A Systematic Literature Review on Counterexample Explanation

Arut Prakash Kaleeswaran[a,b], Arne Nordmann[a], Thomas Vogel[b], Lars Grunske[b]

[a] *Bosch Corporate Sector Research,* Renningen, Germany
{*arutprakash.kaleeswaran*|*arne.nordmann*}*@de.bosch.com*
[b] *Humboldt-Universität zu Berlin,* Berlin, Germany
{*thomas.vogel*|*grunske*}*@informatik.hu-berlin.de*

Abstract

Context: Safety is of paramount importance for cyber-physical systems in domains such as automotive, robotics, and avionics. Formal methods such as model checking are one way to ensure the safety of cyber-physical systems. However, adoption of formal methods in industry is hindered by usability issues, particularly the difficulty of understanding model checking results. **Objective:** We want to provide an overview of the state of the art for counterexample explanation by investigating the contexts, techniques, and evaluation of research approaches in this field. This overview shall provide an understanding of current and guide future research. **Method:** To provide this overview, we conducted a systematic literature review. The survey comprises 116 publications that address counterexample explanations for model checking. **Results:** Most primary studies provide counterexample explanations graphically or as traces, minimize counterexamples to reduce complexity, localize errors in the models expressed in the input formats of model checkers, support linear temporal logic or computation tree logic specifications, and use model checkers of the Smybolic Model Verifier family. Several studies evaluate their approaches in safety-critical domains with industrial applications. **Conclusion:** We notably see a lack of research on counterexample explanation that targets probabilistic and real-time systems, leverages the explanations to domain-specific models, and evaluates approaches in user studies. We conclude by discussing the adequacy of different types of explanations for users with varying domain and formal methods expertise, showing the need to support laypersons in understanding model checking results to increase adoption of formal methods in industry.

Keywords: formal methods, model checking, counterexample explanation

1. Introduction

"*It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature.* – Edmund Clarke [1]"

Accepted Manuscript. Information and Software Technology *December 04, 2021*
https://doi.org/10.1016/j.infsof.2021.106800

The complexity of modern, software-intensive systems continues to increase due to the rising number of features and functionalities. When complex software-intensive systems are used in safety-critical domains such as automotive, robotics, and avionics, their malfunction might lead to severe damages or even loss of lives. Consequently, safety of these systems is of paramount importance. To ensure safety, these systems have to be developed according to standards such as IEC 61508, and ISO 26262 in the automotive domain. These standards require safety methods such as Failure Mode and Effect Analysis (FMEA), Fault Tree Analysis (FTA), or Hazard and Operability (HAZOP). Still today, such safety analysis is often performed manually by engineers to identify potential safety flaws. Machine support to automate the analysis process is a way to overcome this time-consuming, expensive, and error-prone process.

Model checking [2–4] is a computer-assisted verification method for systems that were modeled in a formal way by state-transition systems. Drawing from research in mathematical logic, programming languages, hardware design, and theoretical computer science, model checking is now widely used for the verification of hardware and software in industry [4].

Model checking verifies whether a requirement is satisfied by a system or not. For this purpose, it requires a formal specification of the requirement, typically as a *temporal-logic formula* φ, and a formal description of the system, for instance, as a *Kripke structure* K. Then, a *model checker* as a tool performs the verification by checking whether the specification φ is satisfied by the system model K, that is, $K \vDash \varphi$. The result of the model checking is either that φ is satisfied by K (*i. e.*, $K \vDash \varphi$), or that φ is not satisfied by K (*i. e.*, $K \nvDash \varphi$). In the latter case, the model checker returns a *counterexample* to φ on K. Such a counterexample describes an execution path over system states that leads from the initial state to a state that violates φ. Each state of such a path consists of atomic propositions (AP) over the variables defined by K.

Once the system and requirements are formalized, model checking is attractive as it is an automated method and offers counterexamples if a system model fails to satisfy a requirement, serving as indispensable debugging information [3]. However, counterexamples are only the symptoms of faults and understanding a counterexample to identify a fault in the system model is a complicated task for several reasons: (**R1**) a counterexample is often cryptic and lengthy [5], (**R2**) not all the states in a counterexample are relevant to an error [6], (**R3**) not all the variables in a state have any relation to the violated specification [6], (**R4**) the debugging task is performed manually, which is time-consuming and error-prone [5, 7–9], and (**R5**) the counterexample does not explicitly highlight the source of the error that is hidden in the model [7]. These challenges call for a method to explain counterexamples, assisting system designers in localizing faults in their models [10].

Therefore, we provide an overview of the state of the art in research on explaining counterexamples and how system engineers are supported in interpreting counterexamples. This allows us to assess the state of the art and identify needs for future work. For this purpose, we conducted a system literature review on counterexample explanation with the main focus on (**1**) the different kinds to explain a counterexample, (**2**) the different methods to transform or optimize a counterexample in order to provide an explanation, (**3**) influences of the input system and requirement on counterexample explanation, and (**4**) the different domains and applications to evaluate approaches to counterexample explanation.

By collecting and analyzing literature for these aspects, our survey provides a comprehensive overview of counterexample explanation. To structure and guide our survey, introduce the conceptual model with its terminology that we use in our survey, depicted in Figure 1.

1.1. Conceptual Model and Terminology

The conceptual model shown in Figure 1 consists of two parts, *model checking* and *counterexample explanation*, that we discuss in the following.

Model Checking. In our survey, we use the term *design model* to refer to an informal or formal description of the system that can be expressed in any mod-

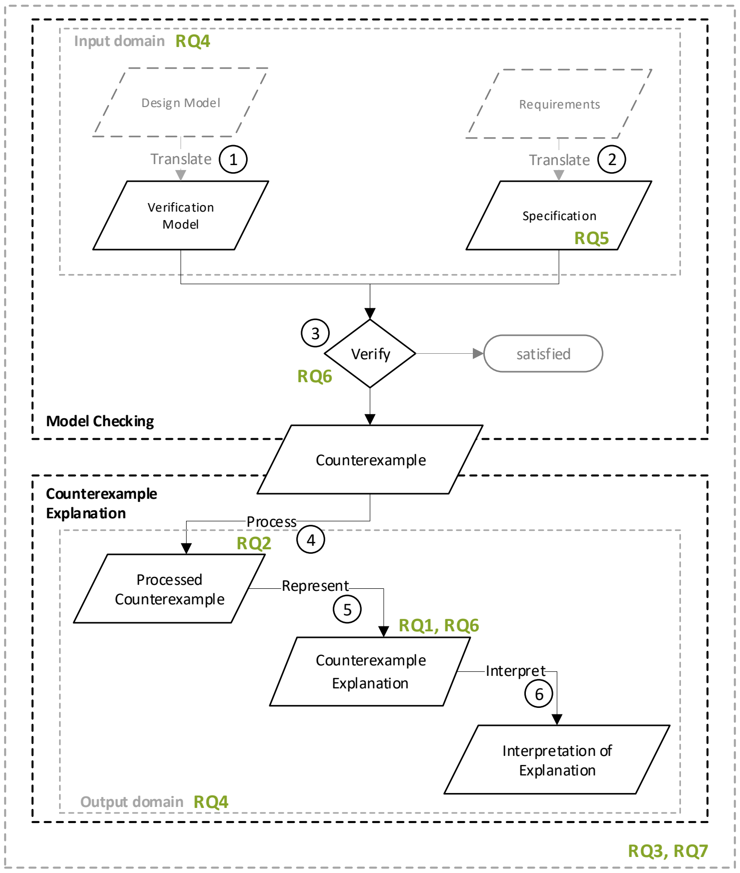


Figure 1: Overview of model checking and counterexample explanation.

eling language such as the Systems Modeling Language (SysML) and Unified Modeling Language (UML). To perform model checking, the design model has to be translated to a *verification model* that is expressed in a specific formalism required by the used model checker (`Step` ① in Figure 1). For instance, the model checker New Symbolic Model Verifier (NuSMV) [11, 12] uses the formalism of a Kripke structure to describe a system.

As a system specification we generally consider technical and system *requirements* that are usually defined informally. To perform model checking, requirements have to be formally specified by translating them into a *specification* expressed in a temporal logic (`Step` ②). For instance, NuSMV supports, among others, Linear Temporal Logic (LTL) [13] and Computation Tree Logic (CTL) [14]. To ease this formalization step, property specification patterns have been proposed [15, 16].

Given a verification model and specification, a model checker *verifies* whether the model satisfies the specification (`Step` ③). The result of the verification is either the fact that the model *satisfies* the specification or a *counterexample* showing a sequence of state transitions of the model that violates the specification: the counterexample. Such a counterexample is produced in a specific format depending on the model checker. For instance, a counterexample created by NuSMV follows the Kripke structure of the verification model.

Counterexample Explanation. The goal of counterexample explanation is to support engineers in interpreting a counterexample and comprehending the violation and particularly, the error in the system design.

The input to the counterexample explanation process is the *counterexample* returned by a model checker and optionally the artifacts of what we call the *input domain* comprising the *verification model*, *specification*, *design model*, and *requirements* of the system. First, the *counterexample* is *processed* to yield a *processed counterexample* that may be easier to interpret (`Step` ④). For instance, it can be a minimized version of the original counterexample reduced to the parts that are relevant for the violation and error. Afterwards, a *representation* of the processed counterexample is created that serves as the *explanation* of the counterexample (`Step` ⑤). The explanation may relate to artifacts of the input domain, which have been provided by the system engineer. Finally, the explanation is presented to a system engineer who *interprets* it, usually manually, to comprehend the error (`Step` ⑥).

2. Related Work

In the following, we discuss surveys related to our work, categorized into two groups depending on whether they are focused on verification for specific application domains or model checking and its algorithms.

Surveys focused on verification for specific application domains. Karna *et al.* [17] present a survey about model checking and various tools or techniques used in software engineering development. An overview of formal verification on real-time systems is given by Wang [18] and on model-based engineering by Gabmeyer *et al.* [19]. Both surveys discuss models, specification languages, verification frameworks, and the state-space construction and representation in their respective domains. More specifically, Ovatman *et al.* [20] discuss the difficulties of the model checking process, tools and specification language on Programmable Logic Controller (PLC) software production. The survey by Grimm *et al.* [21] provides an overview of the most widely used formal verification techniques and discusses their value for critical Systems-on-Chip verification.

Surveys focused on model checking techniques and its algorithms. The surveys by Clarke *et al.* [22, 23] present contributions on symbolic model checking with its algorithms, abstraction, and software verification. The survey by Prasad *et al.* [24] investigates Satisfiability (SAT)-based model checking, bounded model checking, and problems such as application-specific heuristics and conflict-driven learning. Another survey focusing on SAT-based model checking but together with hardware benchmarks is given by Amla *et al.* [25] for eight bounded and unbounded techniques. The survey by D'Silva *et al.* [26] focuses on automatic static analysis of software for three techniques, namely static analysis with abstract domains, model checking, and bounded model checking. Finally, there are surveys discussing the problem of state space explosion in model checking. For instance, Pelanek [27] survey the techniques to address the state space explosion. They categorize such techniques to state space reductions, storage size reductions, parallel and distributed computation, randomization and heuristics. The survey by Edelkamp *et al.* [28] discusses algorithms for directed model checking and mitigating the state explosion problem.

Summary. Given the existing surveys in these two categories, none of them addresses the research problem of explaining counterexamples generated by a model checker. Therefore, to the best of our knowledge, this systematic literature review is the first one that investigates specifically the state of the art in explaining counterexamples in order to support debugging and localizing faults.

3. Research Methodology

For our survey, we followed guidelines from standard practice in systematic literature reviews by Kitchenham and Charters [29] complemented by guidelines on snowball sampling by Wohlin *et al.* [30]. Particularly, we follow these major steps to conduct a systematic literature review [29]: (**1**) define the need for the review, (**2**) define research questions, (**3**) identify primary studies, (**4**) perform data extraction, (**5**) perform data synthesis, and (**6**) study quality assessment. Concerning the first step, we have motivated the need of our survey in the introduction and the necessity of this study based on related work (Section 2).

3.1. Research Questions

We argue that counterexample explanation is key to a broader adoption of model checking because it allows the usage of model checkers without necessarily having to understand their internal intricacies. Moreover, it promises to make the model checking result actionable for engineers by supporting them to localize and understand the fault in the system design. The main motive of this survey is therefore to address the general question: *What is the state of the art for explaining counterexamples generated by model checkers?*

To answer the general question, we refined it into eight specific research questions (RQs). Figure 1 positions these research questions in our conceptual overview of model checking and counterexample explanation.

RQ1: *How are counterexamples explained and what are the effects of this explanation on counterexample interpretation?* With this question, we target the explanation of counterexamples presented to engineers, their style of representation, and their impact on the interpretation of counterexamples.

RQ2: *How are counterexamples processed and what effect does it have on interpreting the counterexample?* Answering this RQ provides insights into the effects of processed counterexamples for their interpretation.

RQ3: *What kind of additional information is used to enrich the counterexample explanation?* This RQ investigates additional information that is provided together with the counterexample explanation to improve debugging and locating the error.

RQ4: *Which input domains are targeted by counterexample explanation approaches? What is the influence of the input domain on the explanation?* This RQ explores the relation between the input domain and output domains of counterexample representation, including the influence of the input domain on the output domain.

RQ5: *What are the different temporal logics used to express system specifications and what type of properties are covered in counterexample explanation approaches?* With this RQ, we enumerate and count the temporal logic formalisms being used to specify properties in counterexample explanation approaches. Moreover, we look at the types of properties supported by these approaches, *e. g.*, safety and liveness.

RQ6: *Which verification tools and frameworks are developed and used to explain counterexamples, and how do they effect the counterexample explanation?* This RQ lists different verification tools and frameworks and further describes their impact on representing and explaining counterexamples.

RQ7: *How are counterexample explanation approaches evaluated?* This RQ provides insights into the evaluation of counterexample explanation approaches, *i. e.*, the evaluated aspects, evaluation methods, domains, and applications.

Considering Figure 1 that positions all of the research questions in a conceptual overview of model checking and counterexample explanation, we note that the

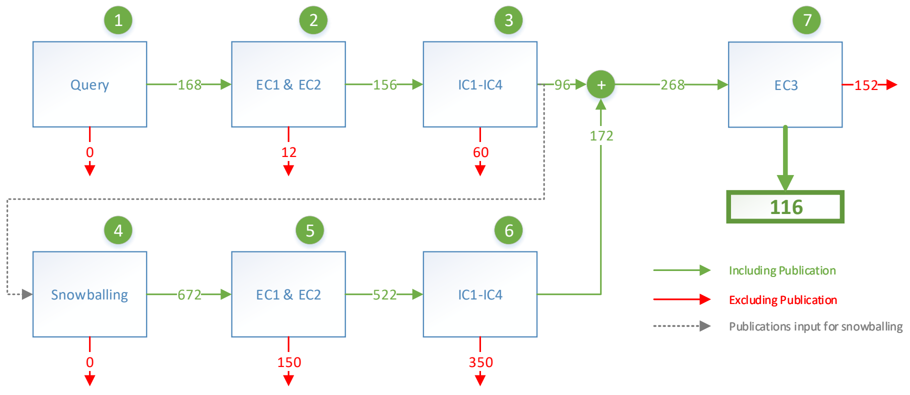


Figure 2: Process to identify the primary studies for our survey.

research questions tackle all relevant elements. Therefore, we are confident that our survey provides a comprehensive overview of the state of the art in counterexample explanation.

