IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 February 2025, accepted 17 February 2025, date of publication 14 March 2025, date of current version 25 March 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3551607

SURVEY

A Systematic Literature Review on Mining LTL Specifications

SAMUELE GERMINIANI[1,2], (Member, IEEE), DANIELE NICOLETTI[3], AND GRAZIANO PRAVADELLI[1], (Senior Member, IEEE)

[1]Department of Engineering for Innovation Medicine, University of Verona, 37134 Verona, Italy
[2]Department of Engineering, University of Guglielmo Marconi, 00193 Rome, Italy
[3]Department of Computer Science, University of Verona, 37134 Verona, Italy

Corresponding author: Samuele Germiniani (samuele.germiniani@univr.it; s.germiniani@unimarconi.it)

This study was partially carried out within the PNRR research activities of the consortium iNEST (Interconnected North-East Innovation Ecosystem) funded by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS 00000043), CUP: B43C22000450006.

ABSTRACT Linear Temporal Logic (LTL) specifications play a crucial role in the verification process of cyber-physical systems, increasing the guarantees of their correctness. These specifications are vital for ensuring that both hardware and software components behave as expected, especially in complex real-world scenarios. In the last decades, researchers have developed several methodologies and tools to automatically generate LTL specifications, creating an urgent need to organize and synthesize existing literature to ease entry into this field and guide future research efforts. Therefore, starting from a pool of over 3000 papers extracted from the Scopus database in the temporal range 2000–2024, this paper employs the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology to produce a systematic review of mining LTL specifications of hardware and software systems. In particular, we provide a taxonomy of the methods and describe with significant detail all the relevant techniques present at the state of the art. Finally, we discuss the challenges of mining LTL specifications and explore potential directions and opportunities for future research.

INDEX TERMS Automata learning, linear temporal logic, specification mining, SVA generation, API rule inference, software reliability, assertion mining, behavior detection, specification, property discovery.

I. INTRODUCTION

In an era where electronic systems govern everything from critical infrastructure to personal devices, the reliability of these systems has become paramount. Yet, ensuring that such systems work as intended remains one of the most challenging aspects of development. To determine what is expected (or not) from a system, engineers must formalize the system's requirements in a way that is both precise and unambiguous: this is the role of formal specifications. Formal specifications are a cornerstone in the design, development, and verification of modern hardware and software systems. During system design, they enable the precise expression of requirements; during implementation, they serve as a foundation for formal verification, ensuring that critical components meet their specifications; finally, during execution, they assist in monitoring system behaviors to detect and react to violations.

Unfortunately, specification definition is a time-consuming and error-prone task, which requires high expertise to reason in terms of logic formulas [1].

To address these challenges, the field has seen significant advancements in **specification mining**—a technique that automates the generation of formal specifications. This approach leverages data from execution traces, source code, or informal specifications to derive meaningful properties that would be difficult to formulate manually.

Specification mining serves several important purposes. The mined specifications are compared against the initial specifications to verify if all expected behaviors have been implemented in the Design Under Verification (DUV). Furthermore, by analyzing the mined specifications, the verification engineer can discover the presence of unexpected behaviors caused by design errors or malicious code deliberately inserted in the DUV. Finally, mined specifications

The associate editor coordinating the review of this manuscript and approving it for publication was Khursheed Aurangzeb.

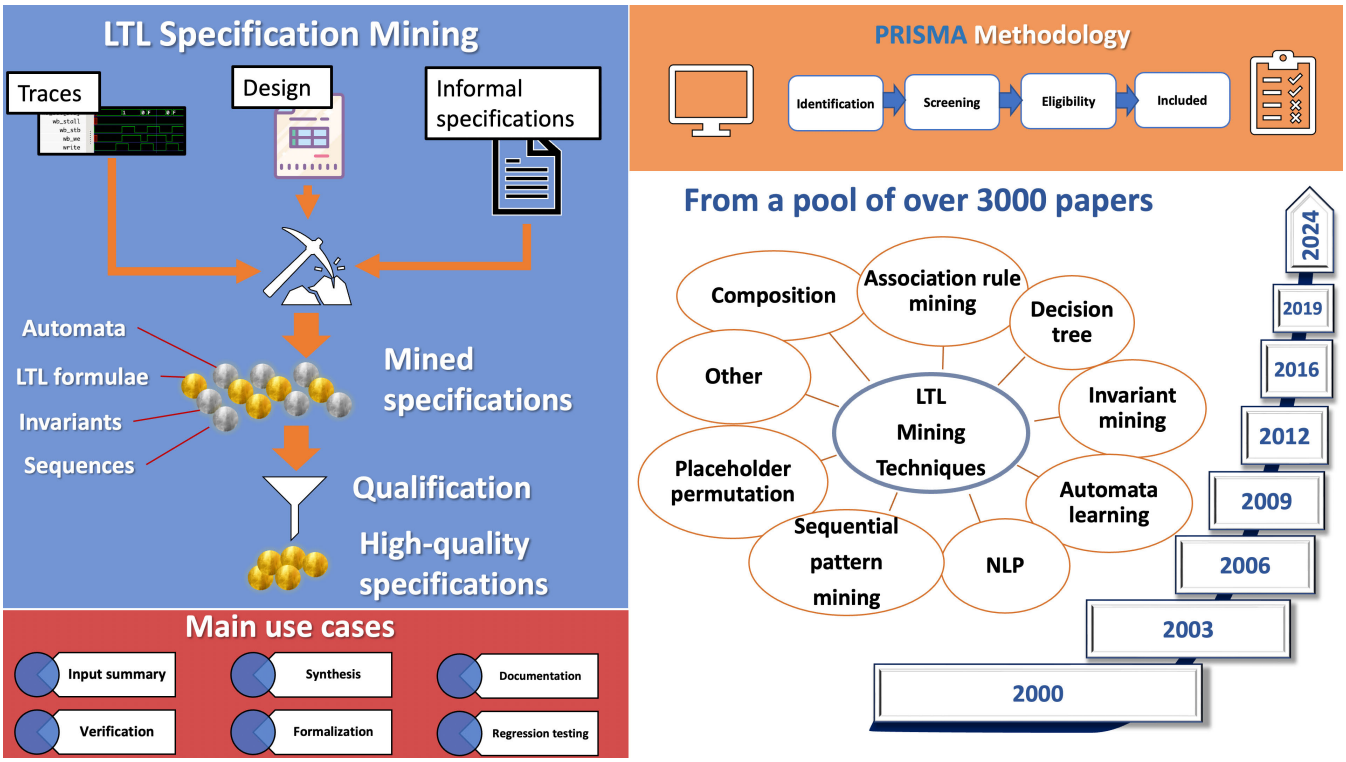


FIGURE 1. Graphical abstract.

can also be used for documentation purposes (we report the full list of use cases in section V-A). Linear Temporal Logic (LTL) is one of the most commonly used formalisms for expressing properties of Hardware (HW) and Software (SW) systems, particularly those that evolve over time. LTL allows for the specification of behaviors that must occur over sequences of events, making it an ideal candidate for expressing requirements related to safety and liveness temporal properties.

In recent years, several papers have been published to automatically generate LTL specifications. As a consequence, it has become increasingly urgent to organize and summarize the current literature to lower the entry barrier of this field of study and to quickly formulate a plan for future research.

A. CONTRIBUTIONS

Starting from a pool of over 3000 papers, we provide a systematic review of the techniques present in the literature for mining LTL specifications for hardware and software systems. In particular, we discuss and organize the possible use cases of specification mining; we provide a meaningful taxonomy of the mining techniques present in the literature; then, we describe those techniques with a sufficient level of detail to depict a clear and precise idea of the state-of-the-art. Additionally, we mention and cite the repositories of all the available open-source tools implementing the techniques described in this review. Finally, we identify key challenges of the current techniques and provide several insights into future research directions.

B. PAPER ORGANIZATION

The paper is organized as follows. Section II describes the methodology we followed to select, screen, and review the papers. Section III reports other reviews and survey papers related to ours. Section IV contains a set of preliminary definitions to formalize and clarify background concepts important for understanding the content of the paper. In section V, we describe the classification scheme of the techniques and the main use cases of LTL specification mining. Section VI contains a detailed description of the mining techniques. In section VII, we describe the techniques used to evaluate the quality of the mined specifications. In section VIII, we discourse on the main challenges of specification mining and the possible future research directions. Finally, in section IX, we draw our conclusions.

II. METHODOLOGY

This section describes the methodology we followed to select and review the papers.

We selected the papers to be included in the review following the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology [2]. The PRISMA methodology was employed in this study to ensure a transparent, comprehensive, and replicable systematic review process. PRISMA provides a structured

framework that guides the reporting of systematic reviews and meta-analyses across key stages. PRISMA ensures that the methods section thoroughly documents critical aspects such as the eligibility criteria, information sources, search strategy, and study selection.

A. SCOPE OF THIS REVIEW

The scope of this review includes all techniques that aim at automatically generating LTL specifications in HW and SW systems. However, we did not narrow the scope to only classical LTL specifications; instead, we also included its extensions, fragments, and other typical formalisms that have similar expressive power. In particular, we included Sequential Extended Regular Expression (SERE), which is widely used to formalize sequential behaviors, especially in the HW domain; SEREs are supported by two major specification languages called Property Specification Language (PSL) and SystemVerilog Assertion (SVA). SERE+LTL allows us to easily define three classes of temporal behaviors. I) *Non-temporal specifications*: they do not have a temporal behavior, often called invariants or immediate specifications. An example of this type would be the ''assert'' function in the C language, which checks the validity of a condition at a specific point in time without considering any temporal evolution. II) *Untimed sequences*: they define the ordering of events without specifying the temporal distance between such events. An example of this is specifying that ''event A must happen before event B'' but without imposing any constraint on how much time or which steps separate these two events. In hardware verification, one might specify that a signal must stabilize before a reset occurs without concern for the exact time interval between these events. III) *Timed sequences*: they specify both the ordering and the temporal distance between events to allow more precise control over the temporal behavior of the DUV. For example, you might require that a variable change occurs exactly within a given number of ''instants'' after a previous event.

Considering the expressive power of LTL+SERE (as portrayed in the Chomsky hierarchic), it is natural to also include automata, particularly Büchi, finite-state and non-deterministic finite-sate automata, which offer comparable and often interchangeable expressive power for specifying temporal and sequential behaviors. Although this review encompasses all techniques that produce a certain output type, we did not discriminate for the input type. Nonetheless, the included papers only treated three main kinds of input: the model of the DUV, for example, its source code; the execution/simulations traces of a DUV, such as a time series or a data log; prior specifications of a DUV, such as informal specifications written in English or formal specifications written in LTL+SERE. We formalize all these aspects in section IV.

B. SELECTION OF DATABASE AND SEARCH STRING

Given the scope outlined in the previous section, the first step of the review is the collection of articles. We searched the papers using the *Scopus* [3] extensive database, which covers over 75 million records, including peer-reviewed journals, conference proceedings, and books across multiple disciplines. Because of its stringent content selection policies, ensuring that only the highest quality content gets indexed, and its powerful and flexible search functionalities, Scopus is often considered the standard de facto for navigating scientific literature. Other popular databases in the area of engineering and computer science are *Web of Science* [4], which offers a comprehensive collection of scientific publications and citation data; *IEEE Xplore* [5], which specializes in engineering and technology; *Google Scholar* [6], a freely accessible database that aggregates research across various disciplines. Each of these databases offers distinct advantages depending on the field of research; however, Scopus seems to inspire the highest consensus in the scientific community.

After choosing the database, we defined a list of keywords related to the ''Mining of LTL specifications in HW and SW systems'' to collect as many related articles as possible. These keywords include ''specification, assertion, requirement, property, formula, invariant, behavior, LTL, PSL, SVA, SERE, mining, generating, inferring, learning, discovery, detection''. Then, a search string was formed by combining the aforementioned keywords by using *AND* and *OR* boolean operators according to the syntax of the Scopus search guide [7] as reported below.

(specification OR assertion OR requirement OR property
OR formula OR invariant OR behaviour OR behavior)
AND
(mine OR mining OR miner OR generate OR generating
OR generation OR generator OR infer OR inferring
OR inference OR learn OR learning OR learner OR discover
OR discovery OR detect OR detection OR detector)
AND
(ltl OR {linear temporal logic} OR psl OR {Property
Specification Language} OR sva OR {System Verilog
Assertion} OR sere OR {Sequential Extended Regular
Expressions})

The sentences inside the curly brackets, such as {linear temporal logic}, find only documents that contain that exact sequence of words. Furthermore, we used several variations of verb keywords that can be used in a paper in several tenses.

We decided not to restrict the temporal window of the searched papers; as a consequence, the earliest reviewed paper was published in 1972, while the latest one was published in early 2025. However, we consider our work a literature review covering the range 2000-2024, as the first discussed relevant papers were published in the early 2000s, while only the first month of 2025 is covered.

We restricted the search to papers written in English and belonging to the scientific areas of computer science

"COMP", engineering "ENGI", or math "MATH". Moreover, the search is performed only on titles, abstracts, and keywords of the records in the databases. The initial number of articles found by this query was 1491.

C. SCREENING PROCESS

We uploaded the papers found with the query to a collaborative platform called Rayyan [8], a tool for systematically screening a large volume of papers. First, we exploited Rayyan's filtering features to remove duplicate papers. Then, we identified the most relevant papers based on their alignment with the target topic. To achieve this, we initially screened the papers by reviewing their titles and abstracts to remove out-of-scope papers. This process removed several papers on related topics such as model-checking, controller synthesis, synthesis of checkers and monitors, and model-driven design, which involve formal specifications but do not focus on mining specifications. Other papers described mining specification approaches but focused on formalisms not covered by our review, such as Signal Temporal Logic (STL) and Computation Tree Logic (CTL). Another notable set of out-of-scope papers is those describing "process mining" techniques, as they belong to a completely different research community and have different goals, even though some of their techniques overlap with those reported in this review. After this first manual selection process, only 83 papers (P_0) were deemed relevant. At this stage, we diverged from the classical PRISMA and applied a new approach: we included in the reviewing process all records cited by the papers in P_0 and all the records that cite the papers in P_0. To retrieve those relations, we developed a Python script that exploits the Scopus Application Programming Interfaces (APIs) [9], while the initial search was performed directly from the Scopus website. In practice, the papers in P_0 become the seed for a wider unconstrained search net based on the expertise of authors of the papers in P_0 and those citing P_0. This step allows us to include relevant papers that might have been excluded by the initial query due to unconventional naming of the mining process or using non-standard formalism that can be interpreted as fragments of LTL+SERE. That process added 1672 papers to be manually reviewed with the same approach, resulting in 75 (P_1) new relevant papers. Therefore, the total number of relevant papers that required further evaluation became 158 ($P_0 + P_1$). After reading the content of all 158 papers, we manually rejected an additional 48 papers following the same criteria as before. Therefore, this review's final number of papers is 110 (P_{final}).

