations *run* and *fire* that change the execution state. The rule *fire* fires transition, i.e., removes one token of each input place of the transition and adds one token to each output place of a transition. The rule *run* calls the rule *fire* for all enabled transitions, i.e., transitions whose input places contain at least one token. In our example, first the transition *t1* is fired, because its only input place *p1* holds one token. Thereafter, there is one token at each of the places *p2* and *p3*, which are the input places of transition *t2*. Thus, the transition *t2* can be fired in the next execution step completing the execution of the Petri net with one token at place *p4*. Metrics that can be captured about the execution of a Petri net are manifold including the reached markings (i.e., token distributions), the number of transition firings in the complete model or per transition, and the number of tokens flowing through places. More complex analyses may be performed on the basis of this data, such as loop detection.

### 2.2 Model Execution Traces

There exist many definitions of the term *tracing* in the literature, such as the use of logging mechanisms to record information about a program's execution [70] or a protocol to capture the behavior of a running program [85]. For this work, we define model execution traces as follows:

**Definition 3** A *model execution trace* captures relevant information about the execution of an executable model. This information may include execution states,

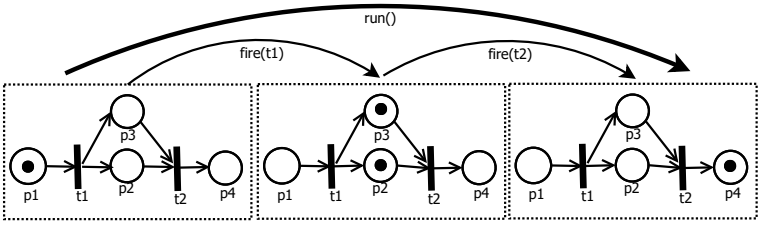


**Fig. 1** Example of a Petri net model execution

events that occurred during the execution, execution state changes, processed inputs, and produced outputs.

Tracing a model execution is needed to support various dynamic V&V activities at the model level. Performing dynamic V&V tasks at the early stages of model-driven development processes is desirable to improve quality and prevent rework at later stages. Typical examples of dynamic V&V include debugging, testing, model checking, trace analysis for program comprehension, and many other dynamic analysis tasks (see [51] for examples). These techniques rely typically on execution traces as a representation of the behavior of the system.

The content of traces depends on the degree of abstraction required by the desired dynamic V&V technique and the runtime concepts provided by the languages as well. Alawneh and Hamou-Lhadj [2] have categorized traces of code-centric systems into statement-level traces, routine call traces, inter-process traces, and system call level traces. In the case of executable models, execution traces may contain different kinds of information depending on the considered executable modeling language. Furthermore, instead of tracing threads and function call stacks, which are general concepts of programming languages, concepts like transitions, states, and actions are often traced in model execution tracing.

The information to be traced may be extracted from a model execution in different ways. For instance, for executable modeling languages with operational semantics, the interpreter of the executable modeling language may provide features to record execution traces (e.g., applied by Combemale et al. [21]), and for executable modeling languages with translational semantics, additional elements may be inserted in the target model/code that are responsible for producing traces (e.g., applied by Want et al. [103]).

An execution trace must conform to a trace format, which defines the concepts required for representing execution traces. A trace format may be defined using different techniques, such as XML schema (e.g., applied by Kemper and Tepper [67, 68]), metamodels (e.g., applied by Hegedus et al. [56, 57]), and grammars (e.g., applied by Fernández-Fernández and Simons [37, 38]). Furthermore, it may be specific to the specific aspects of an executable modeling language (e.g., UML state machines) or generic and applicable to any executable modeling language. The generated execution traces can be stored on a disk using different encoding techniques, e.g. they may be recorded in databases (e.g., applied by Domínguez [35]), as simple text files (e.g., applied by Crane and Dingel [26]) or as XML documents (e.g., applied by Combemale et al. [23]).

Thus, there are many different dimensions (trace purpose, trace content, trace data extraction technique, trace format, etc.) that can be used to classify and compare existing model execution tracing solutions. Our classification schema developed as part of this systematic mapping study is introduced in Section 3.2.2.

## 3 Research Method

To conduct this systematic mapping study, we followed the guidelines presented by Kitchenham and Charters [16, 69], and Petersen et al. [93]. The goal of this study is to answer the following research questions.

**Q1** (Type of Models): Which executable modeling languages are targeted by model execution tracing approaches?
**Q2** (Semantics Definition Technique): Which techniques are used to define the execution semantics of executable modeling languages targeted by model execution tracing approaches?
**Q3** (Trace Data): What kind of data is captured in model execution traces?
**Q4** (Purpose): For what purposes is model execution tracing used?
**Q5** (Data Extraction Technique): Which techniques are used for extracting runtime information from model executions in order to construct execution traces?
**Q6** (Trace Representation Format): Which data representation format is used for defining the trace data structure?
**Q7** (Trace Representation Method): How is the trace data structure defined?

**Q8** (Language Specificity of Trace Structure): Is the data structure specific to the considered executable modeling language, specific to a particular kind of executable modeling language, or considered independent of any executable modeling language (i.e., generic)?
**Q9** (Data carrier format): Which data carrier format is used for storing traces?
**Q10** (Maturity Level): How mature are available tools for model execution tracing?

The study protocol includes three phases, namely planning, conducting, and reporting. In the following, we discuss the planning and conducting of the study. The study results are reported in Section 4.

### 3.1 Review Planning

The first phase of our survey process is planning, which consists of defining the search strategy and review process. In the following, we explain the search strategy. The review process is outlined as part of the discussion of the review conduction in Section 3.2.

We used the following online libraries to find research studies related to model execution tracing. With this list of online libraries, we aimed at achieving a comprehensive coverage of publication venues from all major publishers in the field of software engineering.

- ACM Digital Library (http://dl.acm.org)
- IEEE Xplore (http://ieeexplore.ieee.org)
- ScienceDirect (http://www.sciencedirect.com)
- Springer Link (http://www.springer.com)
- Scopus (http://www.scopus.com)

The terms we used to select relevant research studies are as follows. Each term includes several keywords meaning that at least one of the keywords has to be present in a paper.

A = *model tracing, model-based trace, execution trace, tracing, trace*
B = *MDE, model-driven, model-level, model-based*
C = *meta-model, metamodel, modeling language*
D = *model execution, model verification, dynamic analysis, executable model, xDSML*

The overall search string can be combined in the following way:

$$\textbf{Search String} = (A \wedge (B \vee C \vee D)) \quad (1)$$

The rationale behind using this search string is to identify the largest number of research studies related to model execution tracing. We performed an advanced search in the aforementioned online research databases and search engines using this search string.

Furthermore, we defined two types of inclusion and exclusion criteria to decide whether a publication found in the search should be included in the study or excluded. The first type is a set of format-related criteria, such as publication language and publication type. These criteria have been selected based on the guidelines suggested by Adams et al. [1] and Petersen et al. [93]. The second type is a set of content-related criteria. The selection criteria used in this study are given below. The publications have been selected on the basis of these criteria through a manual analysis of their titles, abstracts, and keywords. When in doubt, also the conclusions of a publication have been considered.

**Inclusion Criteria**:

1. Publications in peer-reviewed journals, conferences, and workshops
2. Publications that address problems in the field of model execution tracing
3. Publications dealing with recording execution traces for models or designing execution trace formats for executable modeling languages

**Exclusion Criteria**:

1. Books, web sites, technical reports, dissertations, pamphlets, and whitepapers (based on the guidelines suggested by Adams et al. [1] and Petersen et al. [93]).
2. Summary, survey, or review publications
3. Non peer-reviewed publications
4. Publications not written in English
5. Publications after February 2018, the time the final search for primary studies was conducted
6. Publications on traceability in model transformation
7. Publications that do not consider executable models
8. Publications not focusing on MDE (e.g., execution tracing in code-centric approaches)

### 3.2 Review Conduction

The second phase of our study process, review conduction, consists of three main activities: article selection, data extraction, and article classification, which are elaborated in the following.

#### *3.2.1 Article Selection*

The article selection comprised a pilot study, the actual selection of primary studies, and the assessment of the quality of the selected primary studies.

#### 3.2.1.1 Pilot Study

Before the actual selection of articles, we performed a pilot study as suggested by Kitchenham and Charters [16, 69] and Petersen et al. [93] to confirm the reliability of our selection criteria.

In this pilot study, a set of ten articles was preselected from different sources and publishers. This list was defined based on the bibliography of one of our earlier papers [62]. This selection included seven articles that should be included in the study as primary studies and three articles that should be excluded. The selection was done by Fazilat Hojaji and Bahman Zamani.

The selected articles were then given to Abdelwahab Hamou-Lhadj, a domain expert who was not involved in the planning phase of the study process. Therefore, he was not biased by the search process. He was asked to decide based on the defined inclusion and exclusion criteria which of the selected articles should be considered as primary studies and which ones should be excluded from the mapping study. The results were then cross-checked against the initial classification of the preselected articles in primary studies and non primary studies.

**Pilot study results**: The first execution of the pilot study failed. Out of the ten articles, only six were correctly classified. In particular, five articles were correctly identified as primary studies and one was correctly identified as non primary study. Based on these results, the selection criteria were refined and the pilot study re-executed. After the second execution, the results were acceptable: Seven articles were correctly identified as primary studies and one was correctly identified as non primary study. The two articles that were assessed differently focused on dynamic analysis in code-centric approaches, which is beyond the scope of this mapping study. This led us to add an extra exclusion criterion excluding tracing approaches not focusing on MDE (exclusion criterion 8).

#### 3.2.1.2 Primary Studies Selection

This step comprised the search for relevant publications using the search string introduced in Section 3.1, the elimination of duplicate publications found in multiple online libraries, and the filtering of the publications by applying the aforementioned selection criteria.

Figure 2 shows the selection process of the primary studies along with the obtained results of the tasks. The initial search process returned 942 results including 297 duplicates. For the studies that were identified in more than one online library, we considered their original publisher. In order to assess the relevance of the found studies to our topic, we reviewed their titles, abstracts, keywords, introduction sections, and if needed conclusion sections. In this step, 506 studies were rejected, while 139 moved on to the next step. Next, we applied the inclusion and exclusion criteria. The result was a set of 71 studies. In the next step, a more detailed filtering was applied by inspecting the entire content of the remaining 71 studies. Most of the studies eliminated in this step were related to traceability in model transformation, and also dynamic analysis in code-centric approaches. The filtering yielded 53 remaining studies. In order to minimize the risk of missing any relevant studies, we performed a snowballing step by checking the references of the remaining 53 studies and identified 11 more studies fulfilling the inclusion criteria. Hence, these studies were added to the primary studies. The final set of primary studies investigated in this mapping study consists of 64 studies.