3.2. Selection of Primary Studies

We select primary studies in two phases. First, we collect the initial set of primary studies by a keyword-based search and filter them by applying the inclusion and exclusion criteria. Second, we perform snowballing with the initial set and again filter the newly identified primary studies using defined inclusion and exclusion criteria. The selection process of primary studies is shown in Figure 2.

3.2.1. Keyword-Based Search

Using keyword-based search, we identify a start set for the subsequent snowballing. We use *Google Scholar* with the search query shown in Listing 1 and limit the results to publications published between 2000 and 2021. We performed the query on February 11th, 2021.

Listing 1: Query for the keyword-based search.

```
(Abstraction|Annotated|Graphical|Localization|Explanation|Guided|
    Interpretation|Lifting|Minimum|Refinement|Pattern|Highlight|
    Reduced|safety|Liveness|Reachability|CNL|DSL|Causality|
    Ontology)(Counterexample|Model Checker|Temporal Logic)
```

We derive the keywords of the query from the survey's focus: explanation and interpretation of counterexamples. Then *2dSearch*[1] generates the combination of the keywords to a query that allows to combine the keywords visually and

[1]`https://app.2dsearch.com/query`

check whether Google Scholar supports specific combinations. We further use the tool *Publish or Perish*[2] that performs the search and automatically extracts the search results from Google Scholar. The query yields **168** primary studies, see `Step` ① in Figure 2.

3.2.2. Applying Inclusion and Exclusion Criteria

All publications resulting from the query are analyzed manually based on the following inclusion and exclusion criteria. This analysis is conducted with a four-eyes principle where one author assesses the analysis of another author.

A primary study is included in our study if it satisfies at least one of the following inclusion criteria (**IC**).

(**IC1**) The primary study describes an approach to represent counterexamples either in a graphical or prose manner.

(**IC2**) The primary study describes an approach that modifies the length of a counterexample to ease counterexample explanation or interpretation.

(**IC3**) The primary study describes a framework to represent counterexamples.

(**IC4**) The primary study describes the problem statements or challenges of explaining or interpreting counterexamples.

Further, we exclude any primary study if it satisfies at least one of the following exclusion criteria **(EC)**.

(**EC1**) The primary study is a book, a PhD thesis, or a patent.

(**EC2**) The primary study is not accessible online from the Bosch or Humboldt University network.

In general, we first apply the exclusion and then the inclusion criteria on the publication's title, abstract, introduction, and conclusion.

Applying the two EC on the **168** primary studies obtained by the query (`Step` ② in Figure 2) filters out **12** studies, leaving **156** remaining studies. Based on the IC, we exclude **60** primary studies and include **96** studies (`Step` ③). These **96** studies are the input for snowballing.

3.2.3. Snowballing

In the second phase, we perform snowballing with the primary studies that we identified in the first phase. Particularly, we use Google Scholar to perform forward (identifying citing publications) and a backward (identifying cited publications based on the publication's references and related articles suggested by Google Scholar) snowball sampling. Then, exclusion and inclusion criteria are applied to the newly discovered primary studies as well. Snowballing is performed by the first author of this paper and the other authors verify it by randomly selecting primary studies and verifying the snowball sampling.

[2]`https://harzing.com/resources/publish-or-perish`

We collect additional **672** primary studies during snowballing with the **96** primary studies of the first phase (`Step` ④ in Figure 2). Applying the two EC on these **672** studies (`Step` ⑤) excludes **150** studies. Applying the four ICs to the remaining **522** studies results in **172** included studies and **350** excluded studies (`Step` ⑥).

3.2.4. Selected Primary Studies

The **96** primary studies obtained by the keyword-based query and the **172** primary studies obtained by snowballing result in a total of **268** primary studies. Among these studies, there are approaches that focus on model abstraction and bounded model checking techniques (*cf.* IC2). However, they are not primarily concerned with counterexample explanation and interpretation, and therefore excluded by adding a third exclusion criterion (**EC3**): *The primary study focuses mainly on model abstraction and bounded model checking techniques.* Applying **EC3** to the **255** studies excludes **152** primary studies and includes **116** primary studies (`Step` ⑦). Overall, we select a corpus of **116** primary studies for our survey.

3.2.5. Data Extraction

The primary studies collected during the two phases are maintained in a shared spreadsheet to provide accessibility to all authors. To answer the research questions, we extract 20 data items from the primary studies.[3] The extracted data is also maintained in the spreadsheet.

When answering the research questions, we present quantitative analysis results wherever reasonable and provide additional qualitative results and highlight exemplary publications.

4. Results

This section provides the results of our study and answers the research questions introduced in Section 3.1. For quantitative RQs, we collect the answers and provide statistical data. For qualitative RQs, we extract relevant, interesting, or representative statements from the primary studies. The primary studies used for each research question are tabulated in [31].

4.1. ***RQ1:*** *How are counterexamples explained and what are the effects of this explanation on counterexample interpretation?*

With this research question we investigate the counterexample explanations presented to engineers, particularly the style of representing a counterexample and what effects these have on counterexample interpretation. The surveyed primary studies use five different styles of representation to explain counterexamples (Figure 3a): 54 primary studies (47% of all primary studies) use a graphical,

[3]The data items and their descriptions are listed in the appendix of this paper [31, Table 1].

35 (30%) a trace, 17 (15%) a textual, 5 (4%) a tabular representation, while the remaining 5 (4%) use a combination of graphical and tabular representations.[4] In the following, we discuss each category with examples from the primary studies. We further provide qualitative statements from these studies to show the effects of counterexample explanation on the interpretation.

4.1.1. Graphical representation

*Graphical representation*s visualize a counterexample in a graphical notation, *e. g.*, as a state machine or block diagram. Nguyen and Ogata [32] state that "a graphical representation of a counterexample would help human users comprehend it more intuitively". Accordingly, we found 54 primary studies (47% of all primary studies) that use graphical explanations. In addition, graphical *animations* promise to further ease debugging of a counterexample [32–34].

UML or SysML graphical diagrams such as component, state-machine and sequence diagrams are used to explain counterexamples in 27 primary studies (50% of the primary studies that use a graphical representation). Elamkulam *et al.* [35] map counterexamples to states and events in the design model. These are presented as a UML *sequence diagram*, making it easy for the user to understand the design flaw. In addition, animation of the counterexample in graphical diagrams like *function, block* [36–39], *component* [40], or *state-machine diagram* [41, 42] enables the user to simulate the counterexample step-by-step in the user-given input models to identify the error behavior.

Fault trees are significantly smaller and easier to understand than corresponding stochastic counterexamples. A fault tree still contains all the information required to recognize the cause for the occurrence of a hazard [43]. Fault trees are used by 7 primary studies to explain counterexamples (13%). For instance, the fault tree generation approaches by Leitner-Fischer and Leue [44–46] compute causal events by performing causality checks for probabilistic models. Notably, Leitner-Fischer and Leue [44] claim from a case study that a "fault tree is a compact and concise visualization of the counterexample, which allows for easy identification of basic events that cause the hazard". Similarly, Nguyen and Ogata [32] argue that "*it would be easier to comprehend a shorter counterexample, but the text representation of a counterexample need not be necessarily understandable. A graphical representation of a counterexample would help human users to comprehend it more intuitively*".

4.1.2. Trace representation

A *trace representation* is a modified form of a counterexample with addition or removal of (sub)-traces. Traces are the second most used type of representation in the primary studies (30% of all primary studies). Traces can be used to represent a witness (correct trace) [47], a minimized counterexample [48], or multiple counterexamples [49]. A trace representation aims to lead to easier

[4] The primary studies for each style of representation are listed in [31, Table 2].

debugging and faster error comprehension. It is true for a minimized counterexample, the shorter the counterexample the simpler the debugging process [50]. Further details of trace representations are discussed in Section 4.2.

4.1.3. Textual representation

Textual representations can be understood easily even by non-experts [6] in formal methods. However, only 17 of the primary studies (15% of all primary studies) use such a representation to provide statements for error notification or highlighting the error in a provided textual input system such as Controlled/-Constrained Natural Language (CNL) [51, 52], Structured English [53–57], or error localization as known from programming languages [58–62].

Feng *et al.* [53] and Luteberget and Johansen [54] generate a minimal set of structured language sentences. These *natural language like* sentences have the potential advantage of providing diagnostic feedback to humans. Berg *et al.* [6] investigate two approaches for interpreting counterexamples: animation and natural language. Animation is a highly effective technique. However, animating and interpreting the railway interlocking and train behavior from the counterexamples is not always intuitive. For instance, "the animation would show two trains using the same route and occupying the same segment at the same time. The user may incorrectly think there is an error as soon as this happens, however this is not the case." For these reasons, authors deemed animation unsuitable for this use case and chose natural language for presentation instead.

4.1.4. Tabular representation in combination with graphical representations

A *tabular representation* is the least used type of counterexample representation by the primary studies (4% of all primary studies). The *combination of tabular and graphical representations* is used by additional five studies (4% of all primary studies) to represent counterexamples. Tabular representations convert the variables and values of a counterexample into rows and columns without altering its original information.

Frameworks like FormAl SpecificaTion ENvironment (FASTEN) [40] and AutoFocus 3 [63] use tabular views to enhance readability. These frameworks transform a counterexample generated by NuSMV into rows and columns, where the columns are the variables and rows represent the states. Arcaini *et al.* [64] as well as Bolton and Bass [65] highlight the variable and its values in the table to show changes from the respective previous state. Arcaini *et al.* [64] state that "such representation is much more readable than the standard counterexample representation of NuSMV in which only the variables that changed their value are shown". However, according to Loer *et al.* [66], tables are primarily suitable for mechanical analysis, but hard to read by human analysts.

4.1.5. Answer to RQ1

The results of our study clearly show that graphical representations are used predominantly for the explanation of counterexamples. A format close to graphical representations, traces, is the second most used representation type, together representing 77% of the surveyed primary studies. Graphical representations are often based on component, state-machine and sequence diagrams, *e. g.*, from UML or SysML, as we observed for 50% of the primary studies that use graphical representations. Concerning the effects of the explanation on the interpretation, primary studies prefer graphical or textual (natural language-like) explanations since they are easier to interpret than the counterexample or respectively an animated counterexample [6, 32].

4.2. ***RQ2:*** *How are counterexamples processed and what effect does it have on interpreting the counterexample?*

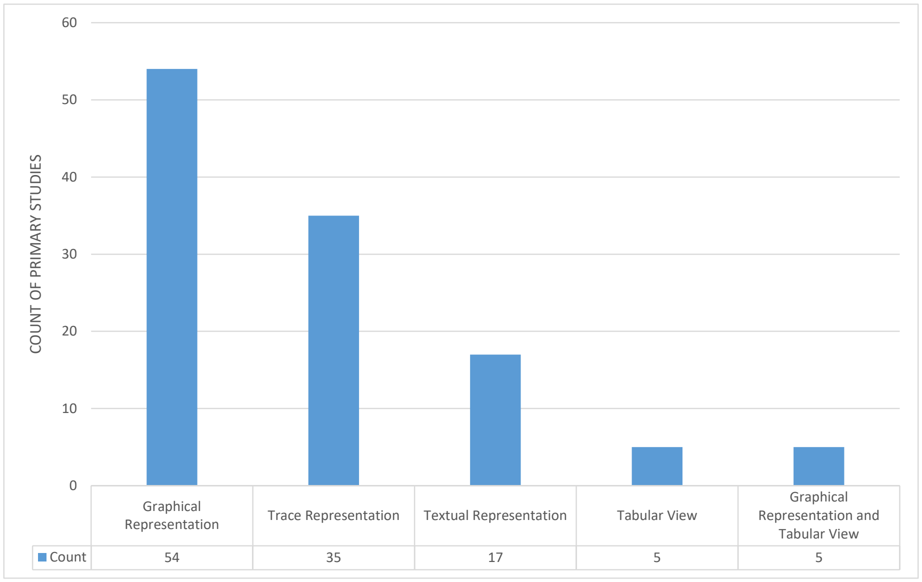
We investigate how counterexamples returned by a model checker are further processed to make them more valuable to the user. Out of the 116 primary studies, 54 (47%) further process the counterexample while the remaining 62 studies (53%) skip the processing and focus directly on representing the counterexample (*cf.* Figure 1).

In our survey we found three main categories of *processing a counterexample*, see Figure 3b: 38 primary studies (70% of all primary studies that process the counterexample) process a counterexample to a minimized counterexample, 9 (17%) to a witnesses, and 7 (13%) to multiple counterexamples.[5] In the following, we discuss each category with examples from the primary studies. We further provide qualitative statements from these studies to show the effects of counterexample processing on the interpretation. The interpretation is also influenced by the representation of the processed counterexample. Out of the 54 primary studies that process the counterexample, 35 (65%) represent the processed counterexample as a trace, and the remaining 19 primary studies (35%) explain the processed counterexample in a graphical or textual format.

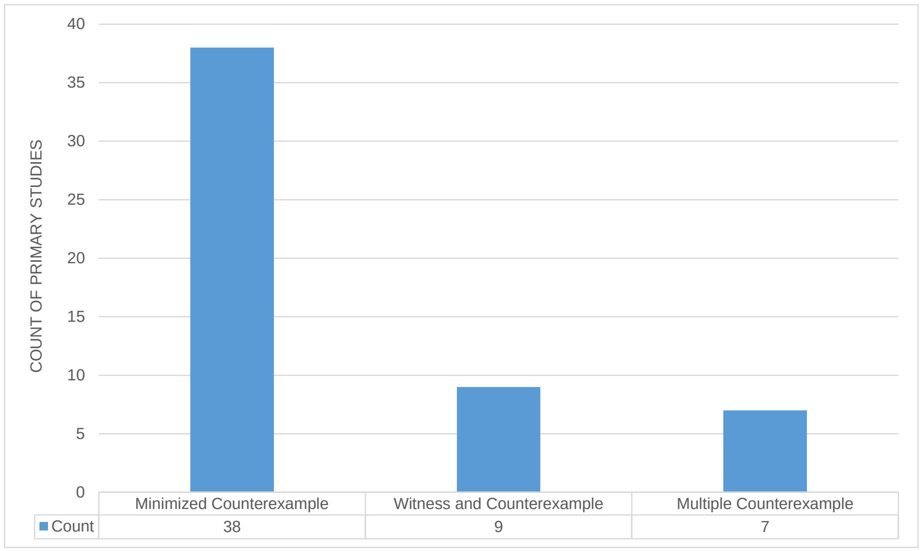
4.2.1. Minimized counterexample

In this survey, we use the term *minimized counterexample* for what is called *shortest counterexample, abstract counterexample, or reduced counterexample* in the surveyed primary studies, or is sometimes called categorizing the error trace, transition, state, or variable in the counterexample. Schuppan and Biere [67] state that “most counterexamples still need to be interpreted by humans, and shorter counterexamples will, in general, be easier to understand”. This is often done by highlighting the erroneous transitions to distinguish them from correct transitions. 38 primary studies (70% of all primary studies that process a counterexample) follow such an approach, and can be grouped into three categories based on the kind of specification and verification model: qualitative, real-time, and probabilistic. Notably, we found only one primary study [68] that uses a real-time specification and minimizes the counterexample by removing variables that are irrelevant for error comprehension. The two other categories are discussed in the following.