D. REVIEW PROCESS

For each paper in P_{final}, we compiled a row of a spreadsheet with the following columns: "doi, Year, N. citations, Authors, Title, Tool, Domain, Use case, Input, Output, Mining Technique, Qualification, Results". The first row of the spreadsheet contains the self-explanatory columns "doi, Year, N. citations, Authors, Title" already compiled with the data of the papers in P_{final}. The rest of the columns are explained below.

- *Tool*: refers to the name and repository link of any software tool mentioned in the paper that is used to implement the proposed mining technique.
- *Domain*: specifies the application domain to which the mining technique is applied, such as "HW" or "SW".
- *Use case*: describes the specific problem or scenario in which the mining technique is applied within the paper, providing context for its practical usage (see section V-A).
- *Input*: refers to the types of data or models the mining technique consumes to produce results, such as execution traces.
- *Output*: details the form of the results produced by the mining technique, for example, LTL formulas, automata, or other types of formal specifications.
- *Mining Technique*: describes the mining technique or approach used in the paper.
- *Qualification*: refers to the techniques used to qualify the mined specifications.
- *Results*: summarizes how the authors observe their findings of the paper, focusing on the effectiveness of the mining technique, including performance metrics, empirical validation, or comparative analyses.

The columns in the spreadsheet are filled with tags: unique short phrases or words used to describe a specific concept, idea, or technique in the paper; a separate sheet contains each tag's description. For fields that require multiple tags, each tag is separated by a pipe symbol in the form $tag_1|tag_2 \mid \ldots \mid tag_n$, allowing several tags to be listed within the same spreadsheet cell. The tagging system uses "Concept trees" (stored in a different sheet), which are structured hierarchies that organize tags to reflect the relationships between broader and more specific concepts. They help to maintain a logical and consistent organization of tags, ensuring that each tag belongs to a meaningful category and simplifying the navigation of the available tags. For example, consider the "Output" column. The concept tree for this column might begin with a broad tag like "LTL specification," which can then branch into more specific tags such as "LTL only next" or "SERE".

The tags used in the "Mining Technique" column are an exception, as they do not use concept trees due to the lack of a clear hierarchical nature of the mining techniques. Instead, this kind of tag requires a set of "technique features" reported in description V to add more information on the nature of the reported technique. These tags follow a specific format to convey additional details: they can include values for the technique features, which are added as a comma-separated list inside parentheses. The general form of such a tag is mining_tech_tag(feature1, feature2, …, featureN). For example, if a tag describes a mining technique called "decision_tree" that is dynamic and supervised, the tag could be written as decision_tree(Supervised, Dynamic).

At the end of the process, we gathered enough data to effectively classify and describe the discovered mining techniques, as reported in section VI.

III. RELATED WORK

Over the years, a few authors have already attempted to tackle the daunting task of navigating the landscape of specification mining techniques.

The paper that overlaps most with ours is [10], in which Robillard et al. provided a comprehensive survey of over a decade of research on techniques for automatically inferring properties of Application Programming Interfaces (APIs). The paper synthesizes and classifies more than 60 techniques into five categories, depending on the type of mined specification: unordered usage patterns, sequential usage patterns, behavioral specifications, migration mappings, and general information. We report a more detailed description below to emphasize the difference between their work and ours.

- *Unordered Usage Patterns* specify which API elements (e.g., methods or classes) are commonly used together, without considering the order in which they should be invoked.
- *Sequential Usage Patterns* focus on the correct order in which API methods should be invoked. Sequential patterns specify temporal constraints on API usage, ensuring that certain API calls are made in the correct sequence (e.g., open() before read()).
- *Behavioral Specifications* describe the expected behavior of an API, that, under certain conditions, may lead to specific states or errors, such as specifying that calling a method without a prerequisite (e.g., read() without open()) leads to an exception.
- *Migration Mappings* specify mappings between different versions of an API, helping developers understand how to migrate from one version to another. They provide information on deprecated or replaced API elements, ensuring compatibility with newer versions.
- *General Information* includes various idiosyncratic properties of APIs, such as constraints, invariants, or performance-related rules that don't fall into the other categories. These are often inferred to assist in improving API documentation or for specific developer tasks.

The ultimate objective of the papers described in the survey is to offer developers guidance on the correct use of complex APIs by discovering latent and undocumented properties, which can help avoid misuse, improve API documentation, and support tasks like bug detection, code migration, and API usage optimization. Their review covers the papers published in the range 2000 to 2010. On the contrary, our paper does not have a strict temporal constraint and covers all papers published before February 2025. Furthermore, we follow a different review process based on the PRISMA methodology, and we classify the reviewed papers in a completely different manner (see section V) while focusing only on LTL specifications and other formalisms with similar expressive power.

In [11], Krka et al. evaluate and compare different techniques for automatically mining specifications of software libraries by inferring finite state automata. Specifically, the paper aims to assess the impact of four strategies (using execution traces, invariants, or a combination of both) on the quality (precision and recall) of the inferred models. The relevant strategies related to LTL specifications are also included in our review.

In [12], Ghafari and Tjortjis proposed a survey of heuristic approaches for association rule mining. The paper concludes that some algorithms like SRmining, PMES, Ant-Association Rule Mining (ARM), and MDS-H perform fastest, while others like HSBO-TS are the most complete. Additionally, the paper introduces a ranking parameter called GT-Rank to evaluate the best heuristic ARM approaches, with ARMGA, ASC, and Kua emerging as top-performing methods. The authors further explore trends in heuristic ARM by considering algorithms and their characteristics as transactional data to generate association rules. Some of the techniques covered by this paper are reported in section VI-C of our review.

The authors of [13] try to evaluate the effectiveness of existing specification mining algorithms in a real-world industrial setting, particularly in the context of debugging large-scale embedded software systems. The authors aim to assess the practical usability of these algorithms, propose potential improvements, and discuss challenges faced during their implementation in an industrial context. The authors reach similar conclusions to ours when discussing the practical limitations of the current mining approaches.

The paper in [14] aims to address the test oracle problem in Cyber-Physical Systems (CPSs) by developing a framework to evaluate specification mining techniques and comparing various methods based on their ability to meet CPS-specific constraints. The test oracle problem refers to the challenge of determining whether a system's output is correct during testing, especially when explicit specifications are lacking, or human verification is costly and prone to errors. The paper seeks to find automated solutions to this issue by reviewing and analyzing existing techniques. The authors claim to be the first to address that particular problem; however, the paper suffers from the lack of a rigorous selection and review process.

The paper in [15] evaluates and compares existing trace mining methods in the context of communication-centric system-on-chip traces. The authors investigate the strengths and weaknesses of seven well-known trace mining methods from both hardware and software domains. The aim is to assess their ability to handle complex and concurrent System on a chip (SoC) traces, evaluate their performance, and determine their usefulness in mining meaningful patterns that could be used for SoC validation. Almost all the approaches of this paper are covered by our review.

Neider and Roy [16] recently published a survey on mining LTL specifications from examples of desired and undesired behaviors. Differently from our work, which aims to provide a comprehensive review of past and recent literature on LTL mining (including automata, invariants, and sequences), their paper focuses only on recent literature with the goal of providing a general understanding of the main techniques, especially those that employ formal methods. Most of their references are also contained and discussed in this paper; however, they provide a more in-depth analysis of techniques related to constraint solving and neural network training.

Finally, Bartocci et al. [17] provide a comprehensive overview of methods for mining STL specifications from cyber-physical systems behaviors. The paper aims to categorize and explain different approaches used in the field of STL specification mining, presenting the key techniques and tools, and examining their applications. This paper aims for a similar goal as our paper but for STL specifications; however, such formalism is out of the scope of our review.

IV. PRELIMINARIES

In this section, we formalize expressions that are useful for clearly understanding the technical content of the paper.

A. TRACE, DUV, AND INFORMAL SPECIFICATIONS

The techniques reported in section VI take three kinds of input: a trace, the DUV, and informal specifications.

Definition 1: *A **trace** can generally appear in two main forms: a log or a time series.*

- *A **log** is a finite sequence of events $\langle e_1, \ldots, e_n \rangle$, where each event e_i represents a discrete occurrence at a specific time or state. Examples of events are function calls, Central Processing Unit (CPU) instructions execution, I/O operations, and state transitions (as in state machines). In general, any observable system activity can be considered an event. A timestamp may be associated with each event, but it is not always present. Events can also include optional metadata, such as variable values or parameters passed during function calls, that capture the system's state at the time of the event.*
- *Given a finite sequence of time units $\langle t_1, \ldots, t_n \rangle$ and a set of variables $\{v^1, \ldots, v^m\}$, a **time series** is a sequence of tuples $(t_i, v_i^1, \ldots, v_i^m)$ such that v_i^j is the value assumed by variable v^j at time t_i.*

The main difference between a time series and a log is that a time series always contains a set of variables whose values are known at each time unit. Time series are often used in HW settings where the value of each variable is sampled at a constant rate by a clock. Logs are more common in SW settings where time is not well-defined or only the order of events matters.

We will refer to "traces", "execution traces," "simulation traces," "logs," and "event sequences" interchangeably. It will be clear from the context if the input of the technique is a log or time series; in some cases, it will not matter for the sake of the description which of one of the two it is.

Definition 2: *The **DUV** refers to the hardware or software (or any other abstraction that can be executed or simulated) model used to extract the specifications.*

In hardware systems, the DUV is often a digital circuit, such as a processor or controller, typically represented as a hardware description language (HDL) model, e.g., in Verilog or VHSIC Hardware Description Language (VHDL). In software systems, the DUV might be a component, module, or full program written in C, Python, Java, or any other high-level language. In both cases, the DUV usually involves the presence of its source code; however, the source code is sometimes unavailable, and the DUV is only used to observe its output after simulation or execution. We will refer to "design", "DUV", "system", "SUV" (System Under Verification), "model" and "source code" interchangeably.

Definition 3: ***Informal Specifications** are descriptions of the expected system behavior typically written in English or other non-formal formats. They capture the requirements or constraints imposed on the system's functionality, safety, or performance but may lack the precision and rigor required for formal verification. By their nature, informal specifications do not have a well-defined formal structure.*

B. FORMAL SPECIFICATIONS

The formal specifications employed in this paper can be classified into three categories: non-temporal, temporal, and automata.

In the paper, the generic term "specification" is used to denote a formal specification. Depending on the context, we use other terms in place of "specification" with slightly different connotations. These terms are "assertion", "property", "formula", "invariant", "automaton" and "behavior".

Non-temporal specification: often called immediate specification or simply "invariant". It is usually implemented in both HW and SW as a simple *assert*(p) function defined inside the source code of the design; it checks if the propositional formula p is satisfied when *assert* is called during execution.

Definition 4: *A **proposition** is a Boolean expression that can be constructed by using Boolean operators (&&, ||, !, $\rightarrow$) between Boolean expressions or relational operators (<, > , >=, <=, ==, ! =) between numeric expressions. Numeric expressions are constructed by using arithmetic operators ($+, -, *, /$) or bitwise operators (&, |, $\sim$, $\gg$, $\ll$). Boolean constants and DUV variables are propositions. Numeric constants and DUV variables are numeric expressions.*

Depending on the context, we will use &&, &, ||, |, !, >=, <=, == when referring to domain-specific operators of programming languages (or other domain-specific contexts) such as C, Verilog, Java or Regex. In all other cases, we will use the more traditional and readable $\wedge$, $\vee$, $\neg$, $\geq$, $\leq$, $=$. The operators $\rightarrow$, $\Rightarrow$, $\mapsto$ will always refer to the logical implication, except when specified otherwise.

Temporal specifications: the formula is formalized using temporal logic. The truth value of the formula is usually checked independently from the execution of the design. Temporal logic is expressive enough to represent most properties that a finite-state system needs to satisfy. Therefore, it is usually the first choice for defining specifications. The most popular languages to formalize temporal specifications are LTL and CTL. LTL can describe properties of individual executions starting from a set of initial states and the semantics is defined as a set of paths; on the other hand, CTL describes properties of a computation tree: formulas can reason about many executions at once, and the semantics is defined in terms of states and paths. CTL logic cannot be used to check specifications with semi-formal techniques involving simulation; instead, this logic is mostly used with formal techniques such as model checking. This review will focus on LTL.

Definition 5: ***Linear temporal logic*** *is a modal, temporal logic used to formalize behaviors spanning multiple instants of time. In LTL, one can encode formulas about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, and so on. Given a finite set of propositions P, the set of LTL formulas over P can be defined, in negation normal form, as follows*:

- $a \in P$ *and* $\neg a$ *are LTL formulas;*
- *if* ϕ_1 *and* ϕ_2 *are LTL formulas then* $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $X\ \phi_1$, $\phi_1 U \phi_2$, $\phi_1 R \phi_2$, $G\ \phi_1$ *and* $F\ \phi_1$ *are LTL formulas.*

Intuitively, the semantics of temporal operators X (next), U (until), R (release), G (always) and F (eventually) is:

- $X\ \phi_1$ *holds at time* t *if* ϕ_1 *holds at time* $t+1$*;*
- $\phi_1 U \phi_2$ *holds at time* t *if* ϕ_1 *holds for all instants* $t' \geq t$ *until* ϕ_2 *holds;*
- $\phi_1 R \phi_2$ *holds at time* t *if* ϕ_2 *holds for all instants* $t' \geq t$ *until and including the instant where* ϕ_1 *first becomes true; if* ϕ_1 *never becomes true,* ϕ_2 *holds forever.*
- $G\ \phi_1$ *holds at time* t *if* ϕ_1 *holds at all instants* $t' \geq t$*. In other words,* ϕ_1 *is true globally in the future.*
- $F\ \phi_1$ *holds at time* t *if* ϕ_1 *holds at some instant* $t' \geq t$*. In other words,* ϕ_1 *eventually becomes true in the future.*

In this paper, we will mostly use well-known LTL operators, such as *Next (X)*, *Until (U)*, *Always (G)* and *Eventually (F)*. Sometimes, we will use a slightly different syntax; for example, we will use the SystemVerilog Assertion syntax; therefore, "nexttime" (SystemVerilog syntax) instead of "X" or "always" and "eventually" instead of "G" and "F". In all cases, it will be clear what LTL operator we are referring to.