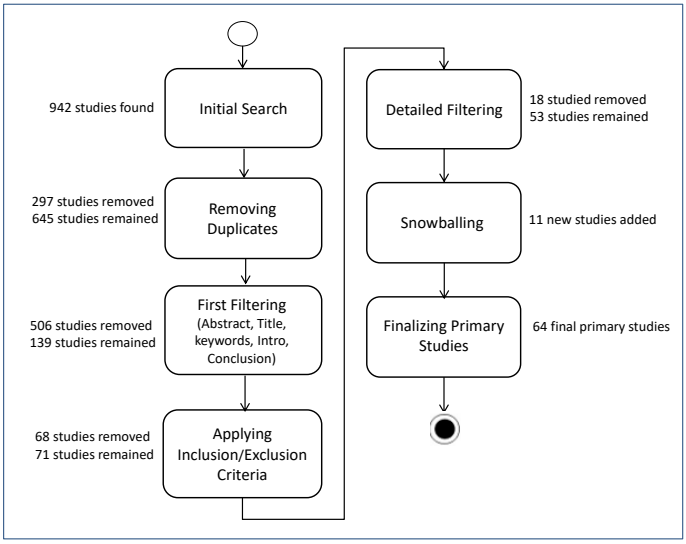


**Fig. 2** Primary studies selection process

**Publication trends**: Figure 3 shows for each used online library the total number of studies that were retrieved using the defined search string (initial studies), the number of studies remaining after the removal of duplicates and first filtering (potential studies), and the number of studies finally included in the mapping study after applying inclusion and exclusion criteria and detailed filtering (primary studies). Figure 4 shows the publication years of the primary studies. As can be seen in this figure, first publications on the topic of model execution tracing appeared around the year 2000, but only in 2007, the topic gained more interest by the scientific community leading to an increase in publications on this topic per year with the highest number of publications in the year 2014. Finally, Figure 5 presents the distribution of the selected primary studies based on the publication type. We only included studies published

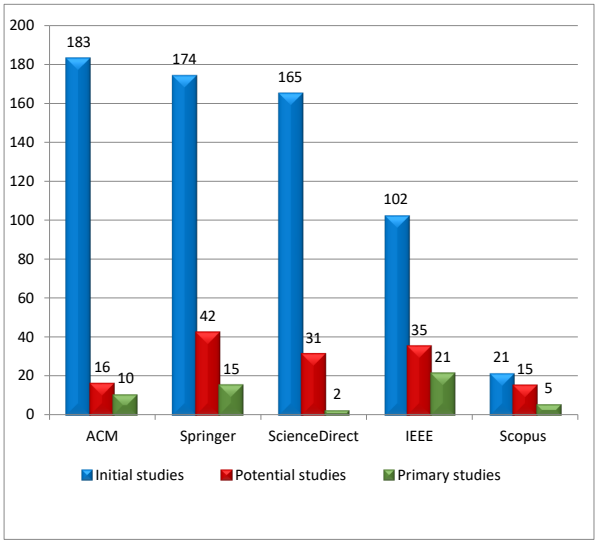


**Fig. 3** Studies retrieved through online libraries

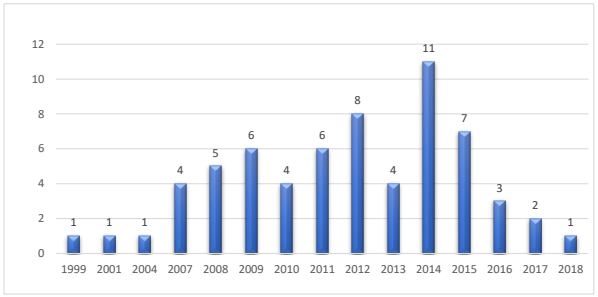


**Fig. 4** Primary studies per year

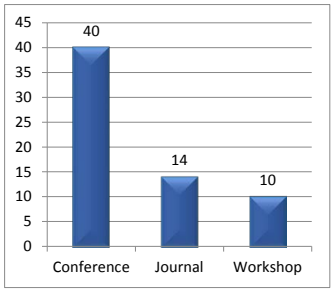


**Fig. 5** Primary studies per publication type

in peer-reviewed workshops, conferences, and journals. Please note that we considered symposia and congresses also as conferences. From the 40 primary studies shown in Figure 5 as conference papers, 4 have been presented at symposia and 1 at a congress. The figure shows that studies in the field of model execution tracing have been mostly published in conference proceedings.

#### 3.2.1.3 Quality Assessment

We also evaluated the quality of the selected primary studies as suggested by Kitchenham and Charters [16, 69] and Petersen et al. [93] to make sure that they are

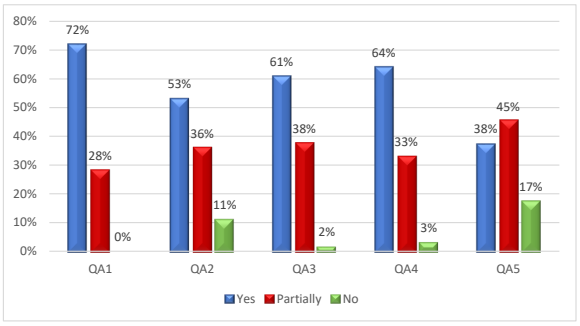


**Fig. 6** Results of quality assessment of primary studies

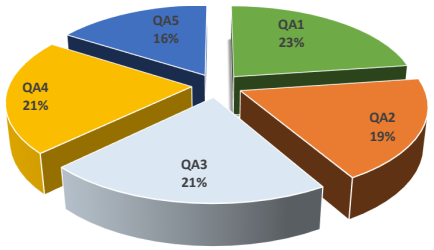


**Fig. 7** Total score for quality assessment questions

of sufficient quality to be included in a systematic mapping study. For this, we developed a checklist containing five quality assessment questions as presented in Table 1. The questions are based on the suggestions given in [16, 69] and [93], as well as the questions used in a study by Santiago et al. [96]. The questions have been answered for all primary studies by Fazilat Hojaji and Bahman Zamani. Thereby, the score values were 'Yes' = 1, 'Partly' = 0.5 or 'No' = 0.

Figure 6 shows the percentages of primary studies assigned 'Yes', 'Partly', or 'No' on the five quality assessment questions. It shows that for each of the five questions, most of the studies (83%-100%) score 'Yes' or 'Partly', which confirms that the selected primary studies are of sufficient quality to be included in this mapping study. We also calculated ("% total score") for each quality assessment question, which shows the percentage of scores obtained by all the primary studies assigned for a given quality assessment question over sum of the scores obtained by all primary studies for all QA1 to QA5. The arithmetic mean of the scores is 3.77 and the standard deviation 0.95. Figure. 7 shows a pie chart depicting the distribution of scores for the assessment questions. It illustrates that QA1 obtained

**Table 1** Quality assessment questionnaire

| ID | Topic | Question |
|---|---|---|
| QA1 | Objective | Did the study clearly define the research objectives? |
| QA2 | Related work | Did the study provide a review of previous work? |
| QA3 | Research methodology | Was the research methodology clearly established? |
| QA4 | Validity and reliability | Did the study include a discussion on the validity and reliability of the procedure used? |
| QA5 | Future work | Did the study point out potential further research? |

the highest score (%23) over the total score, while QA5 has the least (%16).

*3.2.2 Data Extraction and Classification Scheme*

In this section, we describe how we classified the selected primary studies. We developed a classification scheme by inspecting the content of all 64 selected primary studies with the goal of addressing our research questions. In particular, for each research question, we assigned keywords to the primary studies, which provide answers to the research question. For example, we assigned the keyword "interactive model-level debugging" for research question Q4 if the purpose of the model execution tracing approach presented in a primary study was to realize this type of dynamic V&V technique. Subsequently to this key-wording phase, the keywords assigned for each research question were clustered and for each cluster, a keyword encompassing all clustered keywords was assigned. The result was a set of attributes for each research question that was used to classify the primary studies for this research question. The attribute sets are described in the following.

#### 3.2.2.1 Types of Models (Q1)

This attribute set is used to characterize model execution tracing approaches concerning supported executable modeling languages. For this, we defined the following attributes.

- *Any*: refers to approaches that can be applied to any executable modeling language, i.e., approaches that can be used to trace the execution of models conforming to any executable modeling language.
- *UML models*: refers to approaches specifically designed to trace the execution of models conforming to UML or a subset of UML.
- *Workflow models*: refers to approaches specifically designed to trace the execution of workflow models that define the flow of work in processes. Workflow models can be expressed in executable modeling languages like Petri nets and BPMN.
- *Other*: refers to approaches that are designed to trace the execution of models conforming to particular executable modeling languages or kinds of executable modeling languages other than UML and workflow modeling languages. Note that approaches in this category are applicable to a restricted set of executable modeling languages, while approaches in the category *Any* can be applied to any executable modeling language.

#### 3.2.2.2 Semantics Definition Techniques (Q2)

This attribute set refers to the way execution semantics are defined for the executable modeling languages supported by a model execution tracing approach. As introduced in Section 2.1, we distinguish denotational, translational and operational semantics. Hence, the attributes are defined as follows.

- *Denotational*: refers to approaches applicable to executable modeling languages whose execution semantics are defined in algebraic/mathematical terms.
- *Translational*: refers to approaches applicable to executable modeling languages whose execution semantics are defined in a translational way.
- *Operational*: refers to approaches applicable to executable modeling languages whose execution semantics are defined in an operational way.
- *Unknown*: refers to approaches where no information about the kind of supported execution semantics is provided in the associated primary studies.

#### 3.2.2.3 Trace Data (Q3)

With this attribute set, we characterize model execution tracing approaches concerning the data recorded in execution traces. In particular, we used the following attributes.

- *Event*: refers to approaches that trace events occurring during the execution of a model.
- *State*: refers to approaches that trace information about the evolution of the execution state of a model.
- *Parameter*: refers to approaches that trace inputs processed by the execution of a model or outputs produced by the execution of a model.