[5]The primary studies for each of these categories are listed in [31, Table 3].



(a) **RQ1:** Types of counterexample representation.



(b) **RQ2:** Types of processed counterexamples.

Figure 3: Types of counterexample representation and processed counterexamples.

Counterexample minimization for qualitative specifications. 27 of the 38 primary studies (71%) minimize a counterexample for qualitative systems. From the surveyed primary studies, the most prominent model checkers used to perform counterexample minimization are NuSMV [67, 69–72], Simple PROMELA Interpreter (SPIN) [48, 73–76], and Maude [32]. The first novel approach to minimize loop-like and path-like counterexamples for given ACTL properties is proposed by Shen *et al.* [71], while the existing approaches can only deal with path-like counterexamples of invariants of the form AGf. One of the notable approaches we found in our survey is from Hansen and Geldenhuys [75], where the user can specify the size of the shortest counterexample, which is shown to the user simultaneously during the verification run. This allows the user to interrupt the algorithm when the counterexample shrinks short enough. Moreover, the user can easily specify the limits on the number of states and transitions that the algorithm is allowed to explore or—more directly—the time and memory it is allowed to consume. Brute Force Lifting (BFL) is the most effective counterexample minimization algorithm [69, 77]. It performs refutation analysis of counterexample variables, *i. e.*, to extract the set of variables that are irrelevant to the counterexample. Thus, multiple variables can be eliminated with only one call to the SAT solver.

Counterexample minimization for probabilistic specifications. Ten of the 38 primary studies that minimize counterexamples (26%) specifically address probabilistic systems. For probabilistic model checking, a set of counterexamples is typically required to provide an accumulated probability mass that violates the probability bound of the specified probabilistic property [78]. Thus, to compute the probabilities for all traces, several traces need to be understood and analyzed. Therefore, Leitner-Fischer and Leue [78] propose *Causality Checking* [44] that is integrated into the state-space exploration algorithm for qualitative model checking and that computes causality relationships on-the-fly. Consequently, the probability computation can be limited to the causal events. To apply the causality computation to a Probabilistic Symbolic Model Checker (PRISM) model, the Directed Probabilistic Counterexample Generation Tool (DiPro) is used to generate a counterexample. Further, Debbi and Bourahla [79] propose an algorithm for diagnosis generation along with a set of causes from the counterexample generated by DiPro. It is the first approach that introduces the diagnosis for counterexamples in probabilistic model checking.

Abraham *et al.* [80] generate an abstract counterexample by choosing one or more paths from a counterexample, which carry enough probability. The user can choose the level of abstraction of a counterexample and refine the counterexample step-by-step by choosing abstract nodes and replacing them by the concrete input nodes. Similarly, the Graphical User Interface (GUI) presented by Jansen *et al.* [81] generate concrete and abstract graphs for Discrete-Time Markov Chain (DTMC) that can be stored, loaded, abstracted, and concretized by the user, thus controlling the hierarchy of a counterexample. The interactive visual functions used by Aljazzar and Leue [82] allow users to selectively filter the displayed information, which helps to focus during visual analysis.

Methods for counterexample minimization. Out of 38 primary studies that minimize the counterexample, 15 (39%) follow a specific or adapt an existing algorithm such as BFL [69, 77, 83] to perform the minimization. In the following, we discuss other commonly found methods to minimize counterexamples that are either based on search, translation and abstraction, or comparison of the counterexample with the correct system behavior.

Search. Among the 38 primary studies that minimize counterexamples, 14 (37%) use search methods for this purpose. In general, a search method explores a given system state space in a directed way by representing the system behavior in a graph structure, in which states are nodes and transition are edges. For example, Tan *et al.* [84] reduce the size of the counterexample using a heuristic-guided search strategy. Every node is associated with a value denoting the distance to the non-accepting state that does not have any outgoing edge except of self-loops. During the search, the shortest path is computed based on this distance and returned as a minimized counterexample.

Edelkamp *et al.* [74, 76] as well as Aljazzar and Leue [82] use the directed search algorithm A^* that follows a Depth-First Search (DFS) method to find the shortest counterexample [85]. Edelkamp *et al.* [74, 76] further propose a heuristic that accelerates the search in the direction of a specified failure situation, with an improved nested DFS to identify a safety property violation. Aljazzar and Leue [82] use A^* to find the shortest path in a directed graph. First, the state space of a Markov chain is represented as a directed graph, in which nodes represent states and edges represent transitions. Then, A^* explores all edges of the graph and inserts them into a so-called *path graph.* With the generated path graph, an interactive visualization allows filtering out parts of the state space.

Gastin and Moro [73] state that nested DFS approaches strongly rely on the order of the transitions. Therefore, they propose a *polynomial-time algorithm* based on Breath-First Search (BFS) where the ordering of the transitions has no impact. The proposed algorithm computes a counterexample of minimal size for SPIN that uses forward edges and consumes less memory than SPIN while trying to reduce the size of counterexamples. Similarly, Hansen and Kervinen [86] present an algorithm based on BFS that finds a minimal counterexample. Furthermore, the model checker Maude [32] is equipped with a search command that exhaustively traverses the reachable states in BFS manner to generate the shortest counterexample. Aljazzar and Leue [87] use an eXtended Best-First (XBF) algorithms that is guided by heuristics to amplify the power of search in terms of computational efforts and counterexample quality. The algorithm explores the state transition graph of Continuous-Time Markov Chains (CTMCs) and DTMCs, and incrementally determines a sub-graph of the state transition graph, which covers the most probable diagnostic paths.

Apart from using either a BFS or DFS search method, Leitner-Fischer and Leue [46] integrate DFS and BFS for the causality check. The resulting search algorithm distinguishes between bad and good executions based on which the causality relationships are computed. Similarly, the algorithm by Groce and Visser [88] generates a subset of *negatives*—executions that reach a particular error state—and a set of potential *positives*—any execution that does not end up in the error state. Then, it uses a model checker to explore backward from the original counterexample. The algorithm is depth-first, but it can be integrated into both breadth-first and heuristic-based model checking algorithms.

Translation and abstraction. From the 38 primary studies that minimize counterexamples, 5 studies (13%) use a method based on translation and abstraction of the modeled system or subsystem for this purpose. Frameworks as proposed by Gerking *et al.* [68], as well as FASTEN [40] and AutoFOCUS 3 [89] allow engineers to model a system with Domain-Specific Languages (DSLs). In these frameworks, a DSL supports to minimize the counterexample by performing two types of translation: forward and backward translation. The forward translation adds new states and transitions to the verification model to enable model checking when translating the design model expressed in a DSL to the verification model. The backward translation removes the states and transitions being added by the forward translation, thus reducing the size of the counterexample. Consequently, the complexity of counterexample interpretation is reduced when lifting the counterexample to the DSL-based design models.

Jansen *et al.* [90] compute critical subsystems hierarchically by concretizing abstract states and reducing the concretized parts. Such subsystems reduce the number of involved states and transitions when generating a counterexample. The computation of critical subsystems is based on finding most probable paths or path fragments to be contained in the critical subsystems. Concretization of only the user-relevant parts of the abstract critical subsystems allows for an intuitive approach to error correction. To compute state-minimal critical subsystems for DTMCs and Markov Decision Processess (MDPs), Wimmer *et al.* [91] propose Satisfiability Modulo Theories (SMT) and Mixed Integer Linear Programming (MILP)-based formulations. Wimmer *et al.* [92] further propose a tool that yields smaller subsystems than the available heuristic-based tools, with a lower bound for the model checker to provide a minimal counterexample.

Comparison with correct system behavior. Out of the 38 primary studies that minimize counterexamples, 4 studies (11%) identify erroneous and correct states in the counterexample by comparing the counterexample with the correct system behavior. Barbon *et al.* [7, 93–95] propose an approach to simplify the understanding of counterexamples for a Labelled Transition System (LTS). The approach extracts actions from the counterexample that are relevant for the violation to reduce the amount of debugging information. To do this, it compares the generated counterexamples with the correct behaviors of the verification model, and differentiates the error transitions and correct transitions in the counterexample. Similarly, the approach by Kaleeswaran *et al.* [96] identifies erroneous and correct states in the counterexample. It particularly supports engineers in locating faults in a Contract-Based Design (CBD), a user-provided design model, that does not pass the refinement check.

4.2.2. Witness trace

9 primary studies (17% of all 54 studies that process a counterexample) generate *witness traces* from counterexamples. In addition to counterexamples, which show traces that fail to satisfy specifications, witnesses show traces that can

satisfy the failed specification. The user can compare witness traces with the counterexample to understand the error behavior and propagation. As two prominent approaches, Peled *et al.* [47] generate a deductive proof so that the system meets its LTL specification and Groce *et al.* [61] produce the successful execution that is most similar to the counterexample.

Satisfying a specification, a *witness trace* is the opposite view of the counterexample. Peled *et al.* [47] state that "*proof by lack of counterexample* is the main drawback of the model checking approach; some would even say that model checking is a tool for falsification rather than a tool for verification". Thus, their work generates a deductive proof by using the LTL model checking process and presents it as a graph when the system meets its LTL specification. Therefore, it is possible to justify why the system actually works. Similarly shown by Beyer *et al.* [97], the two-step approach provides evidence to support this claim, *i. e.*, whether it produces a witness for correctness or violation of the specification.

Jin *et al.* [98] present an enhanced error trace as an alternative to fated (forced) and free segments. The fated segments show unavoidable progress towards the error while the free segments show the choices that, when avoided, might have prevented the error. Hence, the demarcation into segments tends to highlight critical events. Similarly, Groce and Visser [88] generates *negatives*—executions that reach a particular error state—and *positives*—any executions not ending in the error state—to find a counterexample by searching close to a path the user suspects could lead to an error.

Leue and Befrouei [10] use SPIN to detect deadlocks in concurrent systems, where a set of counterexamples is generated as a "bad" dataset and an additional "good" dataset is generated that does not violate the specification. By comparing the bad with the good dataset, sequences of actions are extracted to locate the cause of the occurrence of a deadlock. Two different approaches are used by Kumar *et al.* [99] for error explanation. The first approach is to compute the minimal change in a system that eliminates the error causing the counterexample. The second approach computes a correct trace of the system that is close to the counterexample.

4.2.3. Multiple counterexamples

Another technique we found in 7 primary studies (13% of all primary studies that process a counterexample) provides *multiple counterexamples.* This is usually the result of generating all possible counterexamples for every possible failure case, instead of providing just a single counterexample. Standard model checkers like NuSMV do not produce multiple counterexample by default. Copty *et al.* [100] present a wizard that can retrieve multiple counterexamples in a single run and can be used to analyze all possible error conditions for the provided specification. PyNuSMV generates multiple or complete counterexamples for branching logics like CTL by translating specifications into μ-calculus [101].

The method of multiple counterexamples enables finding all possible failures in a single run of verification. It also has the ability to identify more than one root cause. This can reduce the number of verification runs greatly [100].

Dominguez and Day [102] provide an approach that produces all counterexamples automatically by modifying the specification and grouping counterexamples together. Not modifying the model checker engine or the verification model, this approach works with any LTL model checker, and produces the complete set of counterexamples. Ball *et al.* [59] propose a technique to localizes the error cause in the error trace and generates multiple error traces. This technique generates multiple error traces by making use of the existing correct traces to localize the error cause. This is similar to SLAM [103] that verifies C programs and localizes the errors in device drivers. The algorithm performs the following functions in the program: highlighting each error location, introducing halt statements, and re-running the model checker to produce additional error traces. Finally, the framework by Chechik and Gurfinkel [49] provides a simple and unified way to interact with the counterexample generator, which combines property-based and model-based choices.

4.2.4. Answer to RQ2

Counterexample minimization is by far the dominant method for processing counterexamples in the context of counterexample explanation. 38 primary studies use this method (33% of all primary studies and 70% of all studies that process a counterexample). This strengthens our hypothesis that highlighting all the essential information like erroneous states and variables in the concrete counterexample is helpful for non-experts for error comprehension.

The model checkers SPIN and Maude are most often used for applying search methods such as DFS and BFS to minimize a counterexample. Model transformation is another method used for minimizing counterexamples. It relies on domain-specific design models from the user and transform them to formal verification models. Further, the counterexample generated during verification is translated back to the design models provided by the user. This hides the complexity of using a formal notation and logic and thus, the approach performs both the model abstraction and the counterexample representation.

One of the biggest advantages of model checking is the generation of a counterexample for error identification. Contrary to counterexamples, witnesses provide traces that satisfy the specification. This enables users to identify quickly the offending behavior and supports debugging by comparing the correct with the offending behavior. According to Copty *et al.* [100], one of the core advantages of multiple counterexamples is that all the possible counterexamples are identified in a single run which, enables a user to find all possible error root causes.

4.3. ***RQ3:*** *What kind of additional information is used to enrich the counterexample explanation?*

According to Pakonen *et al.* [39], “visualization of each step of the counterexample trace in the model may not be sufficient for fast understanding of the essence of the counterexample”. This statement motivates the use of additional information to enrich counterexample explanations to ease error comprehension.

However, in our survey we only found 8 primary studies (7% of all primary studies) that provide additional information along with the counterexample explanation.[6] Consequently, we collected qualitative statements from the primary studies that illustrate such additional information, grouped by the type of counterexample representation.

4.3.1. Information enriching graphical representations of counterexamples

We only found 3 primary studies (6% of all 54 primary studies that use graphical representations of counterexamples) that enrich graphical representations of counterexamples with additional information. A cross-platform tool by Pakonen *et al.* [39] visualizes the counterexample generated for LTL specifications by NuSMV by highlighting the values of atomic propositions, which are important for understanding the counterexample. Similarly, Beer *et al.* [104] find a set of causes for a specification failure on the given counterexample trace, using the notion of causality introduced by Halpern and Pearl [105]. These causes are presented to the user as a visual notification.

Error localization is not only performed for the counterexample or specification, but also for the design or verification model. Structural Analysis of Counter-Examples (STANCE) builds the complete counterexample using the Simulink simulator and analyzes this counterexample to compute the causes [106]. The resulting colored model allows the user to focus on the structural part of the model that is responsible for violating the property.

4.3.2. Information enriching textual representations of counterexamples

5 primary studies (29% of all primary studies that use textual representations of counterexamples) provide additional information that enrich textual representations of counterexamples. Particularly, domain ontologies and vocabularies are used to enhance the counterexample with additional information to ease error comprehension. Crapo *et al.* [55, 107] and Moitra *et al.* [56, 57] use Requirements Analysis Engine (RAE) and Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests (ASSERT) that accept a formal requirement in an easily understandable syntax by making use of a domain ontology. Further, they analyze an incomplete set of requirements and localizes the error by identifying the responsible requirements with an error marker. Errors, warnings, and informational markers are used to make the user aware of the errors and ambiguities. Likewise, Berg *et al.* [6] use a domain-specific vocabulary to present errors in railway interlocking systems in a language similar to natural language. They further present a table explaining an action of the railway interlocking systems relevant to the error, and the corresponding action that permits the error to occur.

[6]These primary studies are listed in [31, Table 6].