Definition 6: ***Linear Temporal Logic on finite traces (LTLf)*** *adapts classical LTL to finite traces. While LTL is designed to reason about infinite sequences, LTLf is tailored to specify properties over finite paths. The syntax of LTLf is similar to LTL, using the same set of logical and temporal operators. However, the semantics of the temporal operators adapt to finite traces*:

- $X\ \phi$ *in LTLf holds at time* t *if* ϕ *holds at time* $t+1$*, and* $t+1$ *exists in the trace.*
- $\phi U \psi$ *holds at time* t *if there exists a time* $t' \geq t$ *such that* ψ *holds at* t'*, and for all* t'' *with* $t \leq t'' < t'$*,* ϕ *holds. Crucially,* t' *must be within the bounds of the trace.*
- $\phi R \psi$ *holds at time* t *if* ψ *holds from* t *onwards or until* ϕ *holds, and if* ϕ *never holds,* ψ *must hold through the end of the trace.*
- $G\ \phi$ *requires* ϕ *to hold at all times from* t *to the end of the trace.*
- $F\ \phi$ *holds at time* t *if* ϕ *holds at some* $t' \geq t$*, where* t' *is within the trace.*

These adaptations ensure that LTLf remains meaningful when the trace is not infinite, addressing scenarios specific to finite sequences where classical LTL might over-specify. Consider the following example in which we compare the interpretation of the formula $G(a \rightarrow Fb)$ in LTL and LTLf. In LTL, $G(a \rightarrow Fb)$ asserts that on any infinite trace, whenever a occurs, b must eventually follow at some future point. The operator G requires the condition to hold continuously at every point along an infinite trace, and $a \rightarrow Fb$ means that if a is true at any point, there must exist some future point where b is true. If b never occurs after a, the specification holds. However, when considering the LTLf semantics, if a occurs, b must follow before the trace concludes; otherwise, the specification does not hold. In the specification mining context, if classical LTL is employed on finite traces, then the above specification might be mined even if b never occurs. However, even when using infinite semantics, such an issue can be avoided by discarding the specifications with no "coverage", that is, with no activations brought to completion on the input traces. Nonetheless, employing infinite semantics on finite traces might be useful to address the problem of incomplete or imperfect traces. Continuing from the example above, if a is followed by b several times on the input traces but, due to incompleteness, b does not follow on the last occurrence of a, then the miner might discard a "good" specification when using LTLf. At the same time, LTL might produce false specifications if the DUV contained accidental or exceptional behaviors where a is not followed by b. In such cases, classical LTL, with its infinite semantics, could assume that b might eventually occur beyond the observed trace, thereby generating potentially misleading specifications.

Since the content of this paper does not benefit from distinguishing between LTL and LTLf we will use the simpler LTL for both.

LTL is often extended with SEREs, which are used to formalize sequences of events over time. This extension is commonly found in languages such as PSL and SVA.

Definition 7: *A* ***Regular Expression****, often abbreviated as regex, is a sequence of characters typically used for string pattern matching; a regex can define sequences based on specific character patterns. It consists of literals and*

metacharacters that form rules for searching or manipulating text, such as those reported below.

- *Literal characters: these are characters that match themselves, e.g., a, b, 1.*
- *Wildcards: special characters representing one or more characters, e.g., . (dot) matches any character.*
- *Quantifiers: define how many times an element must appear, e.g., * (zero or more), + (one or more), ? (zero or one).*
- *Character classes: define sets of characters, e.g., [a-z] matches any lowercase letter.*

Definition 8: *A **SERE** is essentially a regular expression over sequences of propositions, which can be combined with LTL operators to specify rich temporal properties. For example, one can specify that a certain sequence of conditions must eventually occur or that a sequence must hold continuously over time. We refer to [18] and [19] for a comprehensive reference on the syntax and semantics of SERE logic. In this paper, we primarily focus on LTL operators, but SEREs are introduced when necessary to capture more complex behaviors.*

Definition 9: *In some cases, the exact timing between events is irrelevant or unknown, and we are only concerned with the order in which the events occur. An **Untimed Sequence** is a sequence of events where the temporal order of occurrence is specified, but the precise time intervals between these events are not considered. This concept is useful when specifying behaviors or properties where the timing between events does not affect the correctness of the system, only their relative order. Untimed sequences are particularly useful in software systems, where events occur in a specific order, but the timing between them may vary or may not be defined at all.*

Semantically, an untimed sequence is similar to a log, with the main difference being that a log is taken as input while an untimed sequence is usually returned as output. In the paper, we will use the more generic term ''sequence'', and it will be clear from the context if we are referring to a log, a SERE sequence, or an untimed sequence.

Definition 10: ***Finite State Automaton (FSA)**: An FSA is a mathematical model that represents a system with a finite number of states. At any point in time, the system can be in one of these states, and it transitions from one state to another in response to external inputs. Formally, an FSA is a 5-tuple $(S, I, O, \delta, \lambda)$ where:*

- *S is a finite set of states,*
- *I is a finite set of inputs,*
- *O is a finite set of outputs,*
- *$\delta : S \times I \to S$ is the transition function that determines the next state based on the current state and input,*
- *$\lambda : S \times I \to O$ is the output function that determines the output based on the current state and input.*

Definition 11: ***Non-Deterministic Finite State Automaton (NFSA)**: NFSAs extend classical FSAs by allowing multiple possible transitions for a given state and input. In other words, the next state is not uniquely determined by the current state and input, but instead, the system can move to one of several possible states. Formally, an NFSA is similar to an FSA, but the transition function is modified as $\delta : S \times I \to 2^S$, meaning that for each state and input, the system may transition to a set of possible next states. NFSAs are useful for modeling systems with uncertain behaviors or systems that exhibit concurrency.*

Definition 12: ***Büchi Automata**: Büchi automata are defined similarly to finite state automata, but they operate on infinite input sequences, which makes them particularly useful in the context of verifying systems with ongoing behaviors, such as reactive systems. A Büchi automaton accepts a sequence if it passes through a set of "accepting" states infinitely often. Formally, a Büchi automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:*

- *Q is a finite set of states,*
- *Σ is a finite set of input symbols (alphabet),*
- *$\delta : Q \times \Sigma \to 2^Q$ is the transition function,*
- *$q_0 \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of accepting states.*

Büchi automata are particularly useful for verifying a design with LTL specifications (e.g., assertion-based verification approaches such as model checking or simulation-based dynamic approaches) because an LTL formula can be translated into an equivalent checker in the form of a Büchi automaton, which can then be used to check whether a design satisfies the corresponding specification.

V. TAXONOMY OF THE MINING TECHNIQUES

In this section, we explain the reasoning behind our classification scheme.

In recent decades, the main approaches for classifying specification mining techniques (usually mentioned in the ''related work'' section of the reviewed papers) employed the characteristics reported in description V.

Static/Dynamic The technique is dynamic only if the design is executed or simulated or if the mining process requires analyzing the execution traces of the system. Static techniques, in contrast, analyze the system without execution or simulation, typically using static code analysis or model inspection.

Supervised/Unsupervised If the input data is ''labeled'' (i.e., each instance in the dataset comes with an associated class label), the technique is considered supervised. In contrast, if the input data is not labeled, the technique is unsupervised, which attempts to identify patterns or structures within the data without prior knowledge of outcomes. The simplest and most common labeling involves distinguishing between positive (accepted behaviors) from negative (rejected behaviors) traces; however, in most scenarios, negative examples are difficult to observe, in particular, from black-box systems [133]. Therefore, most work focuses on mining specifications only from positive examples.

TABLE 1. Techniques and their corresponding papers.

Method	Description	Papers
Association rule mining	Techniques to mine specifications by using data-mining association rules.	[20], [21], [22], [23], [24], [25]
Automata learning	Techniques to mine automata.	[26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41]
Composition	Techniques to compose automata, invariants, temporal patterns, ..., into more complex expressions or automata.	[42], [43], [44], [45], [46], [47], [48], [33], [49], [50], [51], [52], [53], [23], [54], [39], [55], [56], [57], [58], [59]
Decision tree	Techniques to discover behaviors by recursively partitioning the input data into subsets based on the available feature values.	[60], [61], [62], [21], [63], [64], [65], [66], [67], [24], [68]
Invariant general	The most general approach to mine invariants by efficiently instantiating all possible variables with all possible non-temporal operators. This is mostly covered by Daikon.	[69], [70], [49], [50], [71], [72], [53], [55], [40], [73]
Natural language processing	Techniques to generate specifications by analyzing human language input to extract relevant information.	[74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92]
Placeholder permutation	Techniques that substitute all placeholders in a set of templates with all events/variables from a trace/design.	[27], [93], [94], [95], [70], [96], [43], [61], [44], [45], [97], [21], [31], [32], [98], [99], [48], [100], [101], [52], [102], [103], [104], [105], [106], [54], [107], [108], [109], [110], [111], [112], [113], [114], [73], [115], [116]
Sequential pattern mining	Techniques to discover recurring sequences of events or items in datasets.	[117], [118], [23], [113], [119], [24], [120], [59]
Other	Graph neural networks, bayesian inference, binary decision diagrams, answer set programming, maximum a posteriori inference, symbolic simulation, dependency graphs, taint and tag, mining partial orders, and specification sketching.	[94], [121], [46], [71], [51], [52], [122], [105], [123], [124], [125], [126], [127], [128], [129], [130], [131], [132]

Online/Offline Offline mining refers to learning the specification from a pre-existing, fixed set of available behaviors. Once the specification is inferred, it remains unchanged. Online mining, however, assumes that new behaviors will arrive over time. In response, the mined specification is continually updated and refined to incorporate the effects of new behaviors, allowing the system to adapt to changing conditions. During our screening procedure, we observed that the overwhelming majority of literature focuses on offline mining.

Passive/Active Passive mining involves learning a specification from a fixed, static set of behaviors without interacting with the system. In active mining, the system is available to actively generate new behaviors, steering the inference process. This approach includes the generation of counterexamples, which are used to refine and improve the specifications. According to our findings, most papers employing active approaches aim at mining automata while most LTL specifications miners follow a passive approach.

Temporal/Non-Temporal Temporal mining considers time-based sequences or patterns within the data, where the order and timing of events are crucial. Non-temporal mining, on the other hand, does not take time into account, focusing instead on the relationships or patterns that are independent of any specific order or timing.

Template as Input/Template Mined In this context, either a predefined template is provided as input to guide the mining process (Template as Input), or the mining process itself generates a template based on the data (Template Mined). The template can be predefined (a set of rules or patterns provided before mining) or user-defined, allowing for custom specifications.

Model Required/Model Not Required Some mining techniques require a model of the system to be available beforehand to guide the mining process (Model Required). In contrast, other techniques do not require any such model and can infer specifications directly from the data without relying on an existing system model (Model Not Required).

Description 5.1: Features traditionally used for classifying mining techniques

Even though the above features are widely used for classifications, they present a major drawback: any combination of the above features will group together completely different papers with different fundamentals and methods. For example, the "Dynamic" or "Static" label can applied to at least one paper of any subsection of section VI, where each subsection describes completely different techniques. Furthermore, most papers employ mixed techniques, some of which would require to be labeled with all the features described above. Another scheme might classify the mining techniques according to their use cases; however, most techniques can be applied to several use cases, making such a classification scheme unsuitable to our needs. As a consequence, we decided to classify the papers using the labels presented in Tab. 1. Each label of the table represents a family of techniques with its variants, each of which can exhibit a subset of the aforementioned features. Therefore, our classification scheme utilizes these traditional features, but only after an initial categorization into families of techniques. In the subsequent sections, we will clarify which characteristics are included for each technique discussed (or they will be evident from the context).

Differently from [10], we isolated in a single section (section VII) all the approaches that aim at evaluating the quality of the results achieved by the authors of the proposed techniques. That is to avoid redundancy in the exposition since several authors use the same basic approaches to evaluate the results of most techniques. Finally, the reader should know that only the most notable papers are explicitly cited in the text, while the entire set of papers for each family of techniques can be found in Tab. 1. To avoid redundancy when multiple papers describe the same technique, the papers with more citations will be preferred. Finally, authors who published an open-source tool or a prototype implementing their approach will be preferred to purely theoretical or irreproducible works.

A. USE CASES

In this section, we outline the main use cases of specification mining. Contrary to our previous paper [134], in which we reported the main use cases of assertion mining in list form, here we decided to map the use cases in a more structured way. Since most use cases usually overlap while a few are just subtle variations of the more general ones, we organized the main use cases as reported in Fig. 2.

In our review, the overall majority of papers (74%) do not have a well-defined use case, or the described approach can be applied to several use cases, all related to the discovery of new knowledge from the input. After that, we grouped all the use cases into two main categories. The first group contains all the use cases inherited from the classical uses of formal specifications. Traditionally, these activities were performed manually, often involving considerable time and expertise. Specification mining has significantly sped up

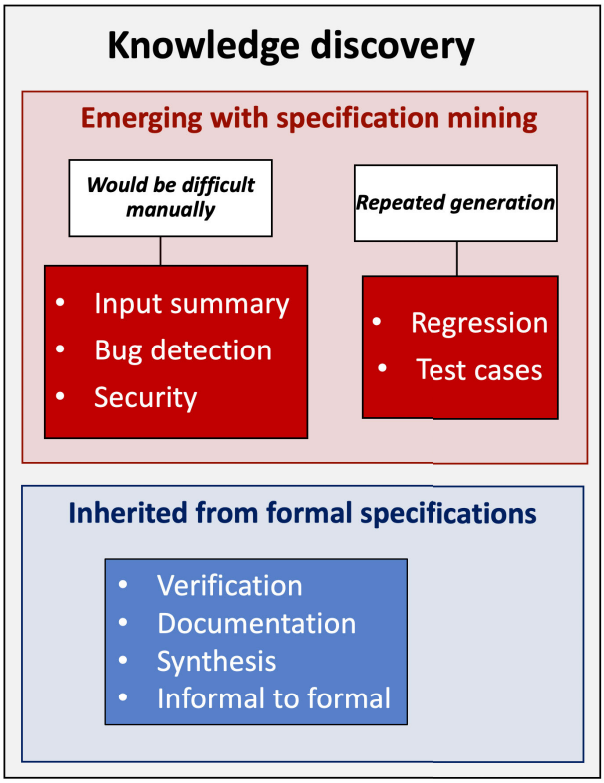


FIGURE 2. Map of specification mining use cases.

these processes. We report the use cases contained in this category below.