#### 3.2.2.4 Purpose (Q4)

This attribute set is concerned with the purpose of the investigated model execution tracing approaches, in particular, the purpose of execution traces produced by the individual approaches. We determined the purposes from the primary studies by considering the applications of execution traces mentioned or explicitly presented by the authors as part of their contribution. This way, we identified the following purposes of model execution traces that serve as attributes for this research question.

- *Debugging*: refers to techniques to interactively control and observe the execution of a model in order to find and correct defects. Model execution traces can be utilized in different ways for the purpose of debugging executable models. For instance, execution traces can be used in omniscient debugging to travel back in time in the execution to visit previous execution states, to replay past executions, or to retrieve the runtime information about the execution of a model that should be shown to a user.
- *Testing*: refers to techniques for testing models concerning functional or non-functional properties or for testing applications with the help of models concerning functional and non-functional properties. Execution traces can be used in testing, for instance, as oracles or as basis for evaluating test cases providing the necessary runtime information to determine the success or failure of a test case.
- *Manual analysis*: refers to techniques for manually analyzing the execution behavior of a model or the modeled system. Such techniques are mostly concerned with the visualization and querying of model execution traces.
- *Dynamic analysis*: refers to the analysis of runtime information gathered from the execution of a model, similar to the definition of dynamic analysis of programs given in [24]. Thereby, gathered runtime information can be analyzed for different properties, including general behavioral properties, functional properties, and non-functional properties. Execution traces have a natural application in dynamic analysis as they record runtime information about a model or program execution.
- *Model checking*: refers to techniques in which all the possible execution states of a model are checked with respect to some property. Model checking may rely on execution traces for representing the state space of model or for representing counter examples found for violated properties.
- *Semantic differencing*: refers to techniques that compare execution traces of models to understand the semantic differences between them.

#### 3.2.2.5 Data Extraction Techniques (Q5)

This attribute set focuses on the techniques used for the extraction of the traced runtime information during model execution. We categorized the data extraction techniques identified in the investigated primary studies using the following attributes.

- *Source instrumentation*: Elements are added to the executable model, which are responsible for the construction of execution trace.
- *Target instrumentation*: This data extraction technique only concerns model execution tracing approaches considering executable modeling languages with translational semantics. In this technique, elements are added to the target model or target code generated from a model for its execution. These introduced elements are responsible for the construction of execution traces.
- *Interpreter*: This data extraction technique only concerns model execution tracing approaches considering executable modeling languages with operational semantics. In this technique, execution traces are constructed by the interpreter of an executable modeling language (i.e., by the executable modeling language's operational semantics), or by the execution engine responsible for executing the operational semantics of an executable modeling language.
- *External tool*: The runtime information to be recorded in an execution trace is provided by an external tool. Such an external tool could be, for instance, a model checker.
- *Other*: This attribute is assigned to approaches that use none of the data extraction techniques represented by the other attributes.

#### 3.2.2.6 Trace Representation Format (Q6)

This attribute set refers to the kind of format used for the representation of execution traces. We categorized the trace representation formats of model execution tracing approaches using the following attributes.

- *Metamodel*: the data structure used for representing traces is defined using a metamodel.
- *Text format*: the data structure used for representing traces is defined through some well-defined text format. In particular, approaches defining a formal grammar for representing traces or producing traces

in the form of well-structured log outputs fall into this category.
- *Other*: this attribute is assigned to model execution tracing approaches that use trace representation formats other than the ones captured by the attributes given above.
- *Unknown*: this attribute is assigned to approaches where the associated primary studies do not mention the used trace representation format.

#### 3.2.2.7 Trace Representation Method (Q7)

This attribute set refers to the method used for defining the trace representation format. It includes the following attributes.

- *FR (framework)*: refers to approaches that provide a framework for defining custom trace representation formats.
- *AG (automatically generated)*: refers to approaches that automatically generate a trace representation format for a given executable modeling language.
- *MD (manually developed)*: refers to approaches that use a trace representation format manually developed for the respective approach.
- *AE (already existing)*: refers to approaches that rely on some existing trace representation format.
- *Unknown*: refers to approaches where the associated primary studies do not mention the used trace representation method.

#### 3.2.2.8 Language Specificity of Trace Structure (Q8)

This attribute set categorizes model execution tracing approaches concerning the language-specificity of the used trace data structure. For this categorization, we defined the following attributes.

- *Language-independent*: refers to approaches that either rely on generic data structures for representing execution traces, i.e., data structures that can be used to represent execution traces of models conforming to any executable modeling language, or approaches that support the creation of executable modeling language-specific trace data structures but for any executable modeling language.
- *Language-specific*: refers to approaches that rely on a data structure specific to a particular executable modeling language.
- *Specific to a certain kind of language*: refers to approaches that rely on a data structure that is specific to a particular kind of executable modeling language, i.e., trace data structures that do not only support the tracing of the execution of models conforming to a single executable modeling language but that are not general enough to trace models of any executable modeling language.

#### 3.2.2.9 Data Carrier Format (Q9)

This attribute set refers to the format used to store execution traces. Based on the investigated primary studies, we identified the following used data carrier formats.

- *Text*: refers to approaches storing execution traces in simple text files.
- *XML*: refers to approaches storing execution traces in XML syntax, which includes, for instance, XMI files.
- *Database*: refers to approaches storing execution traces in databases.
- *Unknown*: refers to approaches where the associated primary studies do not discuss the supported data carrier formats.

#### 3.2.2.10 Maturity Level (Q10)

With this attribute set, we capture whether model execution tracing approaches offer tool support and how mature this tool support is. To measure the maturity level of approaches, we used the four-level scale proposed by Cuadros Lopéz et al. [27] defined as follows:

- *Level 1 (not implemented)*: The approach is not implemented in a tool.
- *Level 2 (partially implemented)*: The approach is implemented in a prototype tool but not all features are supported.
- *Level 3 (fully implemented)*: The approach is completely implemented in a tool. The tool has been used for several applications to validate the approach.
- *Level 4 (empirical evaluation)*: The approach is completely implemented in a tool and the tool has been evaluated empirically.

### *3.2.3 Article Classification*

This step comprised the assignment of attributes to the selected primary studies, the summarization of primary studies into distinct model execution tracing approaches, and the analysis of the resulting classification of investigated approaches to summarize the research body.

#### 3.2.3.1 Attribute Assignment

We classified the selected primary studies based on the attribute sets defined above. We achieved this by read-

ing the complete content of the primary studies. In particular, the first author did the initial classification by reading the paper and applying the classification, and one of the other authors reviewed the classification by also completely reading the paper and assigning the classification attributes. In case there was a disagreement about the classification, in-depth discussions were done and the paper was re-read by both to come to a common classification. In some cases, we had to review the same primary study several times to make sure that we interpret its content correctly. The classification was done in a spreadsheet that was shared among the authors and also used to keep notes about additional details of the primary studies and exchange comments on individual classifications.

#### 3.2.3.2 Summarization of primary studies

There are cases where multiple primary studies present the same model execution tracing approach but on different levels of detail or in different development stages. For instance, some journal articles selected as primary studies are extensions of earlier work of the authors published in the proceedings of conferences or workshops, which were also selected as primary studies. Thus, we decided to summarize such tightly related primary studies as one approach and analyze the classification of approaches instead of the classification of primary studies. Thereby the classification of an approach is the union of the attributes assigned to all primary studies summarized in this approach. This summarization step yielded 33 approaches from the 64 selected primary studies.

#### 3.2.3.3 Classification Results

The attribute assignment and summarization of the 64 selected primary studies resulted in a classification of 33 approaches, which can be regarded as the body of knowledge in model execution tracing. The results of the classification are presented in Figure 8, which shows the frequency in which the individual attributes have been assigned to the investigated model execution tracing approaches per research question. These results are discussed in detail in Section 4. The attributes assigned to each individual approach are shown in Table 2 (for Q1-Q3), Table 3 (for Q4-Q5), and Table 4 (for Q6-Q10), which are given in the end of this paper. In these tables, the investigated approaches are numbered from A01 to A33.

All artifacts prepared for this work including the spreadsheet containing the classification of the selected primary studies and bibliographic information have been collected in a replication package that has been made publicly available[1].

## 4 Results

In this section, we discuss the classification results of the investigated approaches for each research question as given in Fig. 8 and Table 2-4.

### 4.1 Types of Models (Q1)

We found that concerning the targeted executable modeling languages, the investigated model execution tracing approaches can be classified into three categories where each category comprises around one third of the approaches: 36% of the approaches target UML models, 30% target workflow models (attribute 'Worflow models' assigned to 9% of approaches) or models conforming to other executable modeling languages (attribute 'Other' assigned to 21% of approaches), and 33% are independent of any executable modeling language.

Most of the approaches targeting UML models are geared towards tracing UML activity diagrams or UML state machines. Other behavioral UML diagrams, such as UML sequence diagrams, have been a target only by a few approaches in this category. Note that we classified approaches as supporting UML models only if the authors explicitly stated this in the respective paper.

A small number of approaches 9% is devoted to workflow models. Other executable modeling languages are targeted by 21% approaches: MCSE description models [18] are targeted in A01 [92], stochastic discrete event simulation models in A10 [67, 68], COLA models [72] in A12 [49, 50], CCSL clock constraint specifications in A16 [29, 30, 44], story diagrams in A18 [71], live sequence charts in A28 [74], and Event-B models in A30 [64]. It is worth noting that each of these executable modeling languages is targeted by exactly one of the investigated approaches, i.e., none of them is addressed in two or more approaches.

Especially in recent years it seems that more attention is directed towards approaches that provide generic tracing mechanisms that are applied on models conforming to any executable modeling language: 64% of these approaches (seven out of eleven) appeared in the last five years (publication dates from 2013 to 2018), while only 36% (four out of eleven) occurred between 2008 and 2011.