4.3.3. Answer to RQ3

Error localization is used to highlight or to identify the cause of an error in a given design or verification model. For example, Pakonen *et al.* [39] highlight atomic proposition values, Beer *et al.* [104] mark errors in a given specification, and ASSERT [56, 57] highlights the responsible requirements with error markers. These approaches highlight errors in the given input design model, while the approach by Berg *et al.* [6] renders an explicit explanation, further aiming to ease error comprehension.

4.4. ***RQ4:*** *For which input domains are the counterexample explanation approaches available? What is the influence of the input domain on the explanation?*

To answer this research question, we collected the input and output domains used by the primary studies (*cf.* Figure 1). We investigate whether counterexamples are represented and explained within the input domain or whether an output domain different from the input domain is used. Moreover, we collected qualitative data that demonstrate the impact of the used domains on the explanations of counterexamples. For this research question, we excluded 35 primary studies that use trace representation, since these keep the original format of the counterexamples. Thus, we investigated the remaining 81 primary studies that transform counterexamples to graphical, textual, or tabular representations.

4.4.1. Input and output domains in counterexample explanation

Figure 4a shows the count of primary studies that use certain input (blue bars) and output domains (orange bars).[7] In the following, we discuss each domain with examples from the primary studies. We further provide qualitative statements from these studies to show the effects of the domain on counterexample explanation and interpretation.

The most used input domain is the *modeling language of the used model checker* that we found in 35 primary studies (30% of all primary studies). The second-most used input domain are *programming languages* such as C, ANSI-C, or PLC (15 primary studies, 13% of all primary studies). Among these 15 studies, 5 studies explain the counterexample in the given input programming language. For example, the approach by Clarke *et al.* [60] verifies ANSI-C programs, and the error is localized right in the given input program. The graphical interface presents the counterexample traces and allows stepping through the trace in the same way as a debugger allows stepping through a program.

Pakonen and Björkman [38] state in the context of counterexample explanation that "one of the strengths of function block diagrams as a programming language is that it is relatively easy to understand the flow of processing from the inputs to the outputs". Accordingly, we found two component-based input and output domains in our survey: *function block diagrams* and *component diagrams*. Functional block diagrams are used by seven primary studies as an

[7]The primary studies for each input and output domain are listed in [31, Tables 7 and 8].

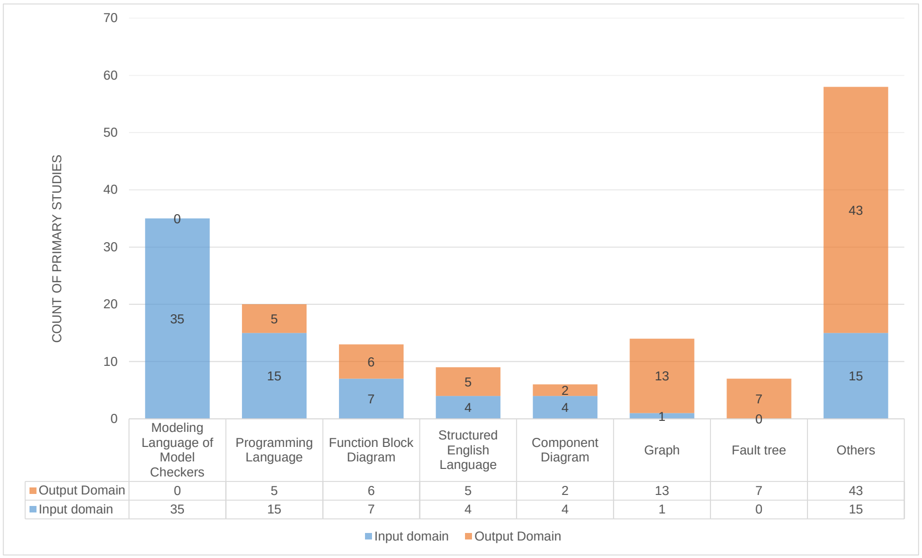
input domain and six primary studies as an output domain. Similarly, component diagrams are used by four primary studies as an input domain and two primary studies as an output domain. In total, 11 of all primary studies (9%) use a component-based domain for the input, and 8 of all primary studies (7%) use such a domain for the output. Frameworks such as MODCHK, FASTEN, and AutoFOCUS 3 represent the input in a component-based domain. MODCHK [37–39] further simulates the counterexample in the given input domain, a function block diagram. Similarly, the frameworks FASTEN and AutoFOCUS 3 simulate counterexamples in the provided input component diagram.

A natural language like input domain that we found in our survey is *Structured English Language* (4 primary studies, *i. e.*, 3% of all primary studies), which overshadows the complexity of formal notations for engineers [55, 56]. The ASSERT framework by Crapo *et al.* [55] and Moitra *et al.* [56, 57], as well as the RailComplete framework by Luteberget and Johansen [54] accept Structured English Language as the input domain. To improve the interpretation of counterexamples, 5 primary studies (4% of all primary studies) use *Structured English Language* as the output domain. One of the examples for generating Structured English Language referring to the counterexample is presented by Feng *et al.* [53] who define a set of Structured English Language sentences to describe the requirement violation of robotic behavior. An example for such a structured sentence is the template "The robot <action> when <proposition>" [53]. A violation of a requirement is explained using the template by referring to the counterexample and replacing <action> with possible robotic actions, and the <proposition> is replaced with (conjunctions of) violated atomic propositions that represent the robot's configuration.

Graphs (13 primary studies, *i. e.*, 11% of all primary studies) and *Fault trees* (7 primary studies, *i. e.*, 6% of all primary studies) are the most-used output domains for counterexample explanation in the primary studies, which does not contribute much for input domains. One of the frameworks that presents a counterexample as a graph is Computing Minimal Counterexamples (COMICS) [81]. It allows the user to visualize either the original graph or the abstract graph, translated from the counterexample. A fault tree provides a compact and concise representation of system failures using a graphical notation well known to safety engineers [43]. The work by Leitner-Fischer and Leue [45, 46] is an example for using visual representations of fault trees as the output domain. To achieve this, they perform a causality check that computes event combinations causing a property violation, together with the order in which the events have to occur to be causal.

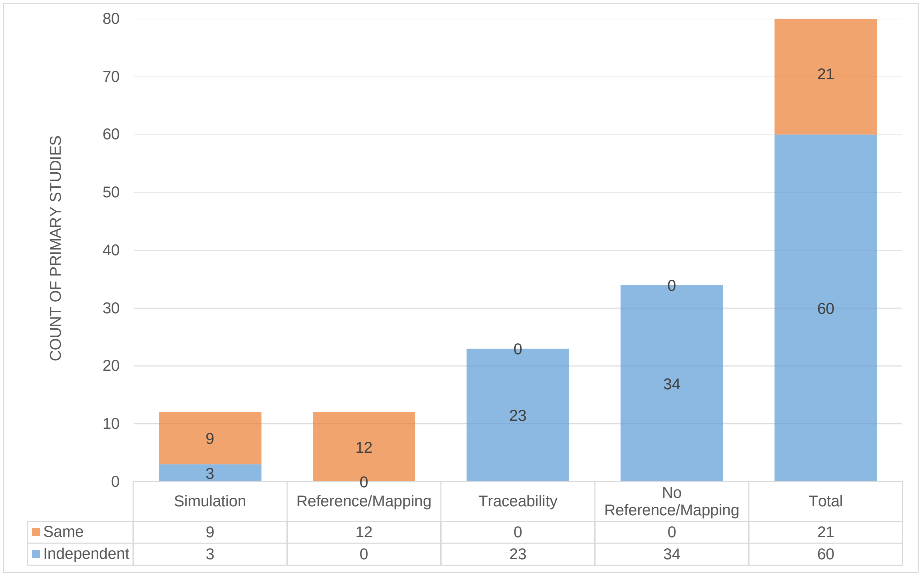
4.4.2. Relationship between input and output domains

The input domain, particularly the way systems are modeled, influences the explanation of the counterexample using a certain output domain. We categorize relationship between the input and output domains mainly into two different groups: explanation of the counterexample is done in an output domain *identical* to the input domain or *different* from the input domain. An overview is given in Figure 4b that further details these two groups into four categories:



	Modeling Language of Model Checkers	Programming Language	Function Block Diagram	Structured English Language	Component Diagram	Graph	Fault tree	Others
Output Domain	0	5	6	5	2	13	7	43
Input domain	35	15	7	4	4	1	0	15

(a) Input (blue) and output (orange) domains used by counterexample explanation approaches. Domains that are used as input or output in less than six primary studies are counted as "Others". These other domains are CNL, SCADE/Simulink model, Control signal table, Requirement, UAV mission planning, GPS Image, BPMN Model, Graph, LSC, RTL Model, UML are input domains. Property, MSC, UAV mission planning, GPS Image, Sequence Diagram, MSC, BPMN Model, LSC and STD, Sequence Diagram, Flow Chart, State machine, and Tabular View are output domains.



	Simulation	Reference/Mapping	Traceability	No Reference/Mapping	Total
Same	9	12	0	0	21
Independent	3	0	23	34	60

(b) Influences of the input domain on the counterexample explanation.

Figure 4: **RQ4:** Input and output domains, and influences of the input domain on the counterexample explanation.

1. *Reference/Mapping:* Variables from the counterexample and their static values are attached to the input model or referred to in the input model.
2. *Simulation:* In addition to reference/mapping, variables from the counterexample and their values are animated in the provided input model or in any of the graphical representation, *e. g.*, their dynamic values can be displayed for each step of the counterexample.
3. *Traceability:* The counterexample is explained independently of the input domain, but the input domain is referred to in the counterexample explanation. In this case, the input and output domains are different.
4. *No Reference/Mapping:* The input and output domains are completely independent of each other; the counterexample explanation does not have any reference to the input.

Figure 4b shows that from 81 primary studies, 60 (74%) use an output domain *different* from the input domain. Only 21 primary studies (26%) provide explanations of counterexamples using an output domain *identical* to the input domain.[8]

Identical input and output domains. 9 of the 21 primary studies that use identical input and output domains (43%) perform *simulation*, and 12 (57%) rely on *reference/mapping.* Simulation by step-wise animation in both forward and backward directions is one of the ways to display and explain a counterexample. FASTEN [40] and AutoFOCUS 3 [89] use the component diagram and MODCHK [36–39] uses a functional block diagram as the input and output domain. In these frameworks, a simulator animates the counterexample back and forth to display every state variable with its values. Similarly, Groce *et al.* [58, 61] present a graphical user interface designed for C Bounded Model Checker (CBMC) [108] that allows the users to interactively step through the counterexample traces, in the provided input programming language.

Another option to visualize counterexample traces in the input domain is by referencing/mapping it to the user-provided design model. STANCE [106, 109] has been developed in the Matlab/Simulink environment and the complete counterexample is visualized using a Simulink model where paths of the cause are colored. Muram *et al.* [5] present the visual support based on the information provided by the counterexample analyzer mapped to the input, a Business Process Model and Notation (BPMN) model. Particularly, the model element that indicates the violation of an LTL property, the containment violations, and the elements that satisfy the property are highlighted in different colors.

Different input and output domains. As shown in Figure 4b, 23 of the 60 primary studies with different input and output domains (38%) use *traceability*, 34 (57%) use *no reference/mapping*, and 3 (5%) use *simulation.* Considering this research question and especially the influence of the input domain on the

[8]The primary studies for each category are listed in [31, Table 9].

counterexample explanation, we cannot determine any such influence for the category *No reference/mapping* since the input and output domains do not have any relationship to each other.

Considering traceability, the approaches by Berg *et al.* [6] and Feng *et al.* [53] use domain terminology as a vocabulary to represent the counterexample in a natural language like format. Such approaches maintain traceability to access the vocabulary from the provided input in order to generate counterexample explanations. Likewise, in the work by Angelov *et al.* [52], the counterexample of Contract Language ANalyser (CLAN) is explained in a CNL. The work by Luteberget and Johansen [54] represent the requirement violation as a textual message to the railway engineer, which contains a reference to the rule source.

Considering simulation, the counterexample representation based on state machines in AutoFOCUS 3 [41, 110] can simulate either a path from the initial state to an erroneous state or a loop. Each step of the counterexample sequence is described by the actual value of variables and the i/o ports of the user-provided design model.

4.4.3. Answer to RQ4

Our results show that most counterexample explanation approaches use the language of the verification tool as the input domain, indicating that most approaches are targeted to tool experts, having a fair knowledge in formal methods. Counterexamples represented as fault trees in programming languages and function block diagrams support the corresponding domain experts to understand the error without the need for deep understanding of the formalism used by the model checker. This is especially true if the counterexample is simulated or animated in the given input domain. However, we only found a few primary studies for this technique [36–40, 89].

In most of the primary studies, the output domain is different from the input domain. Approaches like simulation of counterexamples extract the required information from the input domain and represents the error information independently. Graphical representations and textual languages like Structured English or CNL usually refer to the input domain and use the input domain as the primary source for generating a counterexample explanation.

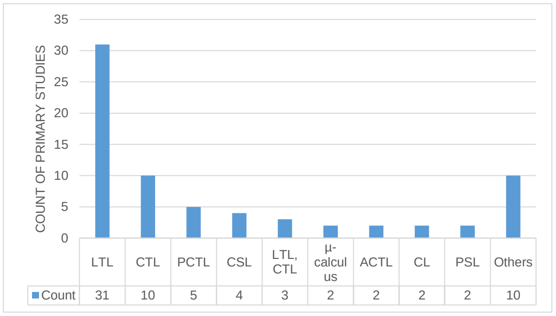
*4.5. **RQ5:** What are the different temporal logics used to express system specifications and what type of properties are covered in counterexample explanation approaches?*

To answer this research question, we provide quantitative results of the temporal logics and types of properties that are used by the primary studies.

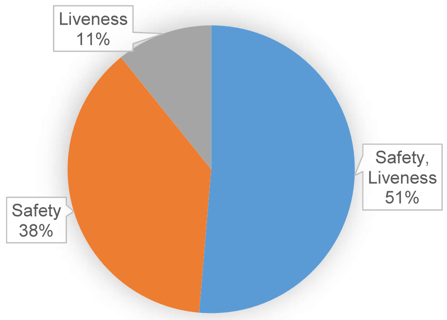
4.5.1. Specification logic

Figure 5a shows the results for the temporal logics used to specify requirements in the primary studies.[9] 31 primary studies (27% of all primary studies)

[9]The primary studies for each temporal logic are listed in [31, Table 10].



(a) Temporal logics used for specifications. Formalisms that only occur once in the surveyed literature are combined into the category "Others" (FLTL, TCTL, Datalog, FOL, ALCCLTL, XCTL, CTLK, ATL, and STLCS).



(b) Types of specification properties. Safety properties are considered in almost 90% of all primary studies.

Figure 5: **RQ5:** Types of system specifications and specification properties.

use *LTL* and 10 (9%) use *CTL*. For both cases, approaches that use graphical representations of counterexamples exist (*e. g.*, [42, 89, 111, 112] for LTL, and [98, 113, 114] for CTL). The frameworks NuSeen [64], PLCverif [115], and MODCHK [36] support both LTL and CTL. Furthermore, 5 primary studies (4% of all primary studies) use *Probabilistic Computation Tree Logic (PCTL)* and 5 (4%) use *Continuous Stochastic Logic (CSL)* to tackle probabilistic specifications. The 5 studies [79–81, 87, 90] that use PCTL minimize counterexamples while the 5 studies [43, 45, 78, 116] that uses CSL generate fault trees.