- **Verification**. This is the most classical use of formal specifications, which can be translated to checkers or monitors (pieces of code that check if a form specification holds during simulation of execution) from formal specifications; the same specifications can be reused with formal methods such as model checking or theorem proving to mathematical prove the correctness of a design.
- **Documentation**. Miners can automatically generate or update documentation by mining specifications, ensuring it accurately reflects the system's behaviors. This is particularly useful for legacy systems where documentation may be sparse or outdated [35]. Most of the reviewed papers claim this as one of their use cases; however, only 2.7% of the papers recognize this as their only use case.
- **Synthesis**. It typically involves the automatic generation of controllers or other system components from formal specifications. This process helps create parts of the system directly from high-level descriptions, reducing manual effort and ensuring that the synthesized components are correct by construction. This automation accelerates the design and deployment of systems, particularly in safety-critical domains. Only 1.8% of the reviewed papers claim this as their main use case.
- **Informal to Formal Transformation**. This involves converting informal specifications, which are

ambiguously written in natural language (such as English), into formal ones. All the papers in section VI-E cover variations of these use cases (13.6%).

The second group contains all use cases that emerged with the mining process. These are further divided into two subgroups: those that would be extremely challenging to achieve manually and are typically enabled through specification mining techniques and those that are possible due to the quickly repeatable nature of the mining process. The first subgroup contains the following.

- **Input summary**. The generated specifications represent a summary of the input data involving the most notable behaviors (usually the most frequent), aiding in understanding and analysis and being particularly useful in large-scale systems. This use case almost always goes together with the "Documentation" use case, as all techniques to do the former can be applied to the latter (56.3%).
- **Bug detection**. Identifies potential errors or vulnerabilities in the system by analyzing patterns and inconsistencies in the data or code. 8.1% of the reviewed papers claim this as their main use case.
- **Security**. Enhances the system's security by uncovering weaknesses or potential attack vectors, providing an additional layer of defense against potential threats. 6.3% of the reviewed papers cover this as a main use case.

The second subgroup involves the following use cases.

- **Regression detection**. Specification mining can be used to find specifications that are violated by a new version of a codebase. This can be useful for identifying regressions, i.e., bugs introduced in the new codebase version. This concept is also related to design refinements, where specifications generated for a prototype or an abstract design model are reused to check the correctness of its refined versions. Most approaches can be applied to this use case; however, heavy filtering might be necessary to keep only a small subset of meaningful specifications capable of detecting regressions (see section VII).
- **Test Cases**. Specification mining can help automatically generate test cases, usually through an iterative process executed in a feedback loop, repeatedly generating new specifications and tests. Only 2.7% of the papers cover this as a main use case.

Our review process identified a few additional "uncommon" use cases worth mentioning. In [25], the primary use case is to improve the performance of software verification, specifically bounded model checking, by reducing the search space that Satisfiability (SAT) solvers need to explore. In [135], the authors exploit assertion mining for identifying the resilient elements of a design to guide the design exploration in approximate computing. This can accomplished by generating assertions on a golden model and then checking how the truth values of the assertions change when the design is modified by inserting mutants. Resilient elements are those associated with assertions that remain true despite the modifications.

VI. MINING TECHNIQUES

This section is the core of the paper. Here, we report a detailed description of the mining techniques. Each subsection is organized as follows (except when the peculiarities of the reported techniques do not allow it). First, we report a high-level description of the family of techniques described in the section. Then, we describe the technique of the most influential (usually the oldest) papers. Finally, we build incrementally with the papers that provide more recent contributions.

A. AUTOMATA LEARNING

Automata learning is a mining technique to infer automata, usually an FSA. This technique is often used to infer specifications of black-box systems using no additional information besides the execution traces; however, some approaches rely on the source code of the DUV or require re-simulating it to obtain more traces.

1) FUNDAMENTALS OF AUTOMATA LEARNING

Ammons et al. [35] proposed the first approach to mine automata specifications for programs; in particular, the goal was to automatically discover the protocols that SW must obey when interacting with an API. The method is based on the observation that "common behavior is often correct behavior", translating the problem into probabilistic learning from execution traces. The proposed technique reduces the specification mining problem to learning regular languages, for which existing learners can be used. Before running the automata learning phase, the approach requires several preprocessing steps.

Overall, the preprocessing annotates each input trace with flow dependencies; then, those annotations are used to extract scenarios from the traces, which are small sets of dependent interactions.

Traces are arranged as a sequence of API interactions, where each interaction is of the form $interaction(attribute_0, \ldots, attribute_n)$ where each attribute refers to a specific piece of data associated with the interaction, for instance, values that are passed into or returned from the function calls, as well as any other relevant pieces of data that are part of the interaction. Hereafter, we report an example of a trace describing a program using a server-side socket API.

```
socket(domain = 2, type = 1, proto = 0, return = 7)
bind(so = 7, addr =0x400120, addr_len = 6, return = 0)
listen(so = 7, backlog = 5, return = 0)
accept(so = 7, addr =0x400200,
                addr_len =0x400240, return = 8)
read(fd = 8, buf =0x400320, len = 255, return = 12)
write(fd = 8, buf =0x400320, len = 12, return = 12)
read(fd = 8, buf =0x400320, len = 255, return = 7)
write(fd = 8, buf =0x400320, len = 7, return = 7)
close(fd = 8, return = 0)
```

```
10  accept(so = 7, addr =0x400200,
                addr_len =0x400240, return = 10)
11  read(fd = 10, buf =0x400320, len = 255, return = 13)
12  write(fd = 10, buf =0x400320, len = 13, return = 13)
13  close(fd = 10, return = 0)
14  close(fd = 7, return = 0)
```

First, the approach performs a "flow dependence annotation" step, which involves annotating program traces with data/temporal dependences to constrain how interactions may be reordered and to identify related interactions that could form scenarios.

For example, in the trace above, the process creates flow dependencies such as the one between the socket creation (line 1) and subsequent bind, listen, and accept calls (lines 2, 3 and 4) using socket descriptor 7. Therefore, the process discovers that those interactions are connected and that the order of some of these interactions can not be reversed (you can not use a socket before creating it).

After that, the process performs a "scenario extraction" step that transforms annotated traces into smaller, meaningful interaction scenarios. A scenario is a set of interactions related by flow dependences. The process receives three inputs: a set of annotated traces, a set of user-defined scenario seeds and a user-tunable parameter N to restrict the number of interactions in the extracted scenarios. Each seed is an interaction skeleton. The extractor searches the input traces for interactions that match the seeds and extracts a scenario from each seed. For example, suppose the extractor was given the trace above and *accept*(*so*, *return*) as the seed. The extractor would produce two scenarios, one around the accept on line 4 and the other around the accept on line 10. Then, a prioritized worklist algorithm is used to collect the ancestors and descendants of a seed interaction in a trace. The goal of the algorithm is to efficiently extract a set containing at most N interactions that are directly related to a chosen seed interaction through data and temporal dependences. After processing up to 10 (N=10) ancestors and descendants for the interaction seed *accept* (at line 4), the collected scenario for the trace above is reported hereafter.

```
1  socket(domain = 2, type = 1, proto = 0, return = 7)
2  bind(so = 7, addr =0x400120, addr_len = 6, return = 0)
3  listen(so = 7, backlog = 5, return = 0)
4  accept(so = 7, addr =0x400200,
                addr_len =0x400240, return = 8) [seed]
5  read(fd = 8, buf =0x400320, len = 255, return = 12)
6  write(fd = 8, buf =0x400320, len = 12, return = 12)
7  read(fd = 8, buf =0x400320, len = 255, return = 7)
8  write(fd = 8, buf =0x400320, len = 7, return = 7)
9  close(fd = 8, return = 0)
```

In this scenario, the ancestors of the seed (*accept*) are *socket*, *bind*, *listen*, whereas the descendants are *read*, *write*, *close*.

Finally, the process simplifies the scenario by removing unnecessary attributes that do not participate in any flow dependencies; then, it ensures that scenarios are in a consistent format (to facilitate learning) by replacing concrete values with symbolic names to generalize the scenarios. Below, we report the simplified and standardized scenario.

```
1  socket(return = x0)                          (A)
2  bind(so = x0)                                (B)
3  listen(so = x0)                              (C)
4  accept(so = x0, return = x1)  [seed]         (D)
5  read(fd = x1)                                (E)
6  write(fd = x1)                               (F)
7  read(fd = x1)                                (E)
8  write(fd = x1)                               (F)
9  close(fd = x1)                               (G)
```

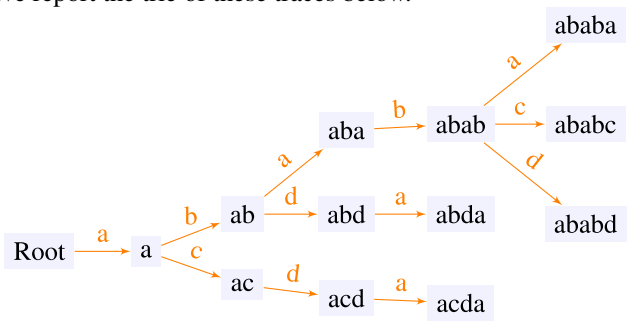
The simplified scenario removes arguments like *domain*, *type*, *proto*, *addr*, *addr_len*, *buf*, and *len* that are not part of flow dependencies. The standardization abstracts values 7 and 8 to x0 and x1, respectively; then, it assigns a unique identifier to each interaction, making sure to assign the same identifier to equivalent ones such as 5, 7 (E), and 6, 8 (F). The final standardized scenario string A B C D E F E F G is the abstract interaction pattern, ready for the automaton learning process.

The automata learning process involves two main steps: I) using an off-the-shelf Probabilistic Finite State Automaton (PFSA) learner, in this case, a variation of the k-tails [136] algorithm called sk-strings [137]; II) employing a post-processing procedure called "corer" that removes infrequently traversed edges to convert the PFSA into a simpler NFSA. The PFSA learner generalizes the training data to ensure coverage of common behaviors. In contrast, the corer calculates the "heat" of each edge, that is, how frequently the corresponding transitions appear in the input traces. Edges below a certain user-defined threshold are pruned, and states unreachable from the Root node are removed, resulting in a simplified NFSA. Below, we report a minimal example of how the entire procedure works. The algorithm works by building a prefix tree (trie) from the traces, annotating each state with k-length suffixes (k-tails), and then merging states that generate the same k-tails. The value of k in the k-tails algorithm is crucial for determining how states are merged. With a small value of k, the algorithm considers only short sequences following a state; this can result in more states being merged because the k-tails are more general. For example, with k=1, the algorithm only considers the immediate next character after each state, so states that have the same possible next characters will be merged: the resulting automata will be more general and less adhering to the training data. On the other hand, with a large value of k, the algorithm considers long sequences of characters when deciding whether to merge states; therefore, states will only be merged if they exhibit very similar long-term behavior. This results in a more detailed automaton that more closely reflects the exact behavior seen in the input traces, but that is also more prone to overfitting.

Consider the following set of traces.

```
1  a b a b a
2  a b a b c
3  a b a b d
4  a b d a
5  a c d a
```

We report the trie of these traces below.

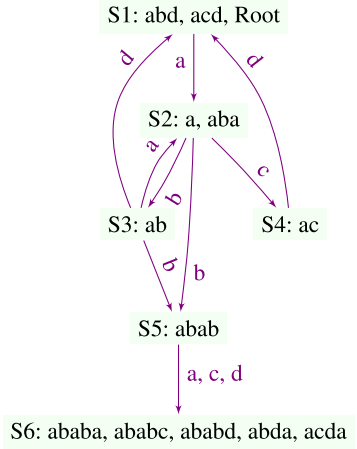


Note that we added an additional *Root* state that connects to all traces. For $k = 2$, the tails are the suffixes of length 2 of each state. Below, we report the k-tails of the example.

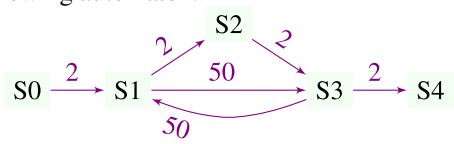
```
State      k-tails
Root       {a}
a          {ba, bd, cd}
ab         {ab, da}
ac         {da}
aba        {ba, bc, bd}
abd        {a}
acd        {a}
abab       {a, c, d}
abda       {}
acda       {}
ababa      {}
ababc      {}
ababd      {}
```

According to the k-tails guidelines, states *abda*, *acda*, *ababa*, *ababc*, and *ababd* can all be merged together as they all have the same "empty" future behavior. Likewise, states *Root*, *abd*, and *acd* can merged as they all have the same tail *a*.

The sk-strings algorithm extends k-tails by merging together states that are likely to generate the same k-strings according to a metric. For instance, if our metric is the percentage of overlap between two states, we could merge together states *a* and *aba* as they present 2 out of 3 overlapping tails (*ba* and *bd*). After that, we can rebuild the automaton with the merged states.



Finally, we can remove infrequent edges to extract the "hot core" of the automaton; however, we can not simply remove edges with low occurrences in the traces. Consider the following automaton.



Four edges have a weight of 2, which is low compared to the two edges with a weight of 50. However, any string through this PFSA must traverse the edge out of state *S*0 and the edge into the state *S*4. Despite their low weight, a string is more likely to traverse these edges than it is to traverse the edges with a weight of 50. A more effective way to assess an edge's "heat" is by determining the average probability that it will be traversed. This task is known as the Markov chain problem. In this context, the PFSA is represented using a *transition matrix* P, where each entry P_{ij} in the matrix represents the probability of transitioning from state i to state j. The goal is to solve for the steady-state vector π, which gives the long-term probabilities of being in each state. Mathematically, the steady-state vector π satisfies the equation $\pi P = \pi$ subject to the normalization condition $\sum_i \pi_i = 1$ where π_i is the steady-state probability of being in state i. To find π, we typically solve the linear system of equations $(I - P)\pi = 0$, where I is the identity matrix. Once the steady-state probabilities π_i for the states are known, the likelihood of traversing each edge can be computed. This likelihood ℓ_{ij} for the transition from state i to state j is given by $\ell_{ij} = \pi_i \cdot P_{ij}$ where P_{ij} is the transition probability from state i to state j, and π_i is the steady-state probability of being in state i. After solving the Markov chain problem and obtaining the edge likelihoods, edges with low likelihoods (below a certain threshold) can be pruned. This step reduces the complexity of the PFSA by focusing on the most relevant transitions, leading to a more efficient model that still accurately represents the probabilistic process it was designed to capture.