[1] https://drive.google.com/drive/folders/1wX1xu10bd5vmXp_UDIjRFB2_WhmFBx5-?usp=sharing

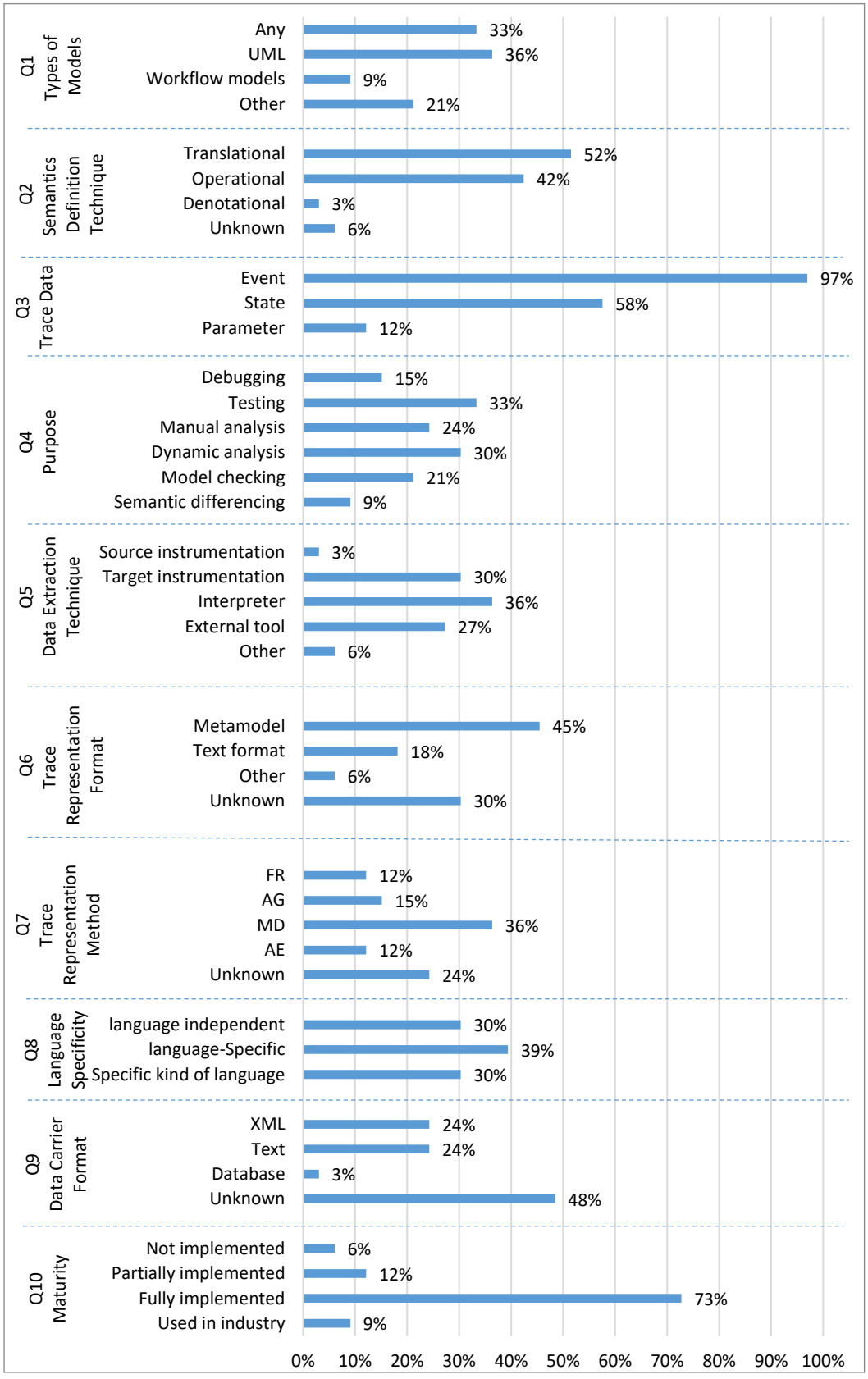


**Fig. 8** Classification of model execution tracing approaches

### 4.2 Semantics Definition Technique (Q2)

Concerning the semantics definition technique, we discovered that about one half of the investigated model execution tracing approaches assume that the execution semantics of supported executable modeling languages are defined in a translational way (52%), while the other half of the approaches rely on operational semantics (42%). Only one approach, A07 [20, 21, 22], supports both translational and operational semantics. However, the authors introduce only a very abstract pattern of how to design executable modeling languages as well as tracing infrastructures for such executable modeling languages, rather than providing a concrete tracing infrastructure or tooling that could be directly used.

Approach A26 [37, 38] considers denotational semantics; hence, it falls into the category "denotational".

For two approaches, we could not identify the supported semantics definition techniques from the respective papers. We assigned them to the category "Unknown".

We conclude that the majority of existing model execution tracing approaches supports executable modeling languages with either operational semantics or translational semantics, while there are no concrete solutions for offering model execution tracing capabilities to executable modeling languages irrespective of how their execution semantics are defined.

Another interesting finding is that the majority of approaches applicable to any executable modeling language, namely 64%, rely on operational semantics. In contrast, 67% of the approaches targeting UML rely on translational semantics. The latter confirms the finding by Ciccozzi et al. [19] that translational semantics are predominantly used for the execution of UML models—in 85% of the investigated solutions—rather than operational semantics.

### 4.3 Trace Data (Q3)

All of the investigated approaches except one (97%) trace events that occur during the execution of a model. Thereby, 42% of all approaches only trace execution events by producing traces that are basically sequences of events that occur during a model execution. Equally many approaches (again 42%) do trace besides execution events also information about the evolution of the execution state of models. Only a very small fraction of the investigated studies, namely 12% capture rich traces that record execution events, execution states, as well as inputs and outputs. This applies to the approaches A09 [77, 78, 79], A12 [49, 50], A20 [82, 83, 84, 87, 88], and A21 [80, 81]. Approach A02 [106] is the only one that does not trace execution events but only execution state information.

### 4.4 Purpose (Q4)

Most existing model execution tracing approaches have been used for testing 33% or dynamic analysis 30%. They are followed by 24% of approaches applied for manual analysis of model behaviours, and 21% of techniques used for model checking. Only few approaches have been applied for debugging and semantic model differencing, namely 15% and 9%.

The low number of approaches applied for debugging and semantic model differencing is due to two reasons: First, in modeling, traces are typically used for a specific type of debugging, omniscient debugging and debugging previous execution (i.e., replaying executions). The second reason is that semantic model differencing is relatively a new research area, at early research development stages.

Interestingly, most of the investigated approaches, namely 70%, have been applied on one type of model analysis technique only, whereas only 27% of approaches have been applied to realize two different kinds model analysis techniques. The most common combination of model analysis techniques is the combination of manual analysis and dynamic analysis, which is reported by 15% of the investigated approaches (or 56% of the approaches applied on a combination of two analysis techniques). It is also worth mentioning that manual analysis is the model analysis technique that has been most often combined with other model analysis techniques (in 21% of all approaches or 78% of approaches applied on a combination of two analysis techniques).

There is only one approach, A20 [82, 83, 84, 87, 88], which uses traces for realizing three different types of model analysis tasks, namely testing, dynamic analysis, and debugging.

Another interesting finding is that more than half of the investigated model execution tracing approaches that are applicable to any executable modeling language are used for model checking, namely 55%. The other model analysis techniques are only considered by 1-2 approaches each. In contrast, half of the approaches targeting UML have been applied for model testing, followed by 33% of approaches applied for manual analysis, and 25% of approaches applied for dynamic analysis. Only two of the approaches targeting UML have been applied for debugging, and one each for model checking and semantic model differencing.

### 4.5 Data Extraction Techniques (Q5)

The most common data extraction techniques used by the investigated model execution tracing approaches are tracing by an interpreter (36%), tracing through target instrumentation (30%), and tracing by an external tool (27%).

As already discussed in Section 3.2.2, tracing by an interpreter only concerns executable modeling languages with operational semantics. From the model execution tracing approaches supporting operational semantics, 86% rely on this data extraction technique. The other 14% rely on external tools for extracting the runtime data to be traced.

Similarly, tracing through target instrumentation is only applicable for approaches supporting translational semantics. From the approaches supporting translational semantics, 59% rely on target instrumentation for data extraction. The majority of the remaining model execution tracing approaches supporting translational semantics extract runtime data through external tools, namely 29%. Only one approach supporting translational semantics, approach A02 [106], relies on source instrumentation for data extraction, and one approach, approach A12 [49, 50], relies on a middleware that is part of the targeted execution platform for data extraction and is hence classified as "Other" for the data extraction technique.

It is also worth mentioning that approach A07 [20, 21, 22], which supports both translational and operational semantics (cf. Section 4.2), uses a model checker (i.e., an external tool) for constructing traces when execution semantics are defined in a translational way, and construct traces through the interpreter when the execution semantics are defined in an operational way.

Looking at the "External Tool" category, 44% of the approaches assigned to this category support only translational semantics, 22% support only operational semantics, and 11% (1 approach) support both translational and operational semantics. For 22% of the approaches extracting data through external tools, the supported semantics definition technique is unknown.

For the "Other" category, we already mentioned approach A12 [49, 50], which supports translational semantics and relies on a middleware for extracting the runtime information to trace. The second approach assigned to this category, approach A26 [37, 38], supports denotational semantics and traces are directly constructed through the denotational semantics implemented in Haskell.

From this data, we conclude that for model execution tracing approaches supporting operational semantics, runtime data is extracted primarily by the interpreter, while for approaches supporting translational semantics, runtime data is extracted primarily through target instrumentation or external tools.

### 4.6 Trace Representation Format (Q6)

Our results on trace representation formats clearly indicate that metamodels are most frequently used to define the data structure for representing model execution traces. In particular, 45% of the investigated approaches rely on metamodels. This result was expected, since we study execution tracing approaches for executable models and executable models commonly instantiate metamodels as well.

More surprisingly, only 18% of model execution tracing approaches define textual trace representation formats and from these approaches, only one approach, approach A26 [37, 38], actually defines a grammar for representing traces while the others use a less formal trace format, such as structured logs.

Approach A10 [67, 68] uses an XML format for representing traces and approach A31 [58, 59] uses a graph-based representation. These two approaches (6%) have been classified as "Other".

For almost one third of the investigated approaches (30%), we could not identify the used trace representation format. Hence, these approaches have been assigned to the category "Unknown" for Q6.

### 4.7 Trace Representation Method (Q7)

Our classification results show that nearly half of the investigated model execution tracing approaches (48%) use a selected trace representation format. In particular, more than one third of the approaches (36%) use trace representation formats that were manually developed particularly for the respective approach, while only 12% rely on existing trace formats. An example of the latter case is approach A30 [64] which reuses event trace diagrams (ETDs) proposed by Rumbaugh et al. [95]) as trace format.