Other specification languages we found in our survey (two primary studies for each) are *Contract Language (CL)*, *ACTL*, *Property Specification Language (PSL)*, and *μ-calculus*. CL is inspired by dynamic, temporal, and deontic logic and is used for specifying contracts containing clauses. Angeloc *et al.* [52] convert the CNL language into CL using the Grammatical Framework (GF) and perform verification with the CLAN model checker. The errors found in the counterexample are then highlighted in the CNL. Frameworks like AutoFocus 3 [110] and PyNuSMV [101] support the μ-calculus along with LTL, CTL, CTLK and ATL. ACTL and PSL logics are verified using the NuSMV model checker. Clarke *et al.* [117] use ACTL to generate multiple counterexamples and Shen *et al.* [71] uses ACTL for counterexample minimization.

4.5.2. Types of specification properties

Specifications may be used to specify *safety* and *liveness* properties. Safety properties assert that something bad never happens, while liveness properties assert that something good will eventually happen [118]. A counterexample to a liveness property in a finite system is lasso-shaped, which consists of a prefix that leads to a loop. For safety properties, a counterexample is finite in length [119].

As shown in Figure 5b, most primary studies address safety properties (38%), or safety and liveness properties (51%) at the same time [67, 74, 76] while only 11% address the liveness properties alone.[10] Out of the 51% of studies that address safety *and* liveness properties, the majority of studies (32%) represent a counterexample in a graphical format, *e. g.*, animate the counterexample with a loop [32, 39, 104]. The remaining 19% follow different methods to process or represent the counterexample generated for *safety* and *liveness* properties, *e. g.*, Edelkamp *et al.* [74] use a nested depth-first search algorithm for analyzing liveness properties, but directed search for safety properties to generate a minimized counterexample.

Mostly the specification of liveness properties makes use of inevitable executions. Therefore, inevitable execution properties belong to a class of liveness properties [95]. The work by Barbon *et al.* [95] focus on improving the comprehension of counterexample specifically for such inevitability properties, which is predominantly used by developers in practice [15].

4.5.3. Answer to RQ5

In our survey, we found that 44 (38% of the primary studies) use either LTL, CTL, or both, and 10 (9%) use probabilistic specifications. However, just one primary study deals with counterexample explanation of real-time specifications [68], in this case based on UPPAAL [120, 121]. Finally, 51% of the primary studies verify both safety and liveness properties while 38% focus exclusively on safety properties and 11% on liveness properties.

*4.6. **RQ6:** Which verification tools and frameworks are developed and used to explain counterexamples, and how do they effect the counterexample explanation?*

To answer this research question, we collected the verification tools and frameworks used to develop counterexample explanation approaches. These results are of particular interest for practitioners looking for reuse of existing solutions. Furthermore, we collected qualitative data to demonstrate the effect of tools and frameworks on the counterexample explanation. Luteberget and Johansen [54] state that "*a verification tool which runs invisibly alongside the design, giving feedback on the current state of the design at any time could have a higher impact on the design process*".

4.6.1. Verification tools

Figure 6a shows the verification tools used by the primary studies.[11] The predominantly used model checker in the context of counterexample explanation is the *NuSMV/nuXmv/Symbolic Model Verifier (SMV)*. It is used by 31 primary studies (27% of all primary studies). nuXmv extends NuSMV that extends SMV [122] and they can verify specifications expressed in LTL, CTL, PSL, or

[10] The primary studies for each type of property are listed in [31, Table 11].

[11] The primary studies for each verification tool are listed in [31, Table 12].

Real Time Computation Tree Logic (RTCTL). We found two particular primary studies that use NuSMV. Schuppan and Biere [67] identify the shortest lasso-shaped counterexample for LTL, and Pakonen *et al.* [39] highlight the value of atomic proposition that are important for understanding counterexample.

12 primary studies use *SPIN* [123, 124] (10% of all primary studies) and 5 use *Maude* [125–127] (4%) to verify LTL specifications. For instance, SPIN is used with minimizing counterexamples [48, 73, 75, 76]. Maude integrated with Draw-SVG is used to animate user-defined state machines [42, 111, 112, 128]. Furthermore, *Verification Interacting with Synthesis (VIS)* [129, 130] is used by 4 primary studies (3% of all primary studies) and A Computational Logic for Applicative Common Lisp (ACL2) by 4 (3%). VIS supports verification of fair CTL specifications, simulation of logic circuits, and explanation of counterexample in a flow chart representation. *ACL2* [131, 132] is a tool for modeling, simulation, and theorem proving. The ASSERT tool integrates ACL2 with ontologies [55–57, 107] to analyze an incomplete set of specifications and localize the error by identifying which specifications are responsible for the error.

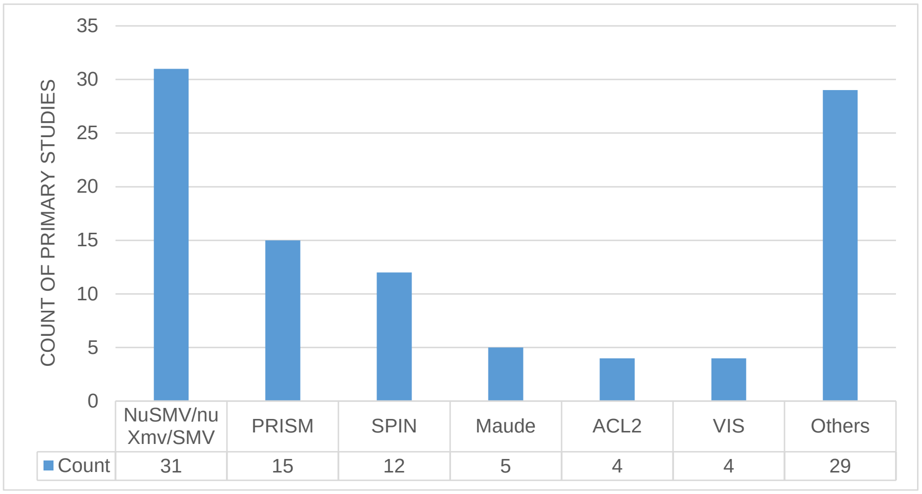
Model checkers like *PRISM* [133, 134] and *Markov Reward Model Checker (MRMC)* [135, 136] are used to verify probabilistic specifications. 15 primary studies (13% of all primary studies) use PRISM and among them 3 primary studies [80, 92, 116] use both MRMC and PRISM for probabilistic model checking, and particularly with minimization of counterexamples [80, 116]. Leitner-Fischer and Leue [45], and Kuntz *et al.* [43] use the FaultCAT and CX2FT frameworks to generate fault trees with the help of counterexample generated by PRISM. Similarly, Leitner-Fischer and Leue [46, 78] use the SpinCause framework to generate fault trees with SPIN and PRISM.

4.6.2. Frameworks

In our survey, we did not find any framework that is reused by more than four primary studies (Figure 6b).[12] The *ASSERT* framework used by 4 primary studies (3% of all primary studies) check the requirements for conflicts and completeness by using an automated theorem prover [55–57, 107]. Further, it generates requirements-based test cases using SMT. Moitra *et al.* [56] state that "ASSERT saves time and cost by identifying errors early in the development process and automating requirements-based test generation".

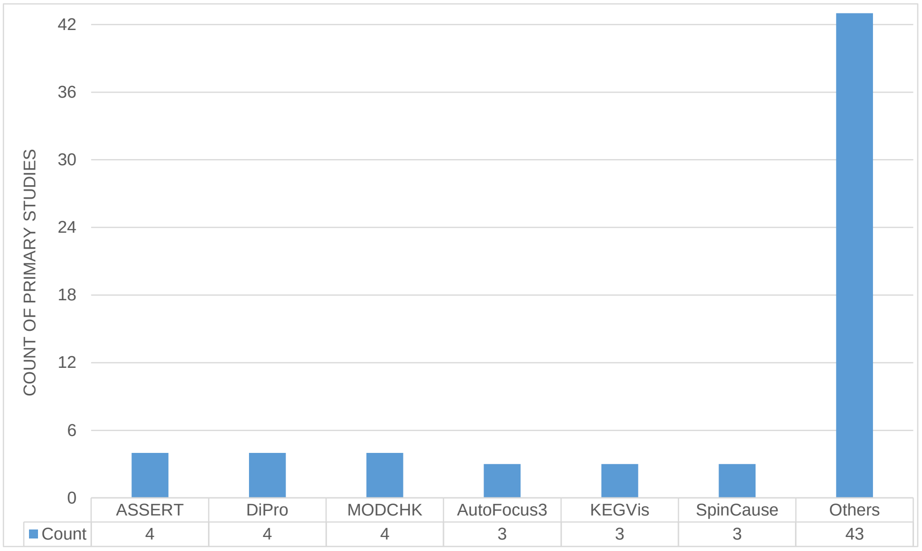
Out of all frameworks shown in Figure 6b, only *DiPro* that is used by 4 primary studies (3% of all primary studies) and *SpinCause* that is used by 3 studies (3%) support verification of probabilistic systems. The open-source tool DiPro is used with PRISM and MRMC for the computation and graphical representation of probabilistic counterexamples for DTMCs, CTMCs and MDPs [116]. SpinCause is based on SpinJa [137], a Java re-implementation of SPIN [78]. Florian and Leue [46] use SpinCause for causality checking of Process or Protocol Meta Language (PROMELA) and PRISM models and the approach is evaluated with industry sized models.

[12]The frameworks and the primary studies that use them are listed in [31, Table 13].



	NuSMV/nuXmv/SMV	PRISM	SPIN	Maude	ACL2	VIS	Others
■ Count	31	15	12	5	4	4	29

(a) Verification tools used by the primary studies. Tools that with less than four occurrences are clustered in the category "Others" (Zchaff, Z3, CADP, ViVe/SESA, topochecker, SpinJa, SLDV, SLAM, Yices, MTSA, LTSA, CPAchecker, Ultimate Automizer, CLAN, CBMC, JPF, XCheck, SAL, CoCoSim, JKind, MCMAS+, and MiniSAT).



	ASSERT	DiPro	MODCHK	AutoFocus3	KEGVis	SpinCause	Others
■ Count	4	4	4	3	3	3	43

(b) Frameworks for counterexample interpretation. Frameworks that occur less than three primary studies are combined into the category "Others" (IFADIS, STANCE, FAULTCAT, CX2FT, SpinRCP, MechatronicUML, AnaCon, RailComplete, [Mc]SQUARE, Pseudo-merge, EOFM, ProofProd, Evidence Explorer, ELARVA, Arcade.PLC, SMART, Alfi, PyNuSMV, RuleBase PE, COMICS, AMASE, NuSeen, FASTEN, PLCverif, Ivy, DSValidator, IBM RoseRT, Theseus, VIS, MACEMC, FLAVERS/Ada, OERITTE, and GraphML).

Figure 6: **RQ6:** Frameworks and verification tools used by primary studies.

Two frameworks that support animating counterexamples are *MODCHK* [37–39] and *AutoFocus 3* [41, 89, 110]. 4 primary studies (3% of all primary studies) use *MODCHK* to animate a counterexample in the user-provided input function block diagram. Similarly, *AutoFocus 3* used by 3 primary studies (3%) animate a counterexample in the user-provided component diagrams.

Kounterexample generator and visualizer (KEGVis) used by 3 primary studies (3% of all primary studies) presents witnesses graphically with the daVinci Presenter [138] for layout and exploration [139]. Besides simply browsing the witnesses, the user can customize the exploration strategy, for instance, in terms of forward and backward exploration, adjusting step granularity, and choosing witnesses based on size. Particularly, KEGVis is used together with the model checkers Multi-valued Model-Checker (XChek) and NuSMV in [49, 139, 140].

In the following, we summarize notable frameworks that are used by less than three primary studies.

Gerking *et al.* [68] present a model-to-model transformation from a design model (in-terms of domain-specific model checking [141]) to a verification model that is based on the input language of the model checker UPPAAL. The key feature of this approach is to bridge large differences between the domain-specific modeling language and the model checker's input language. Thus, it also translates counterexamples back to the level of the domain-specific models. A similar approach is followed for the AutoFocus 3 [41, 89] and FASTEN [40] frameworks.

SMGA is an animation tool that visualizes counterexamples generated by Maude as a *state machine* [111]. In the same way, Nguyen [128], Phyo and Ogata [112], as well as Nguyen and Ogata [32, 42] allow the user to design picture of animations, adjust the speed of animations, and select states that satisfy some given conditions and/or constraints from a finite computation. *Theseus* provides two animation options [142]: automatic playback and an incremental playback option, by using *state diagram* animation and *sequence diagram* generation respectively. The counterexample is animated by automatic playback while the incremental playback animates the counterexample in a step-by-step manner. Similarly, *SpinRCP* [143] transforms a Spin simulation trail into a standard Message Sequence Chart (MSC) and supports interactive simulations.

STANCE is a framework that is integrated with the Simulink toolset that visually shows the structural parts of Simulink models. The tool thus filters out the irrelevant data from the counterexample, thus easing the job of the engineers to focus only on interesting details of its execution [106, 109]. Similarly, the *CLEAR* [94] tool highlights actions that causes violation in the counterexample and specifically supports the debugging of concurrent systems.

Gerking *et al.* [68] present a model-to-model transformation from a design model (in-terms of domain-specific model checking [141]) to a verification model that is based on the input language of the model checker UPPAAL.

4.6.3. Answer to RQ6

Our results show that the use of verification tools and specification languages (Section 4.5) are clearly coupled, because certain languages can only be verified by certain tools. Since LTL is the most used logic (38%), the verification tools

NuSMV/nuXmv/SMV, SPIN and Maude are found to be used by 41% of the primary studies (all of them can verify LTL specifications). 13% of primary studies use probabilistic model checkers such as PRISM and MRMC.

Each of the most used frameworks ASSERT, DiPro or MODCHK is only used by 4 primary studies. Frameworks such as MODCHK, AutoFocus 3, SMGA, FASTEN, and Theseus support animating the counterexample in a user-given design model. By performing the animation, engineers can understand the counterexample by executing it stepwise. Particularly, AutoFocus 3 and FASTEN as well as the approach by Gerking *et al.* [68] support minimizing the counterexample along with explaining the counterexample in a graphical format.

*4.7. **RQ7:** How are counterexample explanation approaches evaluated?*

Among all primary studies, 97 studies (84%) perform an evaluation. Among these 97 studies, we have identified the application domain in 69 studies (71%), while 28 studies (29%) do not target a specific domain. Similarly, 87 of the 97 studies (90%) mention the types of applications used for the evaluation.

4.7.1. Application domains

Figure 7a shows the results for the application domains used for evaluating counterexample explanation approaches by the primary studies.[13] 14 primary studies (14% of all studies that perform an evaluation) perform the evaluation on *communication protocols*, *e. g.*, to evaluate graphical counterexample representations [32, 42, 111, 112, 144] or counterexample minimization [74, 86, 145]. 14 further primary studies (14%) perform the evaluation in the *hardware* domain, *e. g.*, with applications such as PLC software [36, 115, 146] and circuits [69, 100].

In total 28 primary studies (29%) evaluate their approach in a safety-critical application domain such as *automotive* (10), *robotics* (6), *avionics* (5), *nuclear* (4), and *railway* (3). Evaluation of textual representations is performed, *e. g.*, in avionics [55–57, 107], the railway domain [51], and robotics [53]. Evaluation of graphical representations is performed, *e. g.*, in the automotive [46, 109], and nuclear domain [37–39, 147].