2) AUTOMATA LEARNING OF CLUSTERED TRACES

Lo et al. [41] build on Ammons's work by proposing another automata learning specification mining architecture called SMArTIC (Specification Mining Architecture with Trace Filtering and Clustering) designed to enhance the accuracy and scalability. SMArTIC is built on two main ideas: I) input traces should be cleaned of all "erroneous traces" before mining; II) mined specifications will be more accurate when they are obtained by merging the specifications learned from clusters of related traces.

The architecture includes four components: erroneous-trace filtering, related-trace clustering, a PFSA learner, and a merger.

The **filtering block** filters out erroneous or unlikely SW traces: intuitively, the process identifies statistically

significant temporal rules that represent common behaviors in the traces; then, these rules are used to filter out traces that deviate significantly from those behaviors. In particular, the process employs a sequential mining technique [138] (this family of techniques is discussed in section VI-C) to generate LTL specifications of the form $G(pre \rightarrow post)$ where $pre \rightarrow post$ is of the form $pre_1 \wedge XF(pre_2 \wedge \cdots \wedge XF(pre_{end} \rightarrow (post_1 \wedge XF(post_2 \ldots)))$, and where each pre_i is a proposition. Not all statistically significant rules are useful for filtering outliers: the approach focuses on rules with high support and high but less than 100% confidence, as these indicate common behavior with occasional deviations. Where the confidence is the ratio of times in which antecedent true implies consequent true (support) with respect to the total number of times in which the antecedent is true; as a consequence, the generated rules might have instances on a trace in which antecedent true implies consequent false. All traces that "deviate" from the generated sequential behaviors are pruned. Formally, a trace $t = \langle a_1, a_2, \ldots, a_m \rangle$ is an outlier if there exists a rule $G(pre \rightarrow post)$ such that $\exists a_i, \exists a_j.(1 \leq i \leq j) \wedge (a_i \ldots a_j$ satisfies $pre) \wedge (a_{j+1} \ldots a_{end}$ satisfies $post))$.

The **clustering block** is designed to group related traces together. The clustering block ensures that each cluster contains traces that share similar patterns; this allows the learning process to operate within a more homogeneous set of data, making it easier to capture the specific nuances of each pattern without the interference of unrelated traces, mitigating the risk of producing an over-generalized model. In other words, instead of learning one broad, possibly inaccurate model from the entire dataset, the system learns multiple, more precise models from each cluster. To cluster the traces effectively, the authors propose a metric to measure the similarity between two traces. Let $t_i = \langle a_1, a_2, \ldots, a_m \rangle$ and $t_j = \langle b_1, b_2, \ldots, b_n \rangle$ be two traces. They first convert these traces into a hierarchical grammar using the Sequitur algorithm [139], which represents each trace as a regular expression. The trace similarity is then calculated using global sequence alignment [140] on these regular expression representations. Turning traces into regular expressions is necessary to handle variations in trace sequences that arise from loops or repeated actions within the traces, making direct comparisons between traces difficult. For example, consider two traces from a file processing system $t_1 = \langle$open, read, read, read, read, close$\rangle$ and $t_2 = \langle$open, read, close$\rangle$. Although these traces differ in length due to the number of "read" operations, they represent the same basic sequence of actions. By converting the traces into regular expressions such as $(openread + close)$, we can abstract away the specific number of iterations and capture the essence of the sequence.

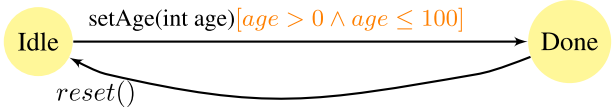
After that, the k-medoids clustering algorithm [141] is used to partition the set of traces $\mathcal{T}$ into k clusters $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$. The number of clusters k is determined dynamically by the algorithm based on the similarity metric $k = \arg\max_k \sum_{i=1}^{k} \sum_{t_j \in \mathcal{C}_i} \text{similarity}(t_j, \mu_i)$ where μ_i is the medoid of cluster $\mathcal{C}_i$, and similarity(t_j, μ_i) is the similarity between trace t_j and the medoid μ_i. Each trace $t_j \in \mathcal{T}$ is assigned to the cluster $\mathcal{C}_i$ that maximizes the similarity.

The **learning block** follows the same approach as Ammons'; it employs the sk-strings algorithm to generate a PFSA for each cluster of traces.

Finally, the **merging block** combines the individual PFSAs $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ generated from each cluster into a single unified automaton $\mathcal{A}$. The merging process identifies and merges equivalent transitions between the PFSAs. Then, since the merged automaton $\mathcal{A}$ must preserve the likelihood of generating each trace as in the original PFSAs, each transition in $\mathcal{A}$ is assigned the probabilities based on the weighted average of the probabilities from the individual PFSAs.

3) AUTOMATA LEARNING OF GUARDED FINITE STATE AUTOMATA

More recently, Mariani et al. [37] proposed an approach called GK-tail+ (it is an extension of a previous approach from the same authors called GK-tail [40]); in essence, it is a variant of the above approaches to efficiently generate software models as a Guarded Finite State Automaton (gFSA). A gFSA is like a traditional FSA but with an added layer of conditions (called guards) that must be true for transitions between states to happen. Consider the gFSA below, where event *setAge*(*int age*) produces transitions only if the guard age satisfies the constraint $age > 0 \wedge age \leq 100$.



The proposed approach comprises the four main steps described below.

- **Merging traces**: the system combines equivalent traces representing the same scenario into a single, generalized trace. Two traces are event equivalent if they have the same sequence of events but possibly different values. This reduces redundancy by summarizing the values in one trace while retaining the sequence of events. For example, consider the traces below.

```
setAge(25) save() close()
setAge(32) save() close()
setAge(27) save() close()
```

  These traces are considered equivalent because they all involve the same sequence of method calls but with different parameters of *setAge*. The generalized trace would merge these into the trace *setAge(25, 27, 30) save() close()*.
- **FSA generation**: the process executes a procedure similar to k-tails to generate an initial FSA in the form of a prefix tree but with the generalized traces. After that, the FSA is simplified by merging states representing the same underlying behavior. This is done by determining when two states are equivalent (same k-tails) or subsumed by one another (the k-tails of a state is a subset of the k-tails of another state).

- **gFSA generation**: the final step involves refining the FSA by replacing the sets of values on transitions with constraints. These constraints are logical conditions that must be met for a transition to occur. This is done using a tool like Daikon (see section VI-B), which analyzes the values and generates relevant constraints. For example, consider the transition in the FSA: $S0 \rightarrow setAge(25, 30) \rightarrow S1$. Daikon might analyze the values 25 and 30 and generate the constraint $age > 20$ for this transition. In that case, the transition in the gFSA would become: S0 $\rightarrow$ $setAge(int\ age)[age > 20] \rightarrow S1$

4) AUTOMATA LEARNING WITH REFINEMENT THROUGH INVARIANTS

In [30], Beschastnikh et al. proposed an alternative method to generate FSA specifications implemented in a tool called Synoptic [142]. The tool only requires as input the system logs and the regular expressions to parse the logs. Synoptic employs an algorithm called BisimH, which combines refinement and coarsening techniques to generate system models. The algorithm parses the logs into trace graphs and then mines temporal invariants. Synoptic refines the model until it satisfies all mined invariants and then coarsens the model to make it more efficient. We report the details of the methodology below.

First, the log is parsed to identify and categorize the events within the log files. Regular expressions are used to extract key components from each line in the log, such as the timestamp (to order events) and the event type (to describe the action performed). For example, a log containing the following entries,

```
74.15.155.103 [06/Jan/2024:07:24:13] "GET HTTP/1.1/check-out.php"
13.15.232.201 [06/Jan/2024:07:24:19] "GET HTTP/1.1/check-out.php"
13.15.232.201 [06/Jan/2024:07:25:33] "GET HTTP/1.1/invalid-coupon.php"
```

might require the input regular expression

```
^(?<ip>\d+\.\d+\.\d+\.\d+) \[(?<timestamp>[^\]]+)]
"GET HTTP/1.1 /(?<event>[\w-]+)\.php"
```

to identify the sequence composed of the events reported below.

```
(TraceID: 74.15.155.103, Timestamp: 07:24:13, Type: check-out)
(TraceID: 13.15.232.201, Timestamp: 07:24:19, Type: check-out)
(TraceID: 13.15.232.201, Timestamp: 07:25:33, Type: invalid-coupon)
```

These events are then grouped into traces based on their trace identifiers.

After that, Synoptic examines the traces to identify consistent patterns in the form of ''temporal invariants'': temporal relations that hold in all traces and throughout the entire trace. The mining of temporal invariants is based on counting the occurrences and relationships between pairs of events across all traces in the log. The temporal invariants mined are of the form: ''always followed by'' ($a \Rightarrow b$): event a is always followed by event b later in the same trace; ''never followed by'' ($a \not\Rightarrow b$): event a is never followed by event b in the same trace; always precedes ($a \Leftarrow b$): event b is always preceded by event a earlier in the same trace. To be clear, the pattern $a \Rightarrow b$ is mined only if the number of occurrences of ''a follows b'' is equal to the occurrences of a.

After that, the tool generates an initial model that serves as the starting point for the consequent refinement process. This model is a simplified version of the system's execution, where each unique event type observed in the log is represented by a single state in the model. For example, if the log contains events like ''check-out'', ''get-credit-card'', and ''reduce-price'', then the initial model will have three nodes, one for each of these event types. Every occurrence of a specific event type across different traces is grouped together in its corresponding node. Directed edges are created between nodes based on the sequence of events observed in the traces. If an event of type a is immediately followed by an event of type b in any trace, then there is a directed edge from the node representing a to the node representing b.

The refinement process iteratively improves the initial model by splitting nodes that violate any of the mined temporal invariants. The goal is to create a model that satisfies all the invariants, accurately reflecting the system's behavior. During refinement, if an invariant is violated, the corresponding state is split to correct the violation. This process continues until the model satisfies all invariants. To be more precise, the tool employs a model checker to generate a counterexample for each failing invariant; then, it uses the counterexample guided abstraction refinement (CEGAR) [143] approach to determine a set of candidate partitions for which a split exists that removes at least one of the counterexamples. Suppose the initial model allows a path where a *valid-coupon* is followed by an *invalid-coupon*, violating the mined invariant $valid\text{-}coupon \not\Rightarrow invalid\text{-}coupon$. The refinement process would identify this violation and split the state containing *check-out* into two: one that can transition from *valid-coupon* and another that cannot, thereby ensuring the invariant is satisfied.

After refinement, the model may become overly detailed with some unnecessary partitions. Therefore, the tool performs a coarsening process to merge these partitions back together, provided the resulting model still satisfies all (or most of) the temporal invariants; this is done by using the k-tail algorithm to merge equivalent states. For example, after refinement, there might be two states for *check-out* that can be merged because they lead to the same subsequent events. The coarsening process would merge these states, resulting in a simpler model.

In [144], the same authors propose another tool called CSight [142] that mirrors Synoptic's inference procedure, but that uses different algorithms and works for concurrent systems.

5) FULLY STATIC AUTOMATA LEARNING

Differently from the approaches described above, Shoham et al. [39] propose a completely static method for mining temporal API specifications from client code. The approach works into two main phases: abstract-trace collection and summarization.

The goal of the **abstract-trace collection** phase is to analyze the source code of the DUV to gather information

about how objects interact with the APIs. To do this accurately, the analysis must handle complexities such as aliasing (where multiple references point to the same object) and unbounded sequences of events (e.g., loops). The result of this phase is a set of abstract histories that summarize possible event sequences for each object in the form of automata. The abstract-trace collection is based on abstract interpretation [145], a technique used in static analysis to approximate the behavior of a program. The main components of this technique are the following.

- Heap abstraction: abstracts the program's memory (heap) to track the flow of objects.
- History abstraction: abstracts the sequence of events for each object. It uses automata to represent the sequences; the abstraction either focuses on recent events that led to the current state or possible future events from the current state.

Additionally, the approach employs an ''extend Operator'' to add a new transition in the automaton representing the object's state when a new event (method call) occurs and a ''merge operator'' that combines histories that arise from different execution paths but represent the same abstract object. For example, consider a Java program that uses the ''SocketChannel'' objects. The goal is to infer the typical sequence of API calls made on these objects. The program might involve operations like ''connect'', ''finishConnect'', ''read'', and ''write''. Suppose a method ''createChannels'' returns a collection of ''SocketChannel'' objects. The heap abstraction will track these objects as they flow through the program, ensuring that each object's state is correctly represented even if they are passed across multiple methods. If a ''SocketChannel'' object is used in a loop where ''read'' is repeatedly called, the history abstraction will capture this behavior as an automaton with a loop, indicating that ''read'' can occur multiple times. Furthermore, when the program calls ''connect'' on a ''SocketChannel'', the extend operator adds a new transition in the automaton representing the object's state, moving from a state where the object is not yet connected to a state where it is awaiting connection completion. Finally, if two execution paths lead to the same state of a ''SocketChannel'' object (e.g., after connecting but before reading), the merge operator combines these paths into a single state in the automaton, reducing the complexity of the resulting specification.

After collecting abstract traces, the next step is to summarize them to eliminate noise and focus on the true usage patterns of the API. The **summarization phase** involves several key techniques:

- Statistical noise elimination: the analysis ranks the frequency of event sequences across different client programs. Sequences that occur infrequently are considered noise and are discarded.
- Clustering Algorithm: abstract histories are grouped into clusters based on their similarity. Histories within the same cluster are assumed to represent the same usage pattern. The clustering is based on automaton inclusion, where one automaton includes another if it accepts all the same sequences and possibly more.
- Weighted union: after clustering, a weighted union is performed within each cluster. This process aggregates the histories by assigning higher weights to more frequent sequences and removing transitions with low weights (considered as noise).

The process of clustering and weighted union is repeated until a stable set of clusters is achieved, ensuring that the final output reflects the true API usage patterns.

For example, suppose the abstract trace collection generated sequences where ''close'' is sometimes called twice due to an imprecise analysis. Since this sequence occurs infrequently, the statistical analysis would identify it as noise and remove it from the final specification. Imagine the analysis produced two different sequences: one where ''finishConnect'' is followed by ''read'' and another where it is followed by ''write''. The clustering algorithm should recognize that these represent two distinct usage patterns and place them in separate clusters. Furthermore, within the cluster where ''finishConnect'' is followed by ''read'', the weighted union might find that ''read'' is frequently followed by another ''read'' and then ''close''. It would then consolidate this pattern into a single automaton that reflects the correct usage sequence, removing any infrequent transitions that don't fit the pattern.