In contrast, 27% of approaches support custom trace representation formats that are tailored towards the executable modeling language used to define the traced executable models. In particular, 15% of approaches automatically generate trace representation formats for executable modeling languages, while 12% of approaches provide a framework for manually defining custom trace representation formats. However, it has to be noted that this kind of approaches are a minority. An example of an automated approach is the generative approach A23 [9, 10, 12, 13, 14], which automatically de-

rives multidimensional domain-specific trace metamodels for xDSMLs. Approach A22 [4, 5, 6, 7] is an example of a framework for manually defining custom trace formats. In particular, it allows the definition of textual trace formats for UML models using so-called *trace directives*.

For the last category of approaches comprising 24%, the trace representation method is unknown, i.e., not mentioned by the associated primary studies.

### 4.8 Language Specificity of Trace Structure (Q8)

Concerning the language specificity of the used trace data structure, the investigated approaches can be categorized into three groups of almost equal size: 39% of the approaches use a trace data structure that includes concepts specific to a certain executable modeling language, 30% of the approaches use a trace data structure that can be reused for executable modeling languages of a specific kind, and 30% of the approaches use a trace data structure that is independent of any executable modeling language.

The majority of the approaches using a trace data structure specific to a particular executable modeling language target the UML language, namely 61%. In contrast, no particular trend could be observed for the approaches using trace data structures specific to a certain kind of executable modeling language. The approaches using a language-independent trace format target no particular executable modeling language or type of executable modeling language but support any executable modeling language.

### 4.9 Data Carrier Format (Q9)

Only little information could be extracted from the primary studies concerning used data carrier formats for storing execution traces: For almost half of the approaches (48%), no data carrier formats have been mentioned.

The other half of the investigated approaches either uses an XML format or a plain text format for storing execution traces, namely 24% each.

Only one approach, approach A19 [35], persists execution traces in a database. In this approach, a UML profile is generated for tracing system executions using a UML state machines. A persistence component transmits the runtime data obtained from the execution to a trace database.

### 4.10 Maturity Level (Q10)

Our results for the classification of model execution tracing approaches concerning maturity level show that the majority of investigated approaches (73%) is fully implemented. From this we conclude that the field of model execution tracing has reached a moderate level of maturity. However, among the investigated approaches, only three (9%) have been subjected to an empirical evaluation, namely approach A05 [40, 41, 42, 43] implemented in the Populo tool, approach A31 [58, 59] implemented in the TRACE tool, and approach A33 [97] implemented for the tool UPPAAL. In fact, the most common method used to validate the investigated approaches is through case studies. While the majority of the case studies demonstrate the complete implementation of the approaches, they fail to show the approaches' usefulness in industrial settings. Hence, little is known about the value of existing model execution tracing approaches in industry and evaluations of this aspect are hence needed to further mature this research field.

## 5 Future Research Directions

In the following, we discuss directions for future work on model execution tracing, building upon our research results presented in Section 4. In our opinion, it is necessary to address the following topics not only from a research perspective, but in collaboration with tool vendors and end users to ensure widespread tool support and to achieve industry adoption.

**Scalable trace data structure**: From the results obtained for research question Q3 on the kind of traced data, we can see that in fact a lot of data is recorded by existing model execution tracing approaches: Almost all approaches record information about occurred execution events and more than 40% keep detailed execution state information. This means, however, that traces are expected to grow large, which may cause scalability issues in both memory needed for storing traces and time needed for processing them. Nevertheless, we could only identify three of the investigated approaches that aim at addressing these scalability issues. In particular, Bousse et al. aim to address this issue in their approach A23 [9, 10, 12, 13, 14] by sharing data among captured states so that only changes in data are recorded. Similarly, Hegedus et al. propose in their approach A14 [57] to reduce traced state information by only capturing state modifications and events related to state modifications. In contrast, Kemper and Tepper propose in their approach A10 [67, 68] to remove repetitive fragments from traces using heuristic methods, such as cycle reduction. While the afore-

mentioned approaches consider some sort of trace compaction, they utilize different optimization potentials and leave open whether the achievable compaction is sufficient for industry applications. Thus, we see the need for more detailed studies on scalable model execution tracing solutions. One direction is to investigate the use of techniques for simplifying execution traces used in code-centric approaches such as the ones presented by Cornelissen et al. [25] and Hamou-Lhadj et al. [54].

**Common trace exchange format**: The results obtained for research question Q7 on trace representation methods clearly shows that most existing model execution tracing approaches rely on their own custom trace formats. Only four of the investigated approaches reuse an already defined trace format. Giving this large variety of trace formats, it is apparent that a common format for exchanging model execution traces is needed. Such an exchange format, however, has to support the representation of executable modeling language-specific concerns in different levels of detail. This is indicated by the results obtained for the research questions Q3 and Q8 that show that existing model execution tracing approaches record information specific to particular executable modeling languages, and that besides execution events, information about execution states and processed inputs are also relevant to be traced in specific contexts. A common trace format should be expressive enough to capture the required runtime information for any executable modeling language. Also, it should represent traces in a compact form to enable scalability of the analysis tools. Thereby, scalability should be considered as a key requirement when defining a common trace format. Examples of trace formats for traces generated from code-centric approaches are Compact Trace Format(CTF) proposed by Hamou-Lhadj and Lethbridge [52, 55] and Message Passing Interface Trace Format (MTF) proposed by Alawneh and Hamou-Lhadj [3]. These trace formats model traces of routine calls and inter-process traces, respectively, in a compact way, in order to facilitate efficient interchange of traces among trace analysis tools. A similar effort should be invested in defining standard trace formats for traces of model executions that would facilitate interoperability among V&V tools and hence make V&V tools available to a broader user base.

**Support of multiple semantics definition techniques**: Our results for research question Q2 show that all approaches except one support either translational semantics or operational semantics but not both. Executable languages use many different techniques for defining operational semantics/ interpreters (e.g., programming languages, action languages, and model transformation languages) and translational semantics/ compilers (e.g., model-to-model transformation languages, target modeling languages, code generators, target programming languages) [84]. Model execution tracing approaches focus on one of these techniques (either translational or operational). Therefore, they are applicable in a very narrow scope. By supporting different semantics definition techniques, we can reuse the same model tracing approach in more scenarios. It is based on the separation of concerns principle in order to separate the concern of how to implement an executable language (i.e., semantics definition technique) from how to trace model executions. This would enable a broader adoption of model execution tracing techniques for executable modeling languages implemented in different ways. This would also enable the application of V&V tools for executable modeling languages defined with either semantics definition technique.

**Empirical validation**: While the majority of investigated model execution tracing approaches has been implemented in prototype tools, there exists very little empirical evidence about the usefulness of these approaches for industry. Even with complete demonstrations, a considerable amount (more than 90%) of the approaches also lacks any empirical evaluation, as shown by our results obtained for research question Q10. No approaches have been evaluated in industrial settings. To mature the field of model execution tracing, empirical validations of existing solutions and validations in industry settings need to be performed.We intend in the future to work on investigating the state of adoption of model execution tracing in industry. We also intend to work with developers of model-driven systems to understand the state of practice of model execution tracing and what the challenges developers face when using the related techniques.

## 6 Limitations and Threats to Validity

Despite the care taken in the definition of the research method, our mapping study is subject to known threats and limitations. The most serious threats to the validity of our research results are researchers' bias in searching, selecting, and classifying studies. To mitigate this risk, we applied and strictly followed the guidelines suggested by Kitchenham and Charters [16, 69] and Petersen et al. [93].

To mitigate the risk of missing relevant studies, we have performed automated searches in the most popular digital online libraries in the field of software engineering. The search strings used for this have been derived from the defined research questions. Furthermore, we have performed a forward snowballing step

to identify additional studies that could be relevant for our research.

To ensure the reliability of our selection criteria for primary studies as well as ensure the quality of selected primary studies, we have also performed a pilot study and a quality assessment. Through the pilot study, we could improve the original selection criteria and the quality assessment showed that the selected primary studies are of good quality.

In order to reduce the threat of misclassifying the selected primary studies, we reviewed their full texts thoroughly instead of reviewing only abstracts, introductions, and conclusions. Furthermore, the classification of each primary study was reviewed by at least one of the authors that was not involved in the original classification. Any discrepancies were resolved by reading the affected primary study again and discussing its classification in detail.

## 7 Related Work

There exist several systematic review studies that have been conducted in the context of MDE. However, none of these studies target approaches for model execution tracing approaches. To the best of our knowledge, this is the first study that aims to survey the state of the art in this area. In this section, we summarize recent surveys in the domain of MDE that are related to our study.

Ciccozzi et al. [19] conduced a systematic review of research studies and tools concerned with the execution of UML models, which is also considered in 36% of model execution tracing approaches investigated in our work. The authors analyzed the identified research studies on UML model execution concerning publication trends, technical characteristics, and evidence provided on industry adoption. Tools were analyzed concerning technical characteristics only, such as UML modeling characteristics (required diagrams, use of action languages, etc.), execution strategy (translation, interpretation, execution tools and technologies, etc.), intended benefits, and readiness level among other characteristics. The findings show a growing scientific interest in UML model execution starting from 2008, which is consistent with our findings for model execution tracing, which show an increase in publications on the topic from 2007. Furthermore, the study revealed that translational semantics has been predominantly used for the execution of UML models rather than operational semantics. Also this finding is consistent with the results of our study. As intended benefits of UML model execution, the study identified reducing the effort for producing executable artifacts and improving the functional correctness of models as the main benefits targeted, the latter being highly related to model execution tracing as it provides the basis for many dynamic V&V techniques used for ensuring functional correctness. However, the study does not investigate in detail the types of V&V techniques provided or applied by the identified research studies and tools. It only investigates whether model-level interactive debugging, model simulation (i.e., execution of models for analysis rather than execution on the target platform), and formal specification languages (e.g., for the purpose of model checking) are supported with the result that model-level interactive debugging and formal specification languages are only supported by few approaches while model simulation is supported by half of the approaches letting the authors suggest that model execution is considered beneficial for early design assessment. In contrast, we investigate in more detail which kinds of dynamic V&V techniques are realized based on model execution tracing. Interestingly, the authors identify the need to further enhance the observability of execution models, which includes the ability to record, play back, and analyze execution traces of system operation on the target platform or in a simulation. The last finding that we want to highlight is that most of the analyzed UML model execution solutions have a low technology readiness level and that only a few of the investigated research studies provide evidence through experimentation in industrial settings and based on empirical evaluations. This is also true for the model execution tracing approaches studied in our work.