4.7.2. Types of Applications

We categorize the applications used for the evaluation in primary studies into three groups listed below. We additionally distinguish between primary studies that introduce novel applications and those that reuse existing applications.

1. *Industrial application:* The evaluation uses a real-world industrial application.
2. *Non-industrial application:* The evaluation uses a real-world non-industrial application such as an application developed in a research lab.
3. *Example application:* The evaluation uses a simple and small exemplary application such as a toy example.

[13]The primary studies for each application domain are listed in [31, Table 14].

Figure 7b shows the results for these types of applications.[14] 16 primary studies (16% of all studies that perform an evaluation) are evaluated with *industrial applications*, which can be considered a good indication of their maturity and scalability. Industrial applications range from Finnish nuclear industry [37] and aero-engine blade forging [148] to desalination plants [41]. Approaches are also evaluated by *referring to existing industrial applications* in 22 primary studies (23%). Examples are the airbag system by Aljazzar *et al.* [149] and referred by Kuntz *et al.* [43] and Aljazzar *et al.* [116], and a flasher manager application by Collavizza *et al.* [150] and referred by Castillos *et al.* [109].

Likewise, 31 primary studies (32%) evaluate their approaches *referring to existing non-industrial applications*, *e. g.*, to evaluate approaches to counterexample minimization [67, 74, 75, 79, 80, 84, 145], graphical representations [35, 42, 111, 112, 144], and textual representations [51–53, 61]. An evaluation based on *non-industrial applications* is performed by 7 primary studies (7%), *e. g.*, to evaluate some of the textual representation approaches [51–53, 61].

Finally, 17 primary studies (18%) use *example applications*. Mostly, such applications are used to demonstrate the proposed approach, *e. g.*, for graphical representations of counterexamples [35, 128, 151].

4.7.3. Evaluation aspects

We distinguish three types of aspects that are the goals of the evaluation [30]:

1. *Efficiency and Performance:* Evaluation focuses on the efficiency and computational demand such as execution time of approaches.
2. *Effectiveness:* Evaluation focuses on effectiveness and usability of counterexample explanation approaches.
3. *Scalability:* Evaluation investigates whether an approach is scalable and can be applied to complex scenarios and systems.

Figure 7c shows the distribution of the evaluation aspects among the primary studies.[15] The aspect of *efficiency/performance* is evaluated in 55 primary studies (57% of all studies that perform an evaluation). Two exemplary studies are the ones by Leitner-Fischer and Leue [46] and Tan *et al.* [84]. The run-time of the proposed approaches is noted with different use cases for minimizing counterexample using the DFS and BFS search methods.

Close to the count of efficiency/performance, *effectiveness* is evaluated in 51 primary studies (53%). The effectiveness of frameworks IFADIS [66] and FASTEN [40] are evaluated in user studies. In our survey we also found 6 additional primary studies that evaluate approaches focusing on both *efficiency/performance and effectiveness* [10, 39, 51, 97, 113].

An evaluation focusing on *scalability* is performed by 13 primary studies (13%). Two examples are the studies by Dominguez *et al.* [102] focusing on

[14]The primary studies for each application type are listed in [31, Table 15].
[15]The primary studies for each evaluation aspect are listed in [31, Table 16].

generating multiple counterexamples, and by Barbon *et al.* [7] focusing on minimizing counterexamples for concurrent systems. The proposed approaches are evaluated with several use cases that are different in size and complexity to show the scalability of the approach.

4.7.4. Evaluation methods

We distinguish three different types of evaluation methods [30]:

1. *Use Case:* Anecdotal evidence, the evaluation aspect is applied to one or more use cases and the results are reported.
2. *Benchmark:* An evaluation aspect is compared to a particular baseline or any established approach.
3. *User Study:* Evaluation from the user's point of view, where feedback is collected from users, *e. g.*, domain experts.

Figure 7d shows the results for the evaluation methods used by the primary studies.[16] Anecdotal evidence based on *use cases* is the most used method, found in 85 primary studies (88% of all studies that perform an evaluation), *e. g.*, to evaluate graphical [38, 41, 114, 130, 142] or textual representations [54, 55].

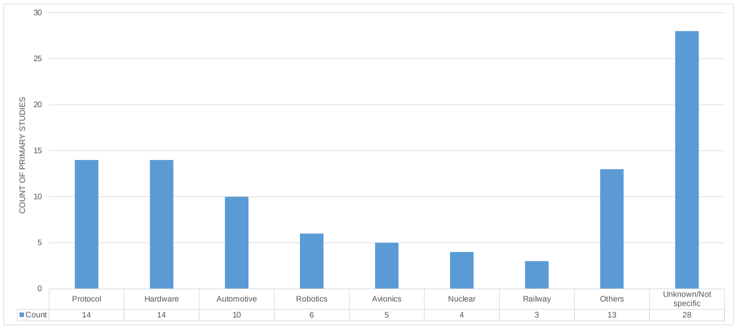
An evaluation with a *benchmark* is performed by 21 primary studies (22%), mostly to evaluate counterexample minimization [48, 74, 75, 81, 83, 86]. Notably, Beyer *et al.* [97] created benchmarks for 3964 verification tasks from all categories of SV-COMP 2015 [152]. Chang *et al.* [50] evaluate their trace minimization approach with nine benchmark designs.

However, only 6 primary studies (6%) perform a *user study* to assess the effectiveness of counterexample explanation approaches. Barbon *et al.* [7] perform a user study with 17 developers to assess their approach to counterexample minimization. The developers are asked to analyze the original and the minimized counterexample. The results of this use study shows that developers spend more time on analyzing the original counterexample than the minimized counterexample. Moitra *et al.* [56] state that "Having engineers from the business units embedded into the team developing the tools was important because it kept us focused on solving the real problems the business was having."

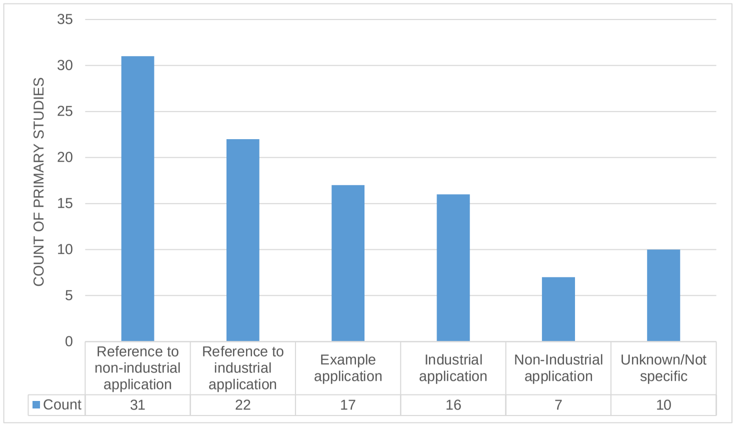
4.7.5. Answer to RQ7

A particular striking point is that almost 29% of the primary studies evaluate their approaches in a safety-critical domain. Furthermore, the number of approaches evaluated with industrial applications is quite significant (39% of all studies that perform an evaluation). This underlines the applicability of the evaluated approaches in critical real-world applications. Efficiency and performance as the evaluation aspect is addressed by 57% while effectiveness is evaluated by 53% of the primary studies. The most used evaluation method is the *use case*, occurring in 88% of the primary studies.

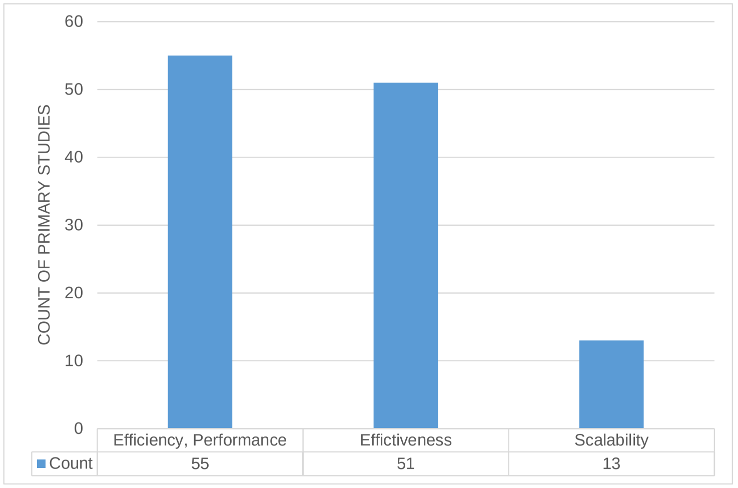
[16]The primary studies for each evaluation method are listed in [31, Table 17].



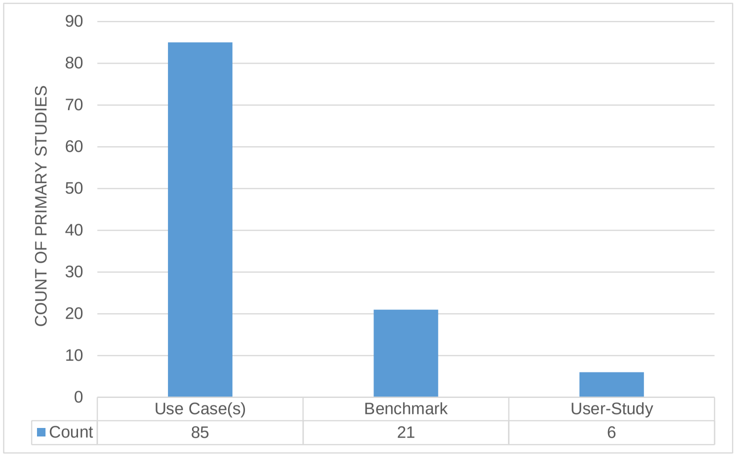
(a) Overview of the different application domains. Domains that were found in less than three primary studies, namely, Internet Service, Desalination Plant, Library, Billing Renewal, GPS Data, and Operating System are grouped as "Others".



(b) Types of applications.



(c) Overview of evaluation aspects.



(d) Overview of evaluation methods.

Figure 7: **RQ7:** Results for the application domain, type of application, evaluation aspect, and evaluation method. For each of them, if a primary study matches more than one category (*e.g.*, it uses several evaluation methods), the study is counted for each category (*i.e.*, for each evaluation method in this example).

5. Threats To Validity

There are several potential validity threats to the design of our systematic literature review. We discuss the external validity of our study and reliability of our analysis according to Wohlin *et al.* [30].

5.1. External Validity

To mitigate threats to the external validity of the survey, we define a systematic selection process as documented in Section 3 and depicted in Figure 2. The process resorts to the huge literature database *Google Scholar*. Google Scholar weights query results by the number of citations, which introduces weighting and validation by the scientific community into the corpus of our surveyed literature.

We are aware that the selection of keywords has an impact on the selection of potential publications. However, we argue that the common vocabulary of counterexample explanation is represented in our list of keywords. The usage of more general keywords such as "formal methods" would increase the initial count of publications at the cost of drastically increasing false positives beyond a reasonable amount. Therefore, we combined general keywords with the keywords that are specific to counterexample explanation.

As another means to increase external validity, we performed snowballing with the publications that we filtered from the initial search. This process potentially incorporates literature that is relevant to the field but is not matched by our keyword list.

In summary, we are confident that the selected literature for this survey is representative for its community, which minimizes threats to external validity.

5.2. Reliability

To ensure reliability of the analysis results presented in Section 4, we applied a four-eyes principle. The literature obtained by our selection process was collected in a table, shared and worked on by all authors of this survey.

For each of the research questions documented in Section 3.1, relevant data for the quantitative RQs was collected, reviewed, and aggregated. We applied descriptive statistics on the aggregated data to investigate the quantitative aspects of the research questions. For each of the qualitative aspects, relevant statements were collected in the shared table and reviewed as well. These statements were clustered, summarized, and exemplary statements were chosen for discussion.

We are confident that this approach minimizes threats to the reliability of the presented results.

6. Discussion

Based on the survey of the literature, we gained significant knowledge and understanding in the domain of counterexample explanation. In the following, we use this knowledge and understanding to discuss the main findings of our study, and suggest future research directions for the community.

6.1. Synopsis

In this survey, we address key concerns in model checking for error comprehension along the research questions introduced in Section 3.1. The result of the analysis shall serve as a reference for the research community, giving insights to existing research approaches. We surveyed publications since the early 2000's, indicating that counterexample interpretation is an active research field for at least two decades by now.

6.1.1. Counterexample explanations for different audiences and usage scenarios

We discuss counterexample explanations from the perspective of three different users: (**U1**) experts with strong knowledge of both the formal methods and the domain, (**U2**) domain experts with limited formal methods knowledge, and (**U3**) domain experts without formal methods knowledge.

Research question **RQ1** (Section 4.1) mainly focuses on four different types of counterexample representation: textual, graphical, trace, and tabular notation. These different representations are tailored to users based on their experience, knowledge, and background. Counterexample representation as a trace or tabular notation seems to be most suitable for user **U1** as it is still close to the raw output of the model checkers. Tabular notations convert a counterexample into rows and columns. To obtain a trace representation, the counterexample is processed and provided as a trace.

Some of the graphical counterexample representation formats discussed in the context of **RQ4** (Section 4.4), *e. g.*, fault trees, function block diagrams, and component diagrams, as well as representations inside a programming language seem to be suited for user **U2**. For instance, compared to a lengthy counterexample, compact and concise fault tree representation is efficient and accustomed to safety engineers, easing comprehension of errors. Similarly, function block diagrams and component diagram are suitable for system engineers or system developers just as programming languages are for software engineers and programmers.

Additionally, with **RQ4** (Section 4.4) we found that a representation of a counterexample within the input domain is easy for a domain experts to comprehend, *e. g.*, by reference/mapping and particularly by simulation. The same argument is made by Loer and Harrison [66], who argue that "to be useful, the results of the analysis need to be visualized in a way that the designer can use. Traces can form the backbone for scenarios that may be analyzed by requirements engineers. They can be used to illustrate and communicate the implications of design decisions to and between design teams."

In most of the primary studies, counterexamples are represented in a graphical format. While such graphical visualization is useful mainly for user **U1** and **U2**, it may not provide enough support for user **U3**. To overcome this, textual explanations of counterexamples seem to be a promising direction, *e. g.*, expressed in a CNL or structured English. These explanations may be enriched by domain ontologies, domain vocabularies, and domain-specific terminology to improve their value. This type of language does not require any formal methods knowledge, thus it is suitable for user **U3**. Furthermore and as discussed with **RQ2** (Section 4.2) and **RQ3** (Section 4.3), natural language like textual representation along with additional information like error highlighting or error localization in the given input source, improves error comprehension and supports user **U3** in analysis and debugging.

6.1.2. Counterexample processing

Traces are one form of representing counterexamples, which we discussed with **RQ2** (Section 4.2) and **RQ3** (Section 4.3). It is further categorized into witness traces, minimized counterexamples, and multiple counterexamples. According to Aljazzar *et al.* [116], the shortest diagnostic paths, *i. e.*, the paths with least transitions, are preferred over longer paths. Copty *et al.* [100] state that using multiple counterexamples allows to find all possible failures in a single verification run, and has the ability to identify more than one root cause so that it can reduce the number of verification runs.