6) ACTIVE AUTOMATA LEARNING

Kang and Lo [27] propose an active mining tool called DICE [146] to generate FSAs from the execution traces of SW systems. The authors claim that existing approaches are often inaccurate because they rely on limited and non-representative test cases; therefore, they present a novel approach called ''adversarial specification mining'' to generate diverse test cases by searching for counterexamples to the mined specifications. The DICE approach consists of three main sequential phases: mining purity-aware temporal specification, adversarial test generation, and FSA inference.

First, the tool mines **purity-aware LTL specifications**; by ''purity-aware'', the approach considers whether the interpretation of event sequences is affected by methods with side effects. The authors single out ''pure'' (side-effect-free) methods through a lightweight static analysis of the source code of the DUV to identify methods that do not modify the system's state: they apply a simple heuristic based on the method names. Specifically, methods whose names start with prefixes like is- or has- are considered likely to be getters, which typically return a value or a property of an object without modifying its state.[1]

After that, the approach uses six LTL binary templates adjusted to account for pure methods, reformulating the

[1]This concept of purity is not new and was also explored in a previous specification mining work [33] where pure methods called ''inspectors'' are used to monitor an SW object after impure methods called ''mutators'' changed its state.

properties to ignore such methods when they occur between the pairs of significant events being analyzed. For example, the template $G(a \rightarrow b)$ (event a must immediately be followed by event b) is converted to $G(a \rightarrow X((p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\ U\ b))$, where $p_1, p_2, \ldots, p_n$ are side-effect free events (an occurrence of event a must be followed by an occurrence of event b, ignoring all occurrence of side-effect-free events).

For example, consider an object where the method "add()" inserts an item, "isEmpty()" checks if it is empty, and "clear()" removes all items. Without purity awareness, $G(clear \rightarrow isEmpty\_true)$ states that immediately after calling "clear()", the "isEmpty()" method should return "true". This would be overly strict, as the actual implementation might involve side-effect-free checks that do not affect the state, erroneously invalidating the property, as in the sequence *clear*() *getSize*() *isEmpty*(). Reformulating this to consider purity would ignore intermediate pure methods, making the specification more robust. The tool generates LTL specifications compliant with those templates using an approach similar to Texada [96] (see section VI-B).

After that, the mined temporal specifications are tested for their validity by **generating new test cases** designed to invalidate them. The idea is that if a specification can be invalidated by a legitimate sequence of method calls, then it was not a correct specification to begin with. This process ensures that only accurate and robust specifications survive. In particular, the mined LTL specifications are transformed into fitness goals for a search-based test generation tool called Evosuite [147]. The search algorithm is guided by fitness functions, which prioritize generating test cases that are close to invalidating a specification. The fitness of a generated test case is evaluated based on how well it can provide counterexamples (traces violating the LTL properties) for the specifications. Continuing with the example above, suppose an initial specification states that after calling "clear()", the data structure must be empty ($isEmpty() = true$). The DICE approach would generate test cases trying to invalidate this by, for example, calling methods that might reinsert items or fail to clear the data structure completely. If a test case showed that calling "clear()" followed by "add()" can result in $isEmpty() = false$, it would invalidate the original specification. The algorithm would then discard this incorrect specification and focus on refining the others. The final phase uses the validated specifications and the generated test cases to **build an FSA** that accurately models the software's behavior. The approach employs a variant of the k-tail algorithm to account for pure methods; moreover, it leverages the mined LTL properties to avoid merging states that would result in invalid transitions that make some LTL properties fail.

7) AUTOMATA LEARNING USING DEEP LEARNING

The works in [28] and [29] introduced Deep Learning-based specification mining, which aims to infer FSA specifications from a DUV using a deep-learning model trained on the execution traces.

The approach is composed of six main steps, as reported below.

1) **Test case generation and trace collection**: the process generates a substantial number of test cases using automated tools. These test cases are designed to cover various scenarios and paths within the software.
2) **Training**: the approach employs deep learning models to analyze the collected traces. These models are trained to predict the following method call based on the sequence of previously called methods. The goal is to capture complex dependencies within the trace data. Examples of models used within this family of techniques are long short-term memory [148] and transformers [149].
3) **Feature Extraction**: once the models are trained, they are used to extract features from the traces. The key features include the likelihood of each subsequent method call given the current sequence and the specific positions and order of methods within the trace sequences. These features encapsulate the behavior and interaction patterns within the software system.
4) **Construction of a Prefix Tree Acceptor (PTA)**: the process constructs a PTA (similar to the trie previously described) using the predictions from the deep learning models. The tree "accepts" a string if a path exists from the root to a leaf node corresponding to that string. Here, the PTA represents all the observed sequences in the execution traces, effectively summarizing the software's behavior.
5) **Clustering and simplification**: clustering algorithms are applied to transform the PTA into a more simplified and generalized FSA. These algorithms group similar states within the PTA based on the extracted features, merging equivalent or similar states to reduce complexity.
6) **Model selection and evaluation**: various clustering methods are employed, which do not require the number of clusters to be predefined, allowing the natural structure of the data to emerge. This step results in a simplified FSA that generalizes the behavior captured in the traces while maintaining accuracy.

B. PLACEHOLDER PERMUTATION

The placeholder permutation technique involves testing all possible combinations of design/trace variables or events within a given set of specification templates. Despite its versatility and wide adoption, the technique often encounters scalability challenges due to the combinatorial explosion of potential permutations. Various heuristics and optimization strategies have been developed to address this issue, enabling more efficient exploration and validation of possible configurations. This section explores the application of placeholder permutation techniques across different mining contexts.

1) FUNDAMENTALS OF PLACEHOLDER PERMUTATION

Yang and Evans [107] propose one of the first works on dynamically inferring properties from program traces. Starting from a predefined set of Quantified Regular Expressions [150] property patterns such as $[-P0] * (P0[-P1] * P1[-P0]*)*$ (alternating patter: $P0$ followed by a sequence without $P1$ until $P1$ occurs, this pattern can repeat). The approach replaces all abstract events $P0$ and $P1$ in the pattern with monitored events from the program's method entries and exits. For example, if $P0 = Pro\_Add$ (producer's add method exit) and $P1 = Con\_take$ (consumer's take method exit), the concrete property would be:

$$[-Pro\_Add]* \\ (Pro\_Add[-Con\_take] * Con\_take[-Pro\_Add]*)* \quad (1)$$

If the program monitors n events, and the pattern is parameterized with m abstract events, there are n^m possible instance properties. For each instantiated concrete property, the approach determines if the concrete property is satisfied by the trace; if it is, then the property is added to the set of inferred properties. For example, consider the following trace of events *Pro_Add, Pro_Add, Con_take, Pro_Add, Pro_wait, Pro_wait, Con_wait, Con_take, Pro_Add, Con_take*. On this trace, the concrete property is satisfied as the sequence of events clearly shows that each Pro_Add event is followed by a Con_take; therefore, the property 1 would be added to the set of inferred properties.

2) EFFICIENT PERMUTATIONS OF LTL FORMULAS

One of the most cited iterations of the placeholder permutation technique is proposed by Lemieux et al. [96], where the authors develop a tool called Texada [151] to mine LTL specifications from traces. The tool allows the definition of LTL (only the classical operators) templates of the form G(antecedent $\rightarrow$ consequent) that includes ''holes''. These holes are placeholders for design variables. An example of a template would be $G(P0 \rightarrow XFP1)$, where $P0$ and $P1$ are placeholders for events found in the log. Texada parses the user-defined LTL property type into a tree-like structure, where the leaves are placeholders for atomic propositions. Texada generates all possible bindings of the event variables in trace with the template placeholders: a binding is a mapping from the placeholders (e.g., $P0$, $P1$) to actual events found in the traces (e.g., *in_x*, *out_y*). For each binding, Texada instantiates the LTL template by replacing the placeholders with actual events from the traces, creating concrete LTL formulas. Texada checks the validity of each instantiated LTL formula on the provided traces through two main approaches: the linear miner and the map miner. The linear miner evaluates the property instance by iterating over the trace and recursively evaluating each operator in the property tree according to its semantics. For example, the expression $G(a \wedge b)$ would check, starting from every point of the traces, that a and b are simultaneously true. The map miner employs a different representation of traces; instead of a simple linear sequence of events, it uses a map where each event is associated with a list of positions in the trace. For example, if the trace is $[a, a, b, b, a, c]$, the map would be: $\{a : [0, 1, 4], b : [2, 3], c : [5]\}$ as it stores the positions of where each event occurs in the trace. This representation allows the miner to efficiently jump to relevant sections of the trace without having to scan through irrelevant parts. The authors utilize three main functions: *First-Occurrence*, *Last-Occurrence*, and *Check-Map*; these functions are designed to improve the efficiency of the mining process by quickly navigating long traces with sparse relevant events. *First-Occurrence* and *Last-Occurrence* identify the first or last position in a given interval where a sub-formula is true; they use binary search on a sorted list of event positions to quickly find these occurrences. For example, for a formula $p\ U\ q$, the *First-Occurrence* function finds the first position where q is true, and *Last-Occurrence* finds the last position before q where $\neg p$ occurs. This helps determine where $p\ U\ q$ holds within the interval. The *Check-Map* function evaluates whether a property holds starting from a given index in the trace, skipping irrelevant sections of the trace; furthermore, when handling LTL operators, the function leverages the results from *First-Occurrence* and *Last-Occurrence*. For instance, to evaluate Gp at index i, *Check-Map* uses *First-Occurrence* to find the first position of $\neg p$ after i. If $\neg p$ never occurs, Gp holds for that interval.

Finally, Texada uses a memoization strategy to minimize redundant computations when checking LTL property instances. This approach is crucial because many permutations can share identical sub-formulae, leading to repeated logic evaluations. Texada leverages the tree-like structure of LTL properties to identify and reuse identical sub-trees across different property bindings: it stores intermediate results of sub-formula evaluations, and when the same sub-formula reappears, it uses the stored result instead of re-evaluating it. More recently, the authors of [99] proposed a fairly similar approach to Texada but for Past-Time Linear Temporal Logic (PTLTL). The authors focus on optimizing the checking of past-time temporal properties with advanced caching mechanisms and multi-threading to check several traces in parallel.

Similar to Texada, the work in [61] proposes a tool called HARM [152] to mine temporal specifications from traces; however, the tool allows the definition of user-defined templates supporting the entire LTL+SERE language. The approach proposes a syntactic-based optimization to avoid generating redundant permutations, leveraging the structural characteristics of the templates. In particular, the tool generates a reduced set of permutations of design variables for the placeholders in the templates. This process considers two types of operators: I) Commutative operators, like $\wedge$ (and) and $\vee$ (or), where the order of operands does not matter; II) Non-reflexive operators, like U and R, where the same operand cannot appear on both sides. For example, in a mining context with template $G(ack \rightarrow P0 \wedge P1)$ containing placeholders $P0$ and $P1$ and using propositions $v1$, $v2$, $v3$, the

tool would avoid generating both $v1 \wedge v2$ and $v2 \wedge v1$, recognizing them as equivalent due to the commutative nature of $\wedge$.

Once the placeholders in the templates are instantiated, the tool proceeds to mine assertions from these instantiated templates by evaluating them against the input traces. To perform the evaluation, HARM converts the LTL expressions to automata and implements a linear-time evaluation function with respect to the length of the trace and the number of states and edges of the automaton. The idea behind the evaluation function is to group evaluation units requiring the same operations and process them together, avoiding redundant computations.

3) PERMUTATIONS USING MATRIX-BASED INFERENCE

Yang et al. [44] proposed a tool called Perracotta [153], which is among the most influential approaches in the field of specification mining for software systems. The most significant additions with respect to the placeholder permutation technique revolve around 1) a preprocessing approach for abstracting execution traces and 2) proposing an efficient algorithm to generate all permutations of events for several binary temporal templates.

1) The authors address the complexity of dealing with numerous instances of similar elements within the program traces by abstracting these instances into single representative units. This abstraction process is essential for making the inference of properties more tractable and meaningful when dealing with large-scale software systems. For example, assuming the software contained the events $mutex1.lock()$ and $mutex2.lock()$, the process may abstract the concrete instructions to $mutex.lock()$.
2) The authors propose a matrix-based inference approach to quickly test all pairs of events on eight binary templates, significantly improving the efficiency and scalability of mining temporal API rules from execution traces. One notable supported template is (PS)*. The first step in the matrix-based approach is to encode each distinct event in the trace with a unique index; for instance, in a system that monitors events like $lock1.acquire$, $lock1.release$, $lock2.acquire$, and $lock2.release$, these might be encoded as 0, 1, 2, and 3, respectively. The next step is to create a matrix M of size n x n, where n is the number of distinct events. Each element $M[i][j]$ in the matrix is an ID corresponding to the current state of the FSA generated from the input template and accepting a sequence of events in the trace. For the (PS)* template, the FSA tracks whether the pair (P, S) satisfies the condition "event P is followed by event S in a strict alternating order". The matrix is initialized to zeros, where each zero represents the initial state of the state machine (waiting for P). As the trace is processed, the matrix is updated to reflect the next states of the automata. This approach tests all pairs in O(nL), where L is the length of the trace. This is true because each event in the trace is processed once, and for each event, the corresponding row and column in the matrix are updated. When an event occurs in the trace, only the state machines involving that event need to be updated. For an event e occurring in the trace, e can be either the first event P or the second event S in a pair. Thus, only the rows and columns of the matrix corresponding to e need to be updated.

Li et al. [45] extended Perracotta's work to handle traces with multiple concurrent events at the same cycle (typical of HW designs) and to mine richer binary patterns. In particular, the scope of mined patterns is expanded to include new LTL operators such as "until", "next" and "eventually". Furthermore, the paper in [49] extends Perracotta by combining arithmetic invariants with temporal operators to mine more complex LTL properties that can handle non-Boolean data types. This extension allows for the mining of properties like $G((x > 0) \rightarrow XF(y < 0))$. The approach uses Daikon (see VI-B8) to generate the non-boolean aspects of the properties.