Szvetits and Zdun [101] applied a systematic literature review for the purpose of classifying and analyzing existing approaches for using models at runtime for self-adapting systems. The existing approaches have been classified concerning objectives, techniques, architectures, and kinds of models used. The authors revealed the usage of different kinds of models at runtime to achieve various objectives, such as adaptation, policy checking, and error handling. Related to our study, the authors identified monitoring as one objective of models at runtime defining monitoring as the activity of monitoring "the system by using models which help to trace application behavior". However, other than model execution tracing targeted in this study, models at runtime trace the execution of an application rather than the execution of models. As discussed by the authors, this distinction becomes blurry when executable models are in fact the executable application, i.e. when there is no other implementation-level artifact manually or automatically generated from executable models. The authors point out that in such scenarios, it is not clear how model execution fits into the models at runtime

paradigm. However, model execution in general definitely has a role in the models at runtime paradigm, as it facilitates the analysis of model-based representations of application behavior to, for example, simulate the consequences of runtime adaptations or to predict system properties influencing runtime adaptation. Nevertheless, we do not see a direct relationship between model execution tracing and models at runtime.

Dias Neto et al. [33] conducted a systematic review of Model-Based Testing (MBT) approaches. The reviewed approaches have been categorized concerning testing level, tool support, application scope, kind of models used for test case generation, used test coverage criteria, used test case generation criteria, and level of automation. Among other findings, the study revealed that there is a need for providing MBT solutions for testing non-functional requirements, such as usability, security, and reliability, and that most MBT approaches have not been evaluated empirically or transferred to industry. Recently, Gurbuz and Tekinerdogan [48] conducted a systematic mapping study to identify and analyze the state of the art in MBT for software safety. The study revealed that 42% of the investigated primary studies were validated using industrial evidence but that they nevertheless provide no strong evidence of positive effects of MBT for software safety. Related to these findings, we have identified two model execution tracing solutions applied for MBT: In [74], MBT is used for testing functional requirements, while in [103], security testing is considered. Furthermore, we have found most existing model execution tracing approaches to also lack an empirical evaluation in industrial settings.

Nguyen et al. [89] conducted a systematic literature review on Model-Driven Security (MDS), which is the application of MDE techniques and technologies to the development of secure systems. The authors categorized the identified MDS approaches concerning considered security concerns, applied modeling approach, used model-to-model and model-to-text transformations, application domains, and evaluation methods. The results revealed the need for addressing several security concerns that have been mostly neglected by existing approaches, as well as multiple security concerns simultaneously and in a systematic manner. Furthermore, they discovered a lack of tool support and empirical evaluations. Related to our work, the authors identified that DSMLs play a key role in MDS but that most of the currently existing security DSMLs lack semantic foundation required for automated analysis. They also point out that behavioral models are rarely used but most approaches employ structural models only, which hampers the ability to deal with multiple security concerns simultaneously.

Nascimento et al. [34] conducted a systematic mapping study on Domain Specific Languages (DSLs) to identify existing DSLs, their application domains, tools for their development and usage, as well as techniques, methods, and processes for creating, applying, evolving and extending DSLs. The study surveys DSLs on a high level of abstractions and provides only a high-level overview of existing DSLs and DSL engineering approaches. While DSMLs are identified as one specific kind of DSLs, no investigations targeting specifically executable modeling languages, model execution, or model execution tracing have been performed.

Giraldo et al. [45] conducted a systematic review to identify definitions of quality in MDE. The authors discovered that only 16 out of 134 reviewed studies provide explicit definitions of quality and that these definitions mostly concern the quality of models or the quality of modeling languages. The remaining studies do not provide such an explicit definition. Among them, 40% of the studies propose solutions for quality assurance, such as behavioral verification of models, performance models, and model metrics. This study is related to our work in the sense that model execution tracing approaches in general aim at enabling the performance of dynamic V&V for ensuring the quality of models or modeled systems in terms of functional or non-functional quality properties. To investigate this aspect, we review in this study the objectives of existing model execution tracing solutions.

Santiago et al. [96] conducted a systematic literature review to analyze the current state of the art in the management of traceability in MDE approaches. However, other than in our work, the considered notation of traceability refers to the establishment of relationships between products of the development process and is hence unrelated with the tracing of model executions.

## 8 Conclusion

In this paper, we presented a systematic mapping study on existing approaches for the tracing of model executions. With this study we aim at identifying and classifying the existing approaches, thereby assessing the state of the art in this area, as well as pointing to promising directions for further research in this area.

From 645 research studies found through automatic searches in popular academic online libraries, we finally selected and analyzed 64 primary studies that present 33 unique model execution tracing approaches. These 33 identified approaches were classified concerning supported types of models, supported execution semantics definition technique, traced data, purpose, data extraction technique, trace representation format, trace rep-

resentation method, language specificity, data carrier format, and maturity.

Our findings show that (i) the majority of approaches target specific executable modeling languages with UML being the most popular one; (ii) almost all model execution tracing approaches either support exclusively executable modeling languages with operational semantics or executable modeling languages with translational semantics; (iii) besides execution events traced by almost all approaches, a significant amount of approaches also record detailed information about execution states; (iv) the majority of model execution tracing approaches has been applied on one kind of model analysis technique only with testing and dynamic analysis being the most frequently used ones; (v) approaches supporting operational semantics rely mainly on executable modeling language interpreters for extracting tracing information, while approaches supporting translational semantics rely mostly on target instrumentation; (vi) metamodels are most frequently used for defining the trace representation format; (vii) thereby, trace representation formats are mostly specific to the respective approach with only a minority of approaches that reuse existing formats; (viii) trace representation formats are mostly dependent on the supported executable modeling language or specific to a certain kind of executable modeling languages; (ix) traces are serialized either in XML format or as plain text; and (x) only a small minority of approaches have been empirically validated.

The results suggest that more research is needed particularly on suitable trace representations and broad applicability of approaches with scalability and interoperability being two concerns that have been mostly neglected so far. Furthermore, empirical validations of the usefulness of approaches in real application scenarios are needed to foster the adoption of model execution tracing approaches in practice.

## Acknowledgement

This work is partially supported by Iranian Ministry of Science, Research and Technology and Isfahan University under the IMPULS Iran-Austria contract no. 4/11937.

## References

1. Richard J Adams, Palie Smart, and Anne Sigismund Huff. Shades of grey: guidelines for working with the grey literature in systematic reviews for management and organizational studies. *International Journal of Management Reviews*, 19(4):432–454, 2017.
2. Luay Alawneh and Abdelwahab Hamou-Lhadj. *Execution traces: A new Domain that requires the Creation of a Standard Metamodel*, volume 59 of *Lecture Notes in Communications in Computer and Information Science book series*, pages 253–263. Springer, 2009.
3. Luay Alawneh and Abdelwahab Hamou-Lhadj. An exchange format for representing dynamic information generated from High Performance Computing applications. *Future Generation Computer Systems*, 27(4):381–394, 2011.
4. Hamoud Aljamaan and Timothy C Lethbridge. Towards Tracing at the Model Level. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 495–498. IEEE, 2012.
5. Hamoud Aljamaan, Timothy C Lethbridge, Omar Badreddin, Geoffrey Guest, and Andrew Forward. Specifying trace directives for UML attributes and state machines. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 79–86. IEEE, 2014.
6. Hamoud Aljamaan, Timothy C. Lethbridge, and Miguel A. Garzón. MOTL: A Textual Language for Trace Specification of State Machines and Associations. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 101–110, Riverton, NJ, USA, 2015. IBM Corp.
7. Hamoud I Aljamaan, Timothy Lethbridge, Miguel Garzón, and Andrew Forward. UmpleRun: a dynamic analysis tool for textually modeled state machines using Umple. In *Proceedings of the First International Workshop on Executable Modeling co-located with MODELS 2015*, pages 16–20, 2015.
8. Earl T. Barr and Mark Marron. Tardis: Affordable Time-travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*, pages 67–82. ACM, 2014.
9. Erwan Bousse, Benoit Combemale, and Benoit Baudry. Towards Scalable Multidimensional Execution Traces for xDSMLs. In *Proceedings of the 11th Workshop on Model Design, Verification and Validation Integrating Verification and Validation in MDE (MoDeVVa 2014)*, volume 1235, pages 13–18, 2014.
10. Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting

efficient and advanced omniscient debugging for xDSMLs. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–148. ACM, 2015.

11. Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien DeAntoni, and Benoît Combemale. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16)*, pages 84–89. ACM, 2016.
12. Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for Executable DSLs. *Journal of Systems and Software*, 137:261–288, 2018.
13. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *European Conference on Modelling Foundations and Applications*, volume 9153 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2015.
14. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. Advanced and Efficient Execution Trace Management for Executable Domain-specific Modeling Languages. *Software and Systems Modeling*, pages 1–37, 2017.
15. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven Software Engineering in practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, second edition, 2017.
16. Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from Applying the Systematic Literature Review process within the Software Engineering Domain. *Journal of systems and software*, 80(4):571–583, 2007.
17. Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 2(8):225–253, 2011.
18. Jean Paul Calvez. *Embedded Real-time Systems. A specification and Design Methodology*. John Wiley, 1993.
19. Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, 2018.
20. Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software (JSW)*, 4(9):943–958, 2009.
21. Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to build Executable DSMLs and associated V&V tools. In *Proceedings of the 19th Asia-Pacific on Software Engineering Conference (APSEC)*, volume 1, pages 282–287. IEEE, 2012.
22. Benoit Combemale, Xavier Crgut, Jean-Pierre Giacometti, Pierre Michel, and Marc Pantel. Introducing simulation and model animation in the MDE Topcased toolkit. In *Proceedings of the 4th European Congress Embedded Real Time Software (ERTS)*, 2008.
23. Benoit Combemale, Laure Gonnord, and Rusu Rusu. A Generic Tool for Tracing Executions back to a DSMLs Operational Semantics. In *European Conference on Modelling Foundations and Applications*, volume 6698, pages 35–51. 2011.
24. Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transaction on Software Engineering*, 35(5):684–702, 2009.
25. Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. pages 684–702, 2009.
26. Michelle L. Crane and Juergen Dingel. Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In *Conference of the center for advanced studies on collaborative research*, pages 96–110. ACM, 2008.
27. Álvaro Julio Cuadros López, Carolina Galindres, and Paola Ruiz. Project maturity evaluation model for SMEs from the software development sub-sector. *AD-minister*, (29):147–162, 2016.
28. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
29. Julien DeAntoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS)*, volume 7304, pages 34–41. Springer, 2012.
30. Julien DeAntoni, Frédéric Mallet, Frédéric Thomas, Gonzague Reydet, Jean-Philippe Babau, Chokri Mraidha, Ludovic Gauthier, Laurent Rioux, and Nicolas Sordon. RT-simex: retro-analysis of execution traces. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 377–378. ACM, 2010.