From our point of view, multiple counterexamples can be useful for experts to understand an error in a detailed and time-efficient manner. However, as per the statement from Aljazzar *et al.* [116], reduction of counterexamples by removing unwanted paths, states, and variables can make the error comprehension easier when compared to concrete counterexamples.

Having witness traces along with a counterexample is an added advantage, which we discussed with **RQ2** (Section 4.2) and **RQ3** (Section 4.3) in detail. Witness traces are proofs that highlights the behavior of the model. It is used to justify the result of the model checker, which are contrary to the counterexample. Comparing witness traces with counterexamples can aid in identification of the erroneous behavior in a counterexample [153, 154].

Moreover, with **RQ2** (Section 4.2) we summarized different methods, techniques, or approaches that are used to process counterexamples. Most of these approaches are used for counterexample minimization. In our view, integration of model transformation with model checker promises to be an efficient method because it performs both model abstraction and counterexample representation when transforming design models to verification models and vice versa. For instance, works by Ratiu *et al.* [40], Campetelli *et al.* [41], Gerking *et al.* [68], and Luteberget *et al.* [51] present comprehensive frameworks that perform both the model transformation and verification. Such frameworks hide the formal models and visualize relevant information from the counterexample in the user-provided design model of the system.

6.1.3. Frameworks and tools

Looking at the quantitative results of **RQ6** (Section 4.6), frameworks for counterexample representation do not seem to be reused in a regular manner. Generally, if a framework is found in multiple publications, these are usually quite homogeneous research groups around the framework's authors. Only a few publications were found that have re-used existing third-party frameworks.

The same is true for verification tools. Even though the large number of verification tools we found in the literature, only a few verification tools such as NuSMV, SPIN, Maude, and PRISM have a wide adoption. NuSMV is the most used verification tool in the surveyed literature. A reason might be that NuSMV supports different types of model checking, model encoding techniques, and input domains.

6.1.4. Application and evaluation

With **RQ7:** (Section 4.7), we discussed evaluation aspects, methods, types of applications, and application domains. We would like to highlight that the surveyed approaches were evaluated with a considerable number of industrial applications in safety-critical domains such as automotive, nuclear industries, and avionics. We show that the preferred counterexample representation depends on the application domain. For certain domains, graphical representation is more feasible, while for other domains textual representation is best suited. For example, the textual representation from Berg *et al.* [6] and Luteberget *et al.* [51, 54] is found to be efficient for the railway domain, while fault tree representation is found to be most suited for safety engineers, *e. g.*, in the automotive domain [43].

6.2. Future directions

This survey reveals several open points and future directions to enhance counterexample explanation approaches. Referring to the quantitative results of **RQ1**, there is a considerable amount of work that uses graphical representations of counterexamples. Still, approaches focusing on textual representations as well as representations suited for laypersons (**U3**) are to be improved. Similarly, the quantitative results of **RQ2** show that a considerable amount of work addresses counterexample minimization and its visualization. However, work focusing on multiple counterexamples particularly, effective visualization of multiple counterexamples and highlighting relevant sources for debugging of multiple counterexamples seems to provide potential for further research. Such research could support comprehending multiple counterexamples for experts in formal methods (**U1**) and domain experts with limited knowledge in formal methods (**U2**).

Looking at the quantitative results for verification tools and temporal logics in **RQ5** and **RQ6**, qualitative model checkers such as NuSMV and SPIN, and qualitative logics such as LTL and CTL are used predominantly. Although there are several challenges in using probabilistic model checkers as discussed with **RQ6**, in this survey we found a considerable number of publications using probabilistic model checkers. However, the number of approaches using real-time model checkers is very low, which indicates room for further research. Similarly, referring to **RQ5** for specification properties, there are plenty of contributions that address safety properties, leaving room for research on counterexample explanation when verifying liveness properties, especially considering the increase in autonomy of modern systems.

We found many frameworks for counterexample explanation (**RQ6**) and a considerable number of these are open-source. Therefore, it seems advisable to build upon existing frameworks and improve them. Also, a large number of publications address either processed counterexamples or counterexample representation. In our view, developing an integrated framework that could perform both: the processing of counterexample and representation of it might be a powerful tool that would help practitioners in adopting and using counterexample explanation approaches.

To improve the effectiveness and usability of counterexample explanations, user studies need to be performed to gain insights in the degree of error comprehension and understandability, the key concerns of counterexample explanation. In our survey, though, we only found a small number of user studies (see **RQ7**). Therefore, researchers should consider performing user studies that allow investigating the error comprehension and understanding that users obtain when using counterexample explanation approaches. Such user studies will provide evidence for the effectiveness and usability of such approaches from a user's point of view and thus can guide future research on counterexample explanation.

7. Conclusion

"*As researchers and educators in formal methods, we should strive to make our notations and tools accessible to non-experts.* – Edmund Clarke [155]"

This article surveys the available literature on counterexample explanation, which addresses key concerns in model checking for error comprehension, along with a set of quantitative and qualitative research questions. Analyzing 116 primary studies we identified in this survey, the quantitative results are provided for different types of counterexample representations, counterexample processing, model checkers, temporal logics, frameworks, applications, application domains, evaluation aspects, and evaluation methods. Further, from a qualitative viewpoint, we discuss aspects such as methods and tools used in counterexample explanation approach as well as their effects, dependencies, and challenges for the explanations. As a result, this systematic literature review shall serve as a reference for the research community, giving insights to existing research approaches focusing on counterexample explanation. To the best of our knowledge, our systematic literature review is the first to provide such a reference for counterexample explanation.

References

[1] E. M. Clarke, The birth of model checking, in: O. Grumberg, H. Veith (Eds.), 25 Years of Model Checking - History, Achievements, Perspectives, Vol. 5000 of LNCS, 2008, pp. 1–26.

[2] E. M. Clarke, O. Grumberg, D. A. Peled, Model checking, MIT Press, 2001.

[3] C. Baier, J. Katoen, Principles of model checking, MIT Press, 2008.

[4] E. M. Clarke, T. A. Henzinger, H. Veith, Introduction to model checking, in: Handbook of Model Checking., Springer, 2018, pp. 1–26.

[5] F. U. Muram, H. Tran, U. Zdun, Counterexample analysis for supporting containment checking of business process models, in: BPM, 2015, pp. 515–528.

[6] L. van den Berg, P. A. Strooper, W. Johnston, An automated approach for the interpretation of counter-examples, Electr. Notes Theor. Comput. Sci. 174 (4) (2007) 19–35.

[7] G. Barbon, V. Leroy, G. Salaun, Debugging of behavioural models using counterexample analysis, IEEE Transactions on Software Engineering (2019) 1–14.

[8] D. Ratiu, A. Nordmann, P. Munk, C. Carlan, M. Voelter, FASTEN: An Extensible Platform to Experiment with Rigorous Modeling of Safety-Critical Systems, Springer International Publishing, Cham, 2021, pp. 131–164.

[9] P. Ovsiannikova, I. Buzhinsky, A. Pakonen, V. Vyatkin, Oeritte: User-friendly counterexample explanation for model checking, IEEE Access 9 (2021) 61383–61397.

[10] S. Leue, M. T. Befrouei, Counterexample explanation by anomaly detection, in: SPIN, 2012, pp. 24–42.

[11] A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri, NUSMV: A new symbolic model verifier, in: CAV, 1999, pp. 495–499.

[12] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: CAV, 2002, pp. 359–364.

[13] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 46–57.

[14] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logics of Programs, Workshop, Yorktown Heights, New York, USA, 1981, pp. 52–71.

[15] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in property specifications for finite-state verification, in: ICSE, 1999, pp. 411–420.

[16] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, A. Tang, Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar, IEEE Trans. Software Eng. 41 (7) (2015) 620–638.

[17] A. K. Karna, Y. Chen, H. Yu, H. Zhong, J. Zhao, The role of model checking in software engineering, Frontiers Comput. Sci. 12 (4) (2018) 642–668.

[18] F. Wang, Formal verification of timed systems: a survey and perspective, Proceedings of the IEEE 92 (8) (2004) 1283–1305.

[19] S. Gabmeyer, P. Kaufmann, M. Seidl, M. Gogolla, G. Kappel, A feature-based classification of formal verification techniques for software models, Software and System Modeling 18 (1) (2019) 473–498.

[20] T. Ovatman, A. Aral, D. Polat, A. O. Ünver, An overview of model checking practices on verification of PLC software, Software and System Modeling 15 (4) (2016) 937–960.

[21] T. Grimm, D. Lettnin, M. Hübner, A survey on formal verification techniques for safety-critical systems-on-chip, Electronics (2018).

[22] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Progress on the state explosion problem in model checking, in: Informatics - 10 Years Back. 10 Years Ahead., 2001, pp. 176–194.

[23] E. M. Clarke, H. Veith, Counterexamples revisited: Principles, algorithms, applications, in: Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday, 2003, pp. 208–224.

[24] M. R. Prasad, A. Biere, A. Gupta, A survey of recent advances in sat-based formal verification, STTT 7 (2) (2005) 156–173.

[25] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, K. L. McMillan, An analysis of sat-based model checking techniques in an industrial environment, in: CHARME, 2005, pp. 254–268.

[26] V. D'Silva, D. Kroening, G. Weissenbacher, A survey of automated techniques for formal software verification, IEEE Trans. on CAD of Integrated Circuits and Systems 27 (7) (2008) 1165–1178.

[27] R. Pelánek, Fighting state space explosion: Review and evaluation, in: FMICS, 2008, pp. 37–52.

[28] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, H. Aljazzar, Survey on directed model checking, in: (MoChArt), 2008, pp. 65–89.

[29] K. BA, S. Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, Tech. rep., Technical report, Ver. 2.3 EBSE Technical Report. EBSE (Jan. 2007).

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, Experimentation in Software Engineering - An Introduction, Vol. 6 of The Kluwer International Series in Software Engineering, Kluwer, 2000.

[31] A. P. Kaleeswaran, A. Nordmann, T. Vogel, L. Grunske, Appendix of the paper: A Systematic Literature Review on Counterexample Explanation, Zenodo, 2021. `doi:10.5281/zenodo.5679227`.

[32] T. T. T. Nguyen, K. Ogata, A way to comprehend counterexamples generated by the maude LTL model checker, in: SATE, 2017, pp. 53–62.

[33] S. Liu, Validating formal specifications using testing-based specification animation, in: FormaliSE@ICSE, 2016, pp. 29–35.

[34] M. Li, S. Liu, Integrating animation-based inspection into formal design specification construction for reliable software systems, IEEE Trans. Reliability 65 (1) (2016) 88–106.

[35] J. Elamkulam, Z. Glazberg, I. Rabinovitz, G. Kowlali, S. C. Gupta, S. Kohli, S. Dattathrani, C. P. Macia, Detecting design flaws in UML state charts for embedded software, in: HVC, 2006, pp. 109–121.

[36] A. Pakonen, T. Matasniemi, J. Lahtinen, T. Karhela, A toolset for model checking of PLC software, in: ETFA, 2013, pp. 1–6.

[37] A. Pakonen, T. Tahvonen, M. Hartikainen, M. Pihlanko, Practical applications of model checking in the finnish nuclear industry, in: NPIC & HMIT, American Nuclear Society ANS, 2017, pp. 1342–1352.

[38] A. Pakonen, K. Björkman, Model checking as a protective method against spurious actuation of industrial control systems, in: ESREL, CRC Press, 2017, pp. 3189–3196.

[39] A. Pakonen, I. Buzhinsky, V. Vyatkin, Counterexample visualization and explanation for function block diagrams, in: INDIN, 2018, pp. 747–753.

[40] D. Ratiu, M. Gario, H. Schoenhaar, FASTEN: an open extensible framework to experiment with formal specification approaches: using language engineering to develop a multi-paradigm specification environment for nusmv, in: FormaliSE@ICSE, IEEE / ACM, 2019, pp. 41–50.

[41] A. Campetelli, M. Junker, B. Böhm, M. Davidich, V. Koutsoumpas, X. Zhu, J. C. Wehrstedt, A model-based approach to formal verification in early development phases: A desalination plant case study, in: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering, 2015, pp. 91–100.

[42] T. T. T. Nguyen, K. Ogata, Graphically perceiving characteristics of the MCS lock and model checking them, in: SOFL+MSVL, 2017, pp. 3–23.

[43] M. Kuntz, F. Leitner-Fischer, S. Leue, From probabilistic counterexamples via causality to fault trees, in: SAFECOMP, 2011, pp. 71–84.

[44] F. Leitner-Fischer, S. Leue, Causality checking for complex system models, in: VMCAI, 2013, pp. 248–267.

[45] F. Leitner-Fischer, S. Leue, Probabilistic fault tree synthesis using causality computation, IJCCBS 4 (2) (2013) 119–143.

[46] F. Leitner-Fischer, S. Leue, Spincause: a tool for causality checking, in: SPIN, 2014, pp. 117–120.

[47] D. A. Peled, A. Pnueli, L. D. Zuck, From falsification to verification, in: FST TCS, 2001, pp. 292–304.

[48] P. Gastin, P. Moro, M. Zeitoun, Minimization of counterexamples in SPIN, in: SPIN, 2004, pp. 92–108.

[49] M. Chechik, A. Gurfinkel, A framework for counterexample generation and exploration, STTT 9 (5-6) (2007) 429–445.

[50] K. Chang, V. Bertacco, I. L. Markov, Simulation-based bug trace minimization with bmc-based refinement, IEEE Trans. on CAD of Integrated Circuits and Systems 26 (1) (2007) 152–165.

[51] B. Luteberget, J. J. Camilleri, C. Johansen, G. Schneider, Participatory verification of railway infrastructure by representing regulations in railcnl, in: SEFM, 2017, pp. 87–103.

[52] K. Angelov, J. J. Camilleri, G. Schneider, A framework for conflict analysis of normative texts written in controlled natural language, J. Log. Algebr. Program. 82 (5-7) (2013) 216–240.

[53] L. Feng, M. Ghasemi, K. Chang, U. Topcu, Counterexamples for robotic planning explained in structured language, in: ICRA, IEEE, 2018, pp. 7292–7297.

[54] B. Luteberget, C. Johansen, Efficient verification of railway infrastructure designs against standard regulations, Formal Methods in System Design 52 (1) (2018) 1–32.

[55] A. W. Crapo, A. Moitra, C. McMillan, D. Russell, Requirements capture and analysis in ASSERT(TM), in: RE, 2017, pp. 283–291.

[56] A. Moitra, K. Siu, A. W. Crapo, H. R. Chamarthi, M. Durling, M. Li, H. Yu, P. Manolios, M. Meiners, Towards development of complete and conflict-free requirements, in: RE, 2018, pp. 286–296.

[57] A. Moitra, K. Siu, A. W. Crapo, M. Durling, M. Li, P. Manolios, M. Meiners, C. McMillan, Automating requirements analysis and test case generation, Requir. Eng. 24 (3) (2019) 341–364.

[58] A. Groce, D. Kroening, F. Lerda, Understanding counterexamples with explain, in: CAV, 2004, pp. 453–456.

[59] T. Ball, M. Naik, S. K. Rajamani, From symptom to cause: localizing errors in counterexample traces, in: SIGPLAN-SIGACT, 2003, pp. 97–105.

[60] E. M. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: TACAS, 2004, pp. 168–176.

[61] A. Groce, S. Chaki, D. Kroening, O. Strichman, Error explanation with distance metrics, STTT 8 (3) (2006) 229–247.