4) PERMUTATIONS WITH LABELED TRACES

In the context of supervised mining, Weimer et al. [102] propose a variation of the classical placeholder permutation technique to identify design corner cases containing bugs in software systems from execution traces. The methodology is based on the observation that programs often behave correctly on normal execution paths but make mistakes along exceptional control-flow paths; therefore, that knowledge is exploited to filter out the generated specifications. The mining algorithm assumes that if a policy is important, it will be enforced at least once using software error handling constructs like "catch" or "finally" blocks; as a consequence, the input traces are classified into normal traces, where no method calls terminate with an exception, and error traces, where at least one method call terminates with an exception. Then, the algorithm attempts to learn pairs of events (a, b) corresponding to the regular expression $(ab)*$. The main contribution of the approach consists in removing irrelevant or less critical event pairs by applying the following criteria: event b must occur at least once in an error trace; both events a and b must be declared in the same code package; there must be at least one error trace where event a occurs without event b to ensures that the specification can potentially reveal program errors. These filters help ensure that the learned specifications are contextually relevant and meaningful for error detection.

5) PERMUTATIONS IN ONLINE MINING

In the context of online mining, Gabel and Su [97] propose an approach that dynamically infers temporal properties (patterns of software method calls) at runtime as the program executes. This online method continuously processes

trace events to detect and enforce correct usage patterns, providing immediate feedback and anomaly detection. The tool is capable of mining the following predefined patterns: (ab), $(ab)+$, $(a?b)$, $(ab|ba)$. The proposed approach operates over a small, finite window of recent trace events to balance the computational overhead. For each window, the system matches the contained events with the predefined templates to identify any new potential specifications; then, a new corresponding FSA is generated for the sequences of events observed within this window. The FSAs act as monitors for potential specifications and are capable of counting how often each pattern is satisfied or has failed. For example, if event a is not followed by b, the failure count for the sequencing pattern (ab) increases. As events are processed, the algorithm updates the state of the FSA for the relevant patterns based on the current event and the events in the window. If a pattern is satisfied repeatedly, it becomes enforced: a new specification is found; on the contrary, if a pattern is violated, an anomaly is reported. To better understand the approach, consider a Java program managing a file resource through the following methods: open(), read() and close(). The correct usage is ''open'' before ''read'' and ''close'' after ''read''. Method calls are traced as [*open*, *read*, *close*] and processed within the boundaries of the finite window, and the observed sequences are mapped to pattern template (ab); one inferred candidate is ''open followed by read''. The candidate's state machine is updated with counts of satisfaction or failure. Since the pattern is satisfied several times, the process generates and enforces the new specification (*open read*). Further failures of the inferred specification (e.g., open followed by close without read) are detected and reported immediately, indicating potential defects.

6) ENUMERATIVE LTL LEARNING WITH GRAPHICS PROCESSING UNITS (GPUs)

Valizadeh et al. [116] proposed a novel Graphics Processing Unit (GPU)-accelerated approach for learning LTL formulas from positive and negative traces. The authors introduced an *enumerative synthesis method* that leverages parallelism on GPUs to accelerate the mining process. The prototype of the tool is available at [154]. The proposed methodology consists of three main components: **branch-free LTL semantics**, **divide-and-conquer decomposition** (D&C), and **formula simplification techniques**. Traditional LTL evaluation methods rely on branching operations, which perform poorly on GPU architectures due to thread divergence. To address this, the authors devised a branch-free logic evaluation strategy using bitwise operations. LTL formulas are represented as contiguous matrices of bits (Characteristic Matrices), enabling fast bitwise parallel computation. Temporal operators like *eventually* and *until* are computed efficiently via an exponential propagation algorithm, reducing complexity from O(n) to O(log n). To scale LTL learning to large trace datasets, the authors introduced a recursive divide-and-conquer method, where the dataset of positive (P) and negative (N) traces is split until each subset is small enough for direct processing. The split is guided by the split window parameter (win), which dynamically adjusts based on GPU memory constraints. The recursive D&C process works as follows.

1) **Dataset splitting**. If $|P| + |N| > win$, traces are split into four subsets (P_1, N_1), (P_1, N_2), (P_2, N_1), and (P_2, N_2).
2) **Recursive learning**. Each subset is processed recursively to learn sub-formulas $\varphi_{11}, \varphi_{12}, \varphi_{21}, \varphi_{22}$.
3) **Formula recombination**. The learned formulas are combined using logical operations:
$$(\varphi_{11} \wedge \varphi_{12}) \vee (\varphi_{21} \wedge \varphi_{22})$$
4) **Simplification**. The resulting formula is simplified using theorem proving and redundancy elimination.

Suppose we have a dataset containing 1000 traces. The D&C method first partitions this dataset into smaller subsets until each subset is small enough for efficient processing. For instance, if the split window parameter allows a maximum of 200 traces per batch, the dataset would initially be split into two equal parts, each containing 500 traces. This recursive partitioning continues until each subset falls below the threshold.

The LTL synthesis process generates a large number of candidate formulas, requiring an efficient redundancy elimination strategy. To achieve this, the authors implemented the following strategies:

- Observational equivalence: if two formulas classify the traces identically, they are considered equivalent and one is discarded;
- Relaxed Uniqueness Checks (RUCs): instead of strict formula uniqueness, RUCs use hash-based caching and probabilistic admission to reduce redundant formula storage;
- Syntax-based simplifications: applying rules like
$$F(F(\varphi)) \to F(\varphi), \quad \neg(\neg\varphi) \to \varphi, \quad G(\varphi) \wedge \varphi \to G(\varphi);$$
- Cost-based approximation: the system stops at the first ''good enough'' formula instead of exhaustively finding the absolute minimal one.

7) PERMUTATIONS IN ACTIVE MINING

Li et al. [115] propose an active mining process to solve the problem of insufficient environmental assumptions when synthesizing a controller from a set of specifications: this issue often leads to specifications being unrealizable. The methodology involves a mining procedure using predefined templates with placeholders for the design's Boolean variables. By generating unique instantiations of these templates, the approach evaluates them against the counter-strategy and user scenarios to find valid assumptions; then, the new assumptions are tested for consistency with the older assumptions to ensure they effectively eliminate the counter-strategy. We report further details hereafter. When a spec-

ification is unrealizable, the approach employs synthesis tools to compute a counter-strategy. This counter-strategy (typically represented by a Mealy machine) acts as a witness to unrealizability by demonstrating how the environment can provide inputs that cause the system to fail to meet the specification. Using the counter-strategy, new environmental assumptions are mined and added to the specifications to block the environment's ability to violate it; this is achieved by adding the negation of the newly mined properties to the environmental assumptions, effectively ruling out the environmental moves that led to the violation. The process is repeated with the new assumptions added, and the realizability of the specification is re-checked. If the specification remains unrealizable, a new counter-strategy is generated, and the process continues until the specification becomes realizable or no further valid assumptions can be found.

8) BASIC PERMUTATIONS OF INVARIANTS

We conclude this section with the most relevant applications of placeholder permutations technique to generate non-temporal invariants.

One of the first attempts to automatically generate invariants from HW designs is reported in [100], where the authors propose a tool called IODINE. The tool implements a preprocessing phase to reduce the complexity of the subsequent analysis by eliminating signals that I) do not contribute meaningful information, such as constants that never change their value throughout the simulation, II) are always equal or complements of each other throughout the simulation, such as equal signals, in this case, only one is kept. The tool uses predefined templates for various types of invariants commonly seen in hardware designs. These templates include patterns like one-hot encoding, mutual exclusion and request-acknowledge pairs. For each type of invariant template, IODINE considers permutations of the design signals to check if they match the predefined templates on the input traces. Depending on the type of templates, the approach proposes several analyzers implementing heuristics to reduce the number of permutations to be checked. For example, the $Req \rightarrow Ack$ analyzer identifies pairs of signals that follow request-acknowledge patterns. The tool identifies ''req'' and ''ack'' candidate signals depending on their activity frequencies: the set of candidates is computed by identifying pairs of signals that are active for the same number of clock cycles.

9) GENERATION OF INVARIANTS USING DAIKON

Daikon [69], [155] is the most successful tool for dynamic invariant detection, with over 1000 overall citations. It has been used in a wide range of applications, including software testing, debugging, and verification; several works use Daikon as a building block for their own tools. Daikon computes likely invariants at each ''procedure exit'' and generalizes them across all exit points to create aggregate invariants, which represent the overall behavior of the procedure; furthermore, Daikon generalizes over all observed objects at entry and exit from public methods, or when objects are passed to or returned from other methods. Daikon implements a generate-and-check algorithm to test potential invariants against the execution traces. Initially, it assumes all potential invariants are true and tests each one against each sample in the traces. Any invariant contradicted by a sample is discarded. At the end of the process, the remaining invariants that are satisfied on the entire input traces are reported. Daikon mines invariants following over 75 templates, some of which are reported below.

- Constant Value invariants: $x = a$
- Non-zero invariants: $x \neq 0$
- Range invariants: $a \leq x \leq b$
- Linear relationships: $y = ax + b$

Since the number of potential invariants can be large, Daikon employs a variety of heuristics to reduce the search space and focus on the most promising candidates. We report such heuristics hereafter.

- Equal variables: if two or more variables are always equal, then any invariant true for one is true for all.
- Dynamically constant variables: a variable that has the same value in all observed samples can simplify other invariants, reducing redundancy. For instance, if $x = 5$, then $x < y$ is redundant if y is known.
- Variable hierarchy: values observed at certain program points affect invariants at multiple points; that is, values observed at method exits affect both method post-conditions and object invariants. For example, in a Java program with an Account class with a *balance* variable, observations of balance at method exits such as *deposit* and *withdraw* contribute to detecting post-conditions like $balance == original(balance) + amount$ for deposits, and $balance == original(balance) - amount$ for withdraws. These observations help establish general object invariants such as $balance >= 0$ for all public method entries and exits.
- Suppression of weaker invariants: an invariant that is logically implied by a stronger one is suppressed to avoid redundancy and improve performance. For instance, $x > y$ implies $x \geq y$, so the latter is not reported separately.

These optimizations allow Daikon to scale effectively to non-trivial programs, reducing the number of invariants that need to be checked by up to 99%.

10) PARALLEL GENERATION OF INVARIANTS USING GPUs

Bombieri et al. [93] propose a tool called Mangrove [156] to dynamically mine likely invariants from execution traces by leveraging parallel programming and inference rules. The users can define invariant templates, which are logical or arithmetic formulas over the system's variables. Examples of templates include relations like $P0 = P1 + P2$ or boolean logic such as $P0 \wedge P1$. After that, Mangrove generates dictionaries D_k containing permutations of k

variables from the execution trace, covering all possible combinations needed for evaluating all candidate invariants. Furthermore, mangrove applies inference rules to filter out redundant entries in the dictionaries and significantly reduces the size of the search space. The parallel evaluation of candidate invariants is performed by GPU threads, which evaluate invariants over a chunk of the execution trace. This enables massive parallelism and efficient data handling and processing. The approach implements several well-known optimization techniques, as reported below.

- Memory coalescing: ensures efficient memory access patterns, which is crucial for performance on GPU architectures;
- Thread synchronization: optimizes thread coordination to minimize overhead and maximize parallel efficiency;
- Kernel auto-tuning: automatically adjusts parameters for optimal GPU utilization based on the characteristics of the target device.

Mangrove exploits inference rules to reduce the number of candidate invariants that need to be evaluated. The inference rules are used to eliminate redundant invariants based on the characteristics of the input data. In particular, Mangrove employs a bottom-up approach, starting with low-order invariants and progressively checking higher-order expressions. Consider the following example where the input trace contains design variables $v1, v2, v3$. If the occurrences show: $v1 = true$ for all instants, $v2 = v3$ for all instants, $v1 = v2 \vee v3$ for all instants; then, the inferred invariants are $v1 = true$, $v2 = v3$, $v1 = v2 \vee v3$. To understand how these relationships are inferred without needing to check every permutation of variables in the templates, the following steps are executed. First, the tool identifies single variable invariants; for example, if $v1$ is true in all instances of the trace, $v1 = true$ is inferred. After that, the tool identifies pairwise invariants; for example, If $v2$ and $v3$ have the same value in all instances, the $v2 = v3$ is inferred. Then, Mangroove uses the previously inferred single and pairwise invariants to check for higher-order relationships. For example, given $v1 = true$ and $v2 = v3$, we can check if $v1$ equals $v2 \vee v3$. Since $v1 = true$, and $v2 \vee v3$ will always be true if $v2$ and $v3$ are equal, we infer that $v1 = v2 \vee v3$ is always true without needing to test it on the trace.

C. ASSOCIATION RULES AND SEQUENCE MINING

This section covers all techniques involving association rules and sequence mining. These two families of techniques are reported in the same section due to their technical similarities.

1) FUNDAMENTALS OF ASSOCIATION RULES AND SEQUENCE MINING

Association rule mining is a data mining technique used to discover interesting relationships, patterns, or associations among a set of items in large databases. This process involves identifying frequent itemsets and generating rules highlighting the likelihood of certain items appearing together. Most association rule mining approaches encompass two main steps reported below.

- **Frequent itemset mining**: the initial step involves finding groups of items (itemsets) that appear frequently together in transactions. Transactions are the individual data records appearing in a dataset. For instance, in a retail scenario, a transaction might represent a single customer's purchase, listing all items bought during that purchase. Frequent items are usually generated using algorithms such as Apriori [157] or FP-Growth [158]. An item is considered frequent if it meets a user-defined minimum support threshold, which indicates the proportion of transactions in which the item appears.
- **Association rule generation**: the second step is to generate association rules. These rules take the form of "if-then" statements such as $A, B \rightarrow C$, meaning "if items A and B appear together, then item C is likely to be related". To be considered relevant, a rule must meet minimum support and confidence thresholds.

Similarly to association rule mining, sequential pattern mining identifies statistically significant patterns in sequences of events or items. For example, sequential pattern mining is employed to capture the order of operations or procedure calls within a program, reflecting the sequence dependencies that must be followed. The goal is to find subsequences that appear frequently across different sequences in the dataset. A subsequence is considered frequent if it occurs in a minimum fraction of the sequences, as defined by a user-specified support threshold. Several association rule mining algorithms are used to perform sequential pattern mining; however, the key difference is that sequential pattern mining considers the order of items in the sequences, while association rule mining does not.

The most famous sequential pattern miners are Generalized Sequential Pattern (GSP) [159], FP-Growth (Frequent Pattern Growth), SPADE [160] (Sequential Pattern Discovery using Equivalent Classes) and BIDE [138] (Bidirectional Extension for Frequent Closed Sequence Mining). The specification mining techniques reported below mainly employ Apriori (and its variations), GSP and PrefixSpan.

GSP is a general sequence mining approach for discovering all frequent sequences in a sequence database. The approach is based on several Apriori-based algorithms adapted to work with sequences. The main algorithms are AprioriAll, AprioriSome, and DynamicSome. To build a minimal background on this family of techniques, we report in more detail the AprioriAll algorithm (Apriori applied to sequences), which can be summarized with the steps below.