31. Anna Derezinska and Marian Szczykulski. Tracing of state machine execution in the model-driven development framework. In *Proceedings of the 2nd International Conference on Information Technology, ICIT 2010*, pages 517–524. IEEE, 2010.
32. Romuald Deshayes, Bart Meyers, Tom Mens, and Hans Vangheluwe. ProMoBox in Practice: A Case Study on the GISMO Domain-Specific Modelling Language. In *Proceedings of the 8th Workshop on Multi-Paradigm Modelling (MPM)*, pages 21–30, 2014.
33. Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A Survey on Model-based Testing Approaches: a Systematic Review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.
34. Leandro Marques do Nascimento, Daniel Leite Viana, Paulo AM Silveira Neto, Dhiego AO Martins, Vinicius Cardoso Garcia, and Silvio RL Meira. A Systematic Mapping Study on Domain Specific Languages. In *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA12)*, pages 179–187, 2012.
35. Eladio Domínguez, Beatriz Pérez, and María A Zapata. A UML profile for dynamic execution persistence with monitoring purposes. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, pages 55–61. IEEE, 2013.
36. João Pascoal Faria and Ana CR Paiva. A Toolset for Conformance Testing against UML sequence diagrams based on event-driven colored Petri nets. *International Journal on Software Tools for Technology Transfer*, 18(3):285–304, 2016.
37. CA Fernández-Fernández and AJH Simons. An Implementation of the Task Algebra, a Formal Specification for the Task Model in the Discovery Method. *Journal of applied research and technology*, 12(5):908–918, 2014.
38. Carlos Alberto Fernández-Fernández and Anthony JH Simons. An Algebra to Represent Task Flow Models. *International Journal of Computational Intelligence: Theory and Practice*, 6(2):63–74, 2011.
39. Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on the Theory and Application of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.
40. Lidia Fuentes, Jorge Manrique, and Pablo Sánchez. Execution and simulation of (profiled) UML models using Populo. In *Proceedings of the international workshop on Models in software engineering*, pages 75–81. ACM, 2008.
41. Lidia Fuentes and Pablo Sánchez. Designing and Weaving Aspect-Oriented Executable UML Models. *Journal of Object Technology*, 6(7):109–136, 2007.
42. Lidia Fuentes and Pablo Sánchez. Towards Executable Aspect-Oriented UML Models. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 28–34. ACM, 2007.
43. Lidia Fuentes and Pablo Sánchez. Dynamic Weaving of Aspect-Oriented Executable UML Models. *Transactions on Aspect-Oriented Software Development*, 5560:1–38, 2009.
44. Kelly Garcés, Julien Deantoni, and Frédéric Mallet. A model-based approach for reconciliation of polychromous execution traces. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 259–266. IEEE, 2011.
45. Fber D Giraldo, Sergio Espana, and Oscar Pastor. Analyzing the concept of Quality in Model-Driven Engineering Literature: A systematic review. In *Proceedings of the 8th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE, 2014.
46. Ankit Goel, Bikram Sengupta, and Abhik Roychoudhury. Footprinter: Round-trip engineering via scenario and state based models. In *Proceedings of the 31st International Conference on Software Engineering - Companion Volume, ICSE-Companion*, pages 419–420. IEEE, 2009.
47. Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In *Modellierung*, volume 225, pages 273–288, 2014.
48. Havva Gulay Gurbuz and Bedir Tekinerdogan. Model-based Testing for Software Safety: a Systematic Mapping Study. *Software Quality Journal*, pages 1–46, 2017.
49. Wolfgang Haberl, Jan Birke, and Uwe Baumgarten. A Middleware for Model-Based Embedded Systems. In *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, pages 253–259, 2008.

50. Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten. Model-level debugging of Embedded Real-time Systems. In *Proceedings of the 10th international conference on Computer and information technology (CIT)*, pages 1887–1894. IEEE, 2010.
51. Abdelwahab Hamou-Lhadj. Techniques to simplify the analysis of execution traces for program comprehension. *Doctoral Dissertation, University of Ottawa Ottawa, Ontario, Canada*, 2006.
52. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A metamodel for dynamic information generated from object-oriented systems. *Electronic Notes in Theoretical Computer Science*, 94:59–69, 2004.
53. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, pages 42–55. IBM Press, 2004.
54. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *Proceedings of the 14th International Conference on Program Comprehension, ICPC*, pages 181–190. IEEE, 2006.
55. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A metamodel for the compact but lossless exchange of execution traces. *Software and Systems Modeling*, 11(1):7798, 2012.
56. Abel Hegedus, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of simulation traces with change-driven model transformations. In *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 145–155. IEEE, 2010.
57. Abel Hegedus, Gábor Bergmann, István Ráth, and Dániel Varró. Replaying execution trace models for dynamic modeling languages. *Periodica Polytechnica Electrical Engineering and Computer Science*, 56(3):71–82, 2013.
58. Martijn Hendriks and Frits W Vaandrager. Reconstructing Critical Paths from Execution Traces. In *Proceedings of the 15th International Conference on Computational Science and Engineering (CSE)*, pages 524–531. IEEE, 2012.
59. Martijn Hendriks, Jacques Verriet, Twan Basten, Bart Theelen, Marco Brassé, and Lou Somers. Analyzing Execution Traces: Critical-path Analysis and Distance Analysis. *International Journal on Software Tools for Technology Transfer*, 19(4):487–512, 2016.
60. Frank Hilken and Martin Gogolla. Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams Using Filmstripping. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 708–713. IEEE, 2016.
61. Frank Hilken, Lars Hamann, and Martin Gogolla. Transformation of UML and OCL models into Filmstrip Models. In *International Conference on Theory and Practice of Model Transformations*, volume 8568 of *Lecture Notes in Computer Science*, pages 170–185. Springer, 2014.
62. Fazilat Hojaji, Bahman Zamani, and Abdelwahab Hamou-Lhadj. Towards a tracing framework for Model-Driven software systems. In *Proceedings of the 6th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 298–303. IEEE, 2016.
63. Zhaoxia Hu and Sol M Shatz. Mapping UML Diagrams to a Petri Net Notation for System Simulation. In *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 213–219. Citeseer, 2004.
64. Adisak Intana, Michael R Poppleton, and Geoff V Merrett. A model-based trace testing approach for validation of formal co-simulation models. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 181–188. Society for Computer Simulation International, 2015.
65. Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.
66. Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.
67. Peter Kemper and Carsten Tepper. Automated analysis of simulation traces-separating progress from repetitive behavior. In *Proceedings of the Fourth International Conference on the Quantitative Evaluation of Systems.QEST 2007*, pages 101–110. IEEE, 2007.
68. Peter Kemper and Carsten Tepper. Automated trace analysis of discrete-event system models. *IEEE Transactions on Software Engineering*, 35(2):195–208, 2009.
69. Barbara Kitchenham and Stuart Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Report, Software Engineering Group, School of Computer Science and Mathematics, Keele University, 2000.
70. Johan Kraft, Anders Wall, and Holger M Kienle. Trace Recording for Embedded Systems: Lessons

Learned from Five Industrial Projects. In *Proceedings of the International Conference on Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2010.
71. Alexander Krasnogolowy, Stephan Hildebrandt, and Sebastian Wätzoldt. Flexible debugging of behavior models. In *IEEE International Conference on Industrial Technology (ICIT)*, pages 331–336. IEEE, 2012.
72. Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, et al. COLA–The component language. Technical report, 2007.
73. Philip Langer, Tanja Mayerhofer, and Gerti Kappel. Semantic model differencing utilizing behavioral semantics specifications. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2014.
74. Liping Li, Xingsen Li, and Shan Tang. Research on web application consistency testing based on model simulation. In *Proceedings of the 9th International Conference on Computer Science and Education (ICCSE)*, pages 1121–1127. IEEE, 2014.
75. Jiexin Lian, Zhaoxia Hu, and Sol M Shatz. Simulation-based analysis of UML statechart diagrams: methods and case studies. *Software Quality Journal*, 16(1):45–78, 2008.
76. Bruno Lima and João Pascoal Faria. An approach for automated scenario-based testing of distributed and heterogeneous systems. In *Proceedings of the 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–10. IEEE, 2015.
77. Shahar Maoz. Model-based traces. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, volume 5421 of *Lecture Notes in Computer Science*, pages 109–119. springer, 2009.
78. Shahar Maoz. Using model-based traces as runtime models. *IEEE Computer Society*, 42:28–36, 2009.
79. Shahar Maoz and David Harel. On tracing reactive systems. *Software and Systems Modeling*, 10(4):447–468, 2011.
80. Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: semantic differencing for activity diagrams. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 179–189. ACM, 2011.
81. Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing semantic model differences. *arXiv preprint arXiv:1409.2307*, 2014.
82. Tanja Mayerhofer. Testing and debugging UML models based on fUML. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1579–1582. IEEE, 2012.
83. Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A runtime model for fUML. In *Proceedings of the 7th Workshop on Models@ run. time*, pages 53–58. ACM, 2012.
84. Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *Proceedings of the International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2013.
85. Katharina Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. *Software Visualization*, 2269:163–175, 2002.
86. Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-specific Property Languages. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, volume 8706 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.
87. Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A framework for testing UML activities based on fUML. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, volume 1069, pages 1–10. Springer, 2013.
88. Stefan Mijatov, Tanja Mayerhofer, Philip Langer, and Gerti Kappel. Testing functional requirements in UML activity diagrams. In *International Conference on Tests and Proofs*, volume 9154 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 2015.
89. Phu H Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. An Extensive Systematic Review on the Model-Driven Development of Secure Systems. *Information and Software Technology*, 68:62–81, 2015.
90. Object Management Group. Business Process Model and Notation (BPMN), Version 2.0, January 2011.

91. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.3, July 2017.
92. Olivier Pasquier and Jean Paul Calvez. An object-based executable model for simulation of real-time Hw/Sw systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 782–783. IEEE, 1999.
93. Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.
94. Carl Adam Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress*, pages 386–390. North Holland, 1962.
95. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.
96. Ivn Santiago, lvaro Jimnez, Juan Manuel Vara, Valeria De Castro, Vernica A. Bollati, and Esperanza Marcos. Model-Driven Engineering as a new landscape for traceability management: A systematic literature review. *Information and Software Technology*, 54(12):1340–1356, 2012.
97. Stefano Schivo, Buğra M Yildiz, Enno Ruijters, Christopher Gerking, Rajesh Kumar, Stefan Dziwok, Arend Rensink, and Mariëlle" Stoelinga. How to Efficiently Build a Front-End Tool for UPPAAL: A Model-Driven Approach. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, volume 10606 of *Lecture Notes in Computer Science*, pages 319–336. Springer, 2017.
98. Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
99. Scopus. *A Generic Framework for Realizing Semantic Model Differencing Operators*, volume 1258, 2014.
100. Dana Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group, 1970.
101. Michael Szvetits and Uwe Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software and Systems Modeling*, 15(1):31–69, 2013.
102. Jérémie Tatibouet, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, volume 8767 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2014.
103. Linzhang Wang, Eric Wong, and Dianxiang Xu. A Threat Model Driven Approach for Security Testing. In *Proceedings of the 3th International Workshop on Software Engineering for Secure Systems, ICSE Workshops*, pages 10–17. IEEE, 2007.
104. Marco A Wehrmeister, Joao G Packer, and Luis M Ceron. Framework to simulate the behavior of embedded real-time systems specified in UML models. In *Brazilian Symposium on Computing System Engineering (SBESC)*, pages 1–7. IEEE, 2011.
105. Marco A Wehrmeister, Joao G Packer, Luis M Ceron, and Carlos E Pereira. Towards Early Verification of UML Models for Embedded and Real-Time Systems. *Embedded Systems, Computational Intelligence and Telematics in Control*, 45(4):25–30, 2012.
106. Levent Yilmaz. Automated object-flow testing of dynamic process interaction models. In *Proceedings of the Simulation Conference, Proceedings of the Winter*, volume 1, pages 586–594. IEEE, 2001.

**Table 2** Classification of model execution tracing approaches for Q1-Q3

| Approach | Types of Models (Q1) | | | | Semantics Definition Technique (Q2) | | | | Trace Data (Q3) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Any | UML models | Workflow models | Other | Translational | Operational | Denotational | Unknown | Event | State | Parameter |
| A01 [92] | | | | * | * | | | | * | | |
| A02 [106] | | | * | | * | | | | | * | |
| A03 [63, 75] | | * | | | * | | | | * | * | |
| A04 [103] | | * | | | * | | | | * | | |
| A05 [40, 41, 42, 43] | | * | | | | * | | | * | * | |
| A06 [26] | | * | | | | * | | | * | * | |
| A07 [20, 21, 22] | * | | | | * | * | | | * | | |
| A08 [23] | * | | | | | * | | | * | | |
| A09 [77, 78, 79] | * | | | | * | | | | * | * | * |
| A10 [67, 68] | | | | * | | | | * | * | * | |
| A11 [46] | | * | | | * | | | | * | | |
| A12 [49, 50] | | | | * | * | | | | * | * | * |
| A13 [56] | * | | | | * | | | | * | | |
| A14 [57] | * | | | | | * | | | * | | |
| A15 [31] | | * | | | * | | | | * | | |
| A16 [29, 30, 44] | | | | * | | * | | | * | | |
| A17 [104, 105] | | * | | | | * | | | * | | |
| A18 [71] | | | | * | | * | | | * | * | |
| A19 [35] | | * | | | * | | | | * | * | |
| A20 [82, 83, 84, 87, 88] | | * | | | | * | | | * | * | * |
| A21 [80, 81] | | * | | | * | | | | * | * | * |
| A22 [4, 5, 6, 7] | | * | | | * | | | | * | * | |
| A23 [9, 10, 12, 13, 14] | * | | | | | * | | | * | * | |
| A24 [60, 61] | * | | | | | | | * | * | * | |
| A25 [47] | * | | | | | * | | | * | * | |
| A26 [37, 38] | | | * | | | | * | | * | | |
| A27 [32, 86] | * | | | | | * | | | * | * | |
| A28 [74] | | | | * | | * | | | * | * | |
| A29 [73, 99] | * | | | | | * | | | * | * | |
| A30 [64] | | | | * | * | | | | * | | |
| A31 [58, 59] | | | * | | * | | | | * | | |
| A32 [36, 76] | | * | | | * | | | | * | | |
| A33 [97] | * | | | | * | | | | * | * | |

**Table 3** Classification of model execution tracing approaches for Q4-Q5

| Approach | Purpose (Q4) | | | | | | Data Extraction Technique (Q5) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Debugging | Testing | Manual analysis | Dynamic analysis | Model checking | Semantic differencing | Source instrumentation | Target instrumentation | Interpreter | External tool | Other |
| A01 [92] | | | | * | | | | * | | | |
| A02 [106] | | * | | | | | * | | | | |
| A03 [63, 75] | | | * | | * | | | * | | | |
| A04 [103] | | * | | | | | | * | | | |
| A05 [40, 41, 42, 43] | * | | | | | | | | * | | |
| A06 [26] | | | * | * | | | | | * | | |
| A07 [20, 21, 22] | | | * | * | | | | | * | * | |
| A08 [23] | | | | | * | | | | * | | |
| A09 [77, 78, 79] | | | * | * | | | | * | | | |
| A10 [67, 68] | | | * | * | | | | | | * | |
| A11 [46] | | * | * | | | | | * | | | |
| A12 [49, 50] | * | | | | | | | | | | * |
| A13 [56] | | | | | * | | | | | * | |
| A14 [57] | | | | | * | | | | * | | |
| A15 [31] | | | * | | | | | * | | | |
| A16 [29, 30, 44] | | | | * | | | | | * | | |
| A17 [104, 105] | | * | | | | | | | * | | |
| A18 [71] | * | | | | | | | | * | | |
| A19 [35] | | | | * | | | | * | | | |
| A20 [82, 83, 84, 87, 88] | * | * | | * | | | | | * | | |
| A21 [80, 81] | | | | | | * | | | | * | |
| A22 [4, 5, 6, 7] | | * | | | | | | * | | | |
| A23 [9, 10, 12, 13, 14] | * | | | | | * | | | * | | |
| A24 [60, 61] | | * | | | * | | | | | * | |
| A25 [47] | | * | | | | | | | | * | |
| A26 [37, 38] | | | | * | | | | | | | * |
| A27 [32, 86] | | | | | * | | | | | * | |
| A28 [74] | | * | | | | | | | * | | |
| A29 [73, 99] | | | | | | * | | | * | | |
| A30 [64] | | * | | | | | | | | * | |
| A31 [58, 59] | | | * | * | | | | * | | | |
| A32 [36, 76] | | * | | | | | | * | | | |
| A33 [97] | | | | | * | | | | | * | |

**Table 4** Classification of model execution tracing approaches for Q6-Q10

| Approach | Trace Representation Format (Q6) | | | | Trace Representation Method (Q7) | | | | | Language Specificity (Q8) | | | Data Carrier Format (Q9) | | | | Maturity (Q10) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Metamodel | Text format | Other | Unknown | FR | AG | MD | AE | Unknown | Language-independent | Language-specific | Specific kind of language | Text | XML | Database | Unknown | |
| A01 [92] | | | | * | | | | | * | | | * | | | | * | 3 |
| A02 [106] | | | | * | | | | | * | | | * | | | | * | 2 |
| A03 [63, 75] | | * | | | | | | | * | | | * | * | | | | 3 |
| A04 [103] | | | | * | | | * | | | | | * | | | | * | 1 |
| A05 [40, 41, 42, 43] | | | | * | | | | | * | | * | | | | | * | 4 |
| A06 [26] | | * | | | | | * | | | | * | | * | | | | 3 |
| A07 [20, 21, 22] | * | | | | * | | | | | * | | | | * | | | 3 |
| A08 [23] | * | | | | | | * | | | * | | | | * | | | 3 |
| A09 [77, 78, 79] | | * | | | | | * | | | * | | | * | | | | 3 |
| A10 [67, 68] | | | * | | | | * | | | | | * | | * | | | 3 |
| A11 [46] | | | | * | | | | | * | | | * | | | | * | 2 |
| A12 [49, 50] | * | | | | | | * | | | | * | | | | | * | 3 |
| A13 [56] | * | | | | * | | | | | * | | | | | | * | 3 |
| A14 [57] | * | | | | * | | | | | * | | | * | | | | 3 |
| A15 [31] | | | | * | | | * | | | | * | | * | | | | 3 |
| A16 [29, 30, 44] | | * | | | | | | * | | | * | | | | | * | 3 |
| A17 [104, 105] | * | | | | | | * | | | | * | | | * | | | 3 |
| A18 [71] | | | | * | | | | | * | | * | | | | | * | 1 |
| A19 [35] | * | | | | | * | | | | | * | | | | * | | 2 |
| A20 [82, 83, 84, 87, 88] | * | | | | | | * | | | | * | | | * | | | 3 |
| A21 [80, 81] | | | | * | | | | | * | | * | | | | | * | 3 |
| A22 [4, 5, 6, 7] | | * | | | * | | | | | | * | | * | | | | 3 |
| A23 [9, 10, 12, 13, 14] | * | | | | | * | | | | * | | | | * | | | 3 |
| A24 [60, 61] | * | | | | | * | | | | * | | | | | | * | 3 |
| A25 [47] | * | | | | | * | | | | * | | | | | | * | 2 |
| A26 [37, 38] | | * | | | | | * | | | | | * | * | | | | 3 |
| A27 [32, 86] | * | | | | | * | | | | * | | | | | | * | 3 |
| A28 [74] | | | | * | | | | * | | | * | | | | | * | 3 |
| A29 [73, 99] | * | | | | | | | * | | * | | | | * | | | 3 |
| A30 [64] | * | | | | | | | * | | | * | | | | | * | 3 |
| A31 [58, 59] | | | * | | | | * | | | | | * | * | | | | 4 |
| A32 [36, 76] | | | | * | | | | | * | | | * | | | | * | 3 |
| A33 [97] | * | | | | | | * | | | | | * | | * | | | 4 |