[62] F. Pu, Y. Zhang, Localizing program errors via slicing and reasoning, in: HASE, 2008, pp. 187–196.

[63] F. Hölzl, M. Feilkas, Autofocus 3 - A scientific tool prototype for model-based development of component-based, reactive, distributed systems, in: Model-Based Engineering of Embedded Real-Time Systems - International Dagstuhl Workshop, 2007, pp. 317–322.

[64] P. Arcaini, A. Gargantini, E. Riccobene, Nuseen: A tool framework for the nusmv model checker, in: ICST, 2017, pp. 476–483.

[65] M. L. Bolton, E. J. Bass, Using task analytic models to visualize model checker counterexamples, in: IEEE International Conference on Systems, Man and Cybernetics, 2010, pp. 2069–2074.

[66] K. Loer, M. D. Harrison, An integrated framework for the analysis of dependable interactive systems (IFADIS): its tool support and evaluation, Autom. Softw. Eng. 13 (4) (2006) 469–496.

[67] V. Schuppan, A. Biere, Shortest counterexamples for symbolic model checking of LTL with past, in: TACAS, 2005, pp. 493–509.

[68] C. Gerking, W. Schäfer, S. Dziwok, C. Heinzemann, Domain-specific model checking for cyber-physical systems, in: MoDeVVa@MoDELS, 2015, pp. 18–27.

[69] S. Shen, Y. Qin, S. Li, A faster counterexample minimization algorithm based on refutation analysis, in: DATE, 2005, pp. 672–677.

[70] F. Weitl, S. Nakajima, Incremental construction of counterexamples in model checking web documents, in: WWV, 2010, pp. 34–50.

[71] S. Shen, Y. Qin, S. Li, Counterexample minimization for actl, in: CHARME, Vol. 5, 2005, pp. 393–397.

[72] K. Heljanko, T. A. Junttila, M. Keinänen, M. Lange, T. Latvala, Bounded model checking for weak alternating büchi automata, in: CAV, 2006, pp. 95–108.

[73] P. Gastin, P. Moro, Minimal counterexample generation for SPIN, in: SPIN, 2007, pp. 24–38.

[74] S. Edelkamp, S. Leue, A. Lluch-Lafuente, Directed explicit-state model checking in the validation of communication protocols, STTT 5 (2-3) (2004) 247–267.

[75] H. Hansen, J. Geldenhuys, Cheap and small counterexamples, in: SEFM, 2008, pp. 53–62.

[76] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Directed explicit model checking with HSF-SPIN, in: SPIN, 2001, pp. 57–79.

[77] S. Shen, Y. Qin, S. Li, A fast counterexample minimization approach with refutation analysis and incremental SAT, in: ASP-DAC, 2005, pp. 451–454.

[78] F. Leitner-Fischer, S. Leue, On the synergy of probabilistic causality computation and causality checking, in: SPIN, 2013, pp. 246–263.

[79] H. Debbi, M. Bourahla, Generating diagnoses for probabilistic model checking using causality, CIT 21 (1) (2013) 13–23.

[80] E. Ábrahám, N. Jansen, R. Wimmer, J. Katoen, B. Becker, DTMC model checking by SCC reduction, in: QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, 2010, pp. 37–46.

[81] N. Jansen, E. Ábrahám, M. Volk, R. Wimmer, J. Katoen, B. Becker, The COMICS tool - computing minimal counterexamples for dtmcs, in: ATVA, Vol. 7561 of LNCS, Springer, 2012, pp. 349–353.

[82] H. Aljazzar, S. Leue, Debugging of dependability models using interactive visualization of counterexamples, in: QEST, 2008, pp. 189–198.

[83] K. Ravi, F. Somenzi, Minimal assignments for bounded model checking, in: TACAS, 2004, pp. 31–45.

[84] J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, S. Leue, Heuristic-guided counterexample search in FLAVERS, in: SIGSOFT, 2004, pp. 201–210.

[85] S. Edelkamp, F. Reffel, Obdds in heuristic search, in: KI-98: Advances in Artificial Intelligence, 1998, pp. 81–92.

[86] H. Hansen, A. Kervinen, Minimal counterexamples in o(n log n) memory and o(nˆ2) time, in: ACSD, 2006, pp. 133–142.

[87] H. Aljazzar, S. Leue, Directed explicit state-space search in the generation of counterexamples for stochastic model checking, IEEE Trans. Software Eng. 36 (1) (2010) 37–60.

[88] A. Groce, W. Visser, What went wrong: Explaining counterexamples, in: SPIN, 2003, pp. 121–135.

[89] S. Kanav, V. Aravantinos, Modular transformation from AF3 to nuxmv, in: MODELS, 2017, pp. 300–306.

[90] N. Jansen, E. Ábrahám, J. Katelaan, R. Wimmer, J. Katoen, B. Becker, Hierarchical counterexamples for discrete-time markov chains, in: ATVA, 2011, pp. 443–452.

[91] R. Wimmer, N. Jansen, E. Ábrahám, B. Becker, J. Katoen, Minimal critical subsystems for discrete-time markov models, in: TACAS, 2012, pp. 299–314.

[92] R. Wimmer, N. Jansen, E. Ábrahám, J. Katoen, B. Becker, Minimal counterexamples for linear-time probabilistic verification, Theor. Comput. Sci. 549 (2014) 61–100.

[93] G. Barbon, V. Leroy, G. Salaün, E. Yah, Visual debugging of behavioural models, in: ICSE (Companion Volume), IEEE / ACM, 2019, pp. 107–110.

[94] G. Barbon, V. Leroy, G. Salaün, Debugging of behavioural models with CLEAR, in: TACAS, 2019, pp. 386–392.

[95] G. Barbon, V. Leroy, G. Salaün, Counterexample simplification for liveness property violation, in: SEFM, 2018, pp. 173–188.

[96] A. P. Kaleeswaran, A. Nordmann, T. Vogel, L. Grunske, Counterexample interpretation for contract-based design, in: IMBSA, 2020, pp. 99–114.

[97] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, A. Stahlbauer, Witness validation and stepwise testification across software verifiers, in: ESEC/FSE, 2015, pp. 721–733.

[98] H. Jin, K. Ravi, F. Somenzi, Fate and free will in error traces, STTT 6 (2) (2004) 102–116.

[99] N. Kumar, V. Kumar, M. Viswanathan, On the complexity of error explanation, in: VMCAI, 2005, pp. 448–464.

[100] F. Copty, A. Irron, O. Weissberg, N. P. Kropp, G. Kamhi, Efficient debugging in a formal verification environment, STTT 4 (3) (2003) 335–348.

[101] S. Busard, C. Pecheur, Producing explanations for rich logics, in: FM, 2018, pp. 129–146.

[102] A. L. J. Dominguez, N. A. Day, Generating multiple diverse counterexamples for an efsm, Technical Report CS-2013–06, University of Waterloo (2013).

[103] T. Ball, S. K. Rajamani, The SLAM project: debugging system software via static analysis, in: SIGPLAN-SIGACT, 2002, pp. 1–3.

[104] I. Beer, S. Ben-David, H. Chockler, A. Orni, R. J. Trefler, Explaining counterexamples using causality, Formal Methods in System Design 40 (1) (2012) 20–40.

[105] J. Y. Halpern, J. Pearl, Causes and explanations: A structural-model approach. part i: Causes, The British journal for the philosophy of science 56 (4) (2005) 843–887.

[106] T. Bochot, P. Virelizier, H. Waeselynck, V. Wiels, Paths to property violation: A structural approach for analyzing counter-examples, in: HASE, 2010, pp. 74–83.

[107] A. W. Crapo, A. Moitra, Using OWL ontologies as a domain-specific language for capturing requirements for formal analysis and test case generation, in: ICSC, 2019, pp. 361–366.

[108] D. Kroening, M. Tautschnig, CBMC - C bounded model checker - (competition contribution), in: TACAS, Vol. 8413 of Lecture Notes in Computer Science, Springer, 2014, pp. 389–391.

[109] K. C. Castillos, H. Waeselynck, V. Wiels, Show me new counterexamples: A path-based approach, in: ICST, 2015, pp. 1–10.

[110] A. Campetelli, F. Hölzl, P. Neubeck, User-friendly model checking integration in model-based development, in: 24th International Conference on Computer Applications in Industry and Engineering, 2011.

[111] M. T. Aung, T. T. T. Nguyen, K. Ogata, Analysis of two flawed versions of A mutual exclusion protocol with maude and SMGA, in: ICSCA, 2018, pp. 194–198.

[112] Y. Phyo, K. Ogata, Analysis of some variants of the anderson array-based queuing mutual exclusion protocol with model checking and graphical animations, in: DSA, IEEE, 2018, pp. 126–135.

[113] S. Patil, V. Vyatkin, C. Pang, Counterexample-guided simulation framework for formal verification of flexible automation systems, in: INDIN, 2015, pp. 1192–1197.

[114] I. Schinz, T. Toben, C. Mrugalla, B. Westphal, The rhapsody UML verification environment, in: SEFM, 2004, pp. 174–183.

[115] D. Darvas, E. Blanco Vinuela, B. Fernández Adiego, Plcverif: A tool to verify plc programs based on model checking techniques, in: ICALEPCS, 2015, pp. 911–914.

[116] H. Aljazzar, F. Leitner-Fischer, S. Leue, D. Simeonov, Dipro - A tool for probabilistic counterexample generation, in: SPIN, 2011, pp. 183–187.

[117] E. M. Clarke, S. Jha, Y. Lu, H. Veith, Tree-like counterexamples in model checking, in: LICS, 2002, pp. 19–29.

[118] A. P. Sistla, Safety, liveness and fairness in temporal logic, Formal Asp. Comput. 6 (5) (1994) 495–512.

[119] A. Biere, C. Artho, V. Schuppan, Liveness checking as safety checking, Electr. Notes Theor. Comput. Sci. 66 (2) (2002) 160–177.

[120] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, UPPAAL in 1995, in: TACAS, 1996, pp. 431–434.

[121] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, STTT 1 (1-2) (1997) 134–152.

[122] K. L. McMillan, The smv system, in: Symbolic Model Checking, Springer, 1993, pp. 61–85.

[123] G. J. Holzmann, The model checker SPIN, IEEE Trans. Software Eng. 23 (5) (1997) 279–295.

[124] G. J. Holzmann, The SPIN Model Checker - primer and reference manual, Addison-Wesley, 2004.

[125] S. Eker, J. Meseguer, A. Sridharanarayanan, The maude LTL model checker, Electr. Notes Theor. Comput. Sci. 71 (2002) 162–187.

[126] S. Eker, J. Meseguer, A. Sridharanarayanan, The maude LTL model checker and its implementation, in: SPIN, 2003, pp. 230–234.

[127] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Vol. 4350 of Lecture Notes in Computer Science, Springer, 2007.

[128] T. T. T. Nguyen, K. Ogata, Graphical animations of state machines, in: DASC/PiCom/DataCom/CyberSciTech, 2017, pp. 604–611.

[129] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, T. Villa, VIS: A system for verification and synthesis, in: CAV, 1996, pp. 428–432.

[130] S. Jeong, J. Yoo, S. D. Cha, VIS analyzer: A visual assistant for VIS verification and analysis, in: ISORC, 2010, pp. 250–254.

[131] H. R. Chamarthi, P. C. Dillinger, P. Manolios, D. Vroon, The ACL2 sedan theorem proving system, in: TACAS, Vol. 6605 of Lecture Notes in Computer Science, Springer, 2011, pp. 291–295.

[132] M. Kaufmann, P. Manolios, J. S. Moore, Computer-aided reasoning: ACL2 case studies, Vol. 4, Springer Science & Business Media, 2013.

[133] A. Hinton, M. Z. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems, in: TACAS, 2006, pp. 441–444.

[134] M. Z. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: CAV, 2011, pp. 585–591.

[135] J. Katoen, M. Khattri, I. S. Zapreev, A markov reward model checker, in: QEST, 2005, pp. 243–244.

[136] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, D. N. Jansen, The ins and outs of the probabilistic model checker MRMC, Perform. Eval. 68 (2) (2011) 90–104.

[137] M. de Jonge, T. C. Ruys, The spinja model checker, in: SPIN, 2010, pp. 124–128.

[138] M. Fröhlich, M. Werner, Demonstration of the interactive graph-visualization system *da Vinci*, in: DIMACS, 1994, pp. 266–269.

[139] A. Gurfinkel, M. Chechik, B. Devereux, Temporal logic query checking: A tool for model exploration, IEEE Trans. Software Eng. 29 (10) (2003) 898–914.

[140] A. Gurfinkel, M. Chechik, Multi-valued model checking via classical model checking, in: CONCUR, 2003, pp. 263–277.

[141] W. Visser, M. B. Dwyer, M. W. Whalen, The hidden models of model checking, Software and Systems Modeling 11 (4) (2012) 541–555.

[142] H. Goldsby, B. H. C. Cheng, S. Konrad, S. Kamdoum, A visualization framework for the modeling and formal analysis of high assurance systems, in: MoDELS, 2006, pp. 707–721.

[143] Z. Brezocnik, B. Vlaovic, A. Vreze, SpinRCP: the eclipse rich client platform integrated development environment for the spin model checker, in: SPIN, 2014, pp. 125–128.

[144] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, S. Shoham, Ivy: safety verification by interactive generalization, in: PLDI, 2016, pp. 614–630.

[145] Y. Zhao, X. Jin, G. Ciardo, A symbolic algorithm for shortest EG witness generation, in: TASE, 2011, pp. 68–75.

[146] S. Biallas, N. Friedrich, H. Simon, S. Kowalewski, Automatic error cause localization of faulty plc programs, IFAC-PapersOnLine 48 (7) (2015) 79–84.

[147] A. Pakonen, I. Buzhinsky, K. Björkman, Model checking reveals design issues leading to spurious actuation of nuclear instrumentation and control systems, Reliab. Eng. Syst. Saf. 205 (2021) 107237.

[148] Z. Zheng, J. Tian, T. Zhao, Refining operation guidelines with model-checking-aided FRAM to improve manufacturing processes: a case study for aeroengine blade forging, Cognition, Technology & Work 18 (4) (2016) 777–791.

[149] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, S. Leue, Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples, in: QEST, 2009, pp. 299–308.

[150] H. Collavizza, N. L. Vinh, O. Ponsini, M. Rueher, A. Rollet, Constraint-based BMC: a backjumping strategy, STTT 16 (1) (2014) 103–121.

[151] P. Ovsiannikova, I. Buzhinsky, A. Pakonen, V. Vyatkin, Visual counterexample explanation for model checking with OERITTE, in: Y. Li, A. W. Liew (Eds.), ICECCS, IEEE, 2020, pp. 1–10.

[152] D. Beyer, Software verification and verifiable witnesses - (report on SV-COMP 2015), in: TACAS, 2015, pp. 401–416.

[153] S. Shen, Y. Qin, S. Li, Localizing errors in counterexample with iteratively witness searching, in: ATVA, 2004, pp. 456–469.

[154] T. Kumazawa, T. Tamai, Counterexample-based error localization of behavior models, in: NFM, 2011, pp. 222–236.

[155] E. M. Clarke, J. M. Wing, Formal methods: State of the art and future directions, ACM Comput. Surv. 28 (4) (1996) 626–643.