1) Identify frequent 1-sequences (single items) by scanning the database and counting the support of each item, i.e., the number of transaction sequences that contain the candidate sequence. Note that a subsequence is considered supported by a transaction sequence if it appears in the same order but not necessarily consecutively. This means you can skip elements in

between. For example, subsequence (A D) is supported by (A B C D) because both A and D appear in the same order.

2) Generate candidate k-sequences from frequent $(k-1)$-sequences by joining $(k-1)$-sequences that share a common prefix. For example, the 2-sequence (A B) can be joined with (A C) to form the candidate 3-sequence (A B C) because they share the same first element A. The reason for using this joining approach is based on the Apriori principle: if a k-sequence is frequent, all of its (k-1)-length subsequences must also be frequent. By joining sequences with a common prefix, the algorithm ensures that only those candidate sequences that have a chance of being frequent are generated.
3) Prune the k-candidates: for each candidate k-sequence, check all of its (k-1)-subsequences. A (k-1)-subsequence is any subsequence formed by removing one itemset from the candidate sequence. If any of these subsequences is not in the set of (k-1)-candidates, then the candidate sequence is not considered frequent enough and is removed. The reason for this pruning approach is also justified by the Apriori principle.
4) Scan the database to count the support of each candidate k-sequence, and prune the candidates that do not meet the minimum support threshold.
5) Repeat the candidate generation, support counting, and pruning steps for increasing values of k (sequence length) until no more frequent sequences are found.
6) After identifying all frequent sequences, prune non-maximal sequences, which are subsequences of longer frequent sequences. This is usually done through algorithms involving hash or prefix tree data structures.

We report an example of the AprioriAll algorithm below.

Customer ID	Transactions
1	⟨(A, E), C, D⟩
2	⟨A, C, D, (C, E)⟩
3	⟨A, C, D⟩
4	⟨A, C, E⟩
5	⟨D, E⟩
6	⟨B⟩

Each row reports the sequence of purchases made by each customer. For example, in the first row, ⟨(A,E), C, D⟩means that customer 1 bought A and E together, then he bought C, then he bought D.

The minimum support threshold is 3. First, generate frequent 1-sequences by scanning the database to count the support of each item and filter those that do not satisfy the minimum threshold: ⟨A⟩: 4, ~~⟨B⟩: 1~~, ⟨C⟩: 4, ⟨D⟩: 4, ⟨E⟩: 4; here ⟨A⟩is supported by four sequences with Customer ID 1,2,3, and 4.

Then, generate candidate 2-sequences by joining frequent 1-sequences, count their support, and filter the sequences with support lower than 3. The resulting sequences are ⟨A,C⟩: 4, ⟨A,D⟩: 3, ⟨A,E⟩: 3, ⟨C,D⟩: 3, ~~⟨C,E⟩: 2~~, ~~⟨D,E⟩: 2~~, where the eliminated candidates do not meet the minimum support.

After that, generate candidate 3-sequences by joining frequent 2-sequences: ⟨A,C,D⟩: 3, ~~⟨A,D,C⟩~~, ~~⟨A,C,E⟩~~, ~~⟨A,E,C⟩~~, ~~⟨A,D,E⟩~~, ~~⟨A,E,D⟩~~, where the eliminated candidates were pruned according to step 3 of the algorithm.

Finally, discard non-maximal sequences by checking each frequent candidate to determine if it is a subsequence of any longer sequence: sequences ⟨A,C⟩, ⟨A,D⟩, and ⟨C,D⟩are subsequences of ⟨A,C,D⟩; ⟨A,E⟩and ⟨A,C,D⟩are not subsequences of any longer sequence; therefore, they are the maximal subsequences.

2) ASSOCIATION RULE MINING WITH TRACE TEMPORAL ALIGNMENT

One of the latest works on applying association rule mining to generate LTL specifications is reported in [20], where Heidari et al. propose a tool called ARTmine [161]. The tool mines specifications of the forms:

- $G(P0 \rightarrow \mathit{consequent})$, where the consequent is of the form $X[k]P1$ (with $k \geq 0$) or $F(P1)$;
- $G(P0\ U\ P1)$.

where $P0$ and $P1$ are propositions; typically, P0 and P1 contain input and output variables, respectively. The proposed methodology works in several steps, as reported below. First, the tool applies association rule mining to identify frequent combinations of propositions. To achieve that, the input trace is transformed to temporally align the trace's variables with respect to one of the aforementioned templates. For example, when considering the *next* template $G(P0 \rightarrow X[N]P1)$, the values of the candidate output variables for $P1$ are moved N records ahead in the trace to align them with the values of the input variables of $P0$. After that, the tool applies the Apriori algorithm (or other similar approaches such as FP-growth) to generate association rules outlining meaningful non-temporal relations in the transformed trace such as $(in_a, in_b) \rightarrow (out_c)$. Then, the tools perform a time-notation step to integrate the concept of time into the association rules generated in the previous step to produce temporal association rules. This involves assigning time labels to the rules based on their template. For example, if the approach mines the association rule $(in_a, in_b) \rightarrow (out_c)$ from a trace where variable out_c was shifted forward by two instants (next template), the rule would be labeled with "$@X[2]$". Finally, the tool transforms the temporal association rules into temporal assertions using the SVA format according to the labeled association rules. For example, the rule $(in_a, in_b) \rightarrow (out_c)@next[2]$ would be converted to the SVA $always(in_a \,\&\&\, in_b \rightarrow nexttime[2]out_c)$.

3) ASSOCIATION RULE MINING TO GENERATE INVARIANTS

Danese et al. [21] proposed a tool called A-TEAM to mine LTL assertions with user-customizable templates. The first step of the tool applies association rule mining to generate

propositions of the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$; then, it composes the propositions into LTL expressions using a different technique, in this section, we focus only on the association rule mining aspect. To generate the aforementioned output, the algorithm takes as input a set of user-defined propositions called *apSetOut* and iterates over every simulation instant of the input trace to collect itemsets: an itemset contains a list of atomic propositions of *apSetOut* that hold true at specific instants of the trace. After that, the Apriori algorithm is employed to identify itemsets that frequently appear in the collected data. Finally, the itemsets are converted to propositions. Below is an example to help you better understand the process. Consider the following input trace

t	v1	v2	v3	v4	v5	v6	v7
0	false	001	3	2	01	false	false
1	true	001	3	2	01	false	false
2	true	011	3	3	01	false	true
3	true	111	3	3	01	false	true
4	false	001	4	3	01	true	false
5	false	010	4	3	10	true	false
6	false	011	4	4	10	true	true
7	false	111	4	4	10	true	true
8	false	001	5	4	10	true	false
9	false	010	5	4	10	true	false

with $apSetOut = \{v3 = v4, v5 = 01, v5 = 10, v6 = true, v6 = false, v7 = true, v7 = false\}$. First, the tool collects itemsets for each simulation instant resulting in:

```
S0: {v5=01, ¬v6, ¬v7}
S1: {v5=01, ¬v6, ¬v7}
S2: {v3=v4, v5=01, ¬v6, v7}
S3: {v3=v4, v5=01, ¬v6, v7}
S4: {v5=01, ¬v6, v7}
S5: {v5=10, v6, ¬v7}
S6: {v3=v4, v5=10, v6, v7}
S7: {v3=v4, v5=10, v6, v7}
S8: {v5=10, v6, v7}
S9: {v5=10, v6, ¬v7}
```

These itemsets are collections of *apSetOut* propositions that hold true at each specific simulation instant. For example, at $t = 0$, the itemset S0 includes $v5 = 01$, $\neg v6$ and $\neg v7$, since $v6$ is false, $v5$ is equal to 01, and $\neg v7$ is false at time 0. After that, the Apriori algorithm identifies the following frequent itemsets: $v5 = 01, \neg v6$, $v5 = 01, v6$, $\neg v7$, $v3 = v4, v7$. These itemsets frequently appear across the execution trace. For instance, $v5 = 01, \neg v6$ frequently appear, indicating that $v5 = 01$ and $v6$ being false is a common condition. Finally, the following propositions are generated from the itemsets: $v5 = 01 \wedge \neg v6, v5 = 01 \wedge v6$, $\neg v7$, $v3 = v4 \wedge v7$. Cheng and Hsiao [25] propose a similar approach that employs the Apriori algorithm to generate invariants of the form $(p_1 \wedge p_2 \ldots \wedge p_n) \rightarrow (p_1 \wedge p2 \ldots \wedge p_m)$ where each p_i is a Boolean variable.

4) STATIC MINING WITH ASSOCIATION RULES AND SEQUENCES

One of the earliest attempts at statically mining specification using sequence mining and association rules is proposed by Ramanathan et al. [118]. The approach analyses the source code of the design to identify conditions that must hold true at various points in the execution. This process is based on a combination of data-flow and control-flow analysis. Two main types of predicates are considered: I) data-flow predicates to describe conditions on the state of variables, for example, a variable or reference being written or an expression that reflects the outcome of a conditional statement; II) control-flow predicates to describe the sequence of operations or procedure calls, for instance, they indicate that a specific procedure must be called before another procedure. In particular, predicates are collected at each program point through a set of rules applied to the Control Flow Graph (CFG) of each procedure. The CFG is traversed to identify the points where variables are read, written, allocated, and where procedure calls are made. While traversing the CFG, the order in which control-flow predicates appear is recorded: this order is essential because it reflects the sequence of operations in the program. Each path in the CFG may yield a different sequence of control-flow predicates, capturing different execution scenarios. After this step, the collected predicates are combined and refined using data mining techniques to identify the most significant ones. The process follows two different paths.

1) If the ordering of events is not important, apply an algorithm called "Maximal frequent itemset mining" to identify data-flow predicates that occur frequently across different call sites. That algorithm extends Apriori by focusing on identifying the largest frequent itemsets that are not subsets of any other frequent itemset for reducing redundancy and simplifying the analysis; see [162] for more info on this.
2) Otherwise, if the ordering matters, the approach employs GSP to capture the order of control-flow predicates, making the derived specifications reflect the correct sequence of operations.

5) SEQUENCE MINING WITH PREFIXSPAN

Lo et al. [117] proposed a variant of the PrefixSpan algorithm to generate sequential specifications of the form *always*(*a* `followed by` *b* `followed by` *c* ...). PrefixSpan improves on a previous algorithm called FreeSpan [163]; FreeSpan projects sequence databases by considering all the possible occurrences of frequent subsequences; instead, PrefixSpan focuses only on frequent prefixes. The original PrefixSpan can be summarized as follows.

1) Find length-1 sequential patterns. Scan the sequence database to find all frequent items (length-1 sequential patterns). Each item is considered frequent if its support is no less than the minimum support threshold. In this context, the support is the number of sequences in the database containing the item. For example, if the sequences are ⟨A, B, C, A⟩, ⟨A, C, D⟩, and ⟨B, D, A, E⟩, and the minimum support is 2, then the frequent items are ⟨A:3, B:2, C:2, D:2⟩, ⟨E⟩ is discarded since it has support equal to 1 (lower than the minimum).

The complete set of sequential patterns is partitioned into subsets according to the prefixes found in the previous step; following from the example above, the search space is divided into the sequential patterns having ⟨A⟩ as a prefix, those having ⟨B⟩ as a prefix and so on.

2) Recursive step: project and grow. For each new frequent prefix, construct a projected database that includes all suffixes of the sequences containing the prefix. For example, for the prefix ⟨A⟩ and the sequences above, the projected database includes the suffixes of sequences after the first occurrence of ⟨A⟩, which are ⟨B, C, A⟩, ⟨C, D⟩, and ⟨E⟩. Terminate the procedure for the current prefix if it does not meet the minimum support. In this case, the support of the prefix is the number of sequences in the projected database, for prefix ⟨A⟩ in the example above the support would be 3. Recursively apply step 3 by growing the current prefix with one element at a time until no more extensions are possible (i.e., no further frequent items can be appended). In the example, prefix ⟨A⟩ can be extended to ⟨A, B⟩, ⟨A, C⟩ and ⟨A, E⟩. All prefixes that satisfy the minimum support are added to the set of mined frequent sequences.

By systematically applying the steps above, the PrefixSpan algorithm efficiently mines a complete set of sequential patterns, ensuring that the search space is pruned at every stage to retain only frequent sequences.

The paper proposes the following variations with respect to the original PrefixSpan to mine specifications.

- Instead of the usual projection, the authors propose a "projected-all" database that captures all occurrences of the pattern, even if they overlap or are nested within one another; in other words, the algorithm handles repeated occurrences within sequences by tracking multiple instances of a pattern within a single sequence. This allows the algorithm to handle the repeated nature of iterative patterns.
- Use of a pruning mechanism that maintains infix extensions. Infix Extensions are events that can be inserted within a pattern while maintaining its support. For example, for pattern ⟨A, C⟩, if inserting B between A and C yields a frequent pattern ⟨A, B, C⟩ with the same support, then B is an infix extension.
- Application of closure check mechanism to ensure that a sequence is closed: no super-sequences have the same support.

6) ASSOCIATION RULE MINING AND INPUT SPACE COVERAGE

Liu et al. [24] describe an atypical association mining approach that does not involve the traditional algorithms described above.

The approach defines a coverage metric called "input space coverage" to evaluate how much of the possible input combinations are covered by an assertion. The approach attempts to extract all possible correlations between trace variables of the form $(in_1 \wedge in_2 \ldots \wedge in_n) \rightarrow out$ where each in_i is an input Boolean variable, and out is an output Boolean variable; the process generates every assertion that complies with the given input trace and the coverage metric. In practice, the values of the boolean input variables over time (the input trace is a time series) are treated as a truth table, and the coverage of an assertion is the proportion of rows covered. The process is iterative, starting with high coverage constraints (50%) and gradually lowering (halving them) them to discover more assertions. The algorithm terminates when 100% coverage has been reached or when the maximum number of iterations has exceeded a threshold. To clarify the procedure, consider the example reported below, starting from an input trace with three input variables a, b, c and one output variable z.

Time	a	b	c	z
1	0	0	0	0
2	0	1	0	0
3	1	0	1	0
4	1	1	1	1

Set the initial coverage gain constraint at 0.50 (50%). Identify all assertions Ai such that the coverage of Ai is at least 0.5. According to the alphabetical order of variables, A1: $\neg a \rightarrow \neg z$ is the first generated, which covers 50% of the input space: a is equal to 0 (false) in half of the truth table entries. Assertions A2: $\neg b \rightarrow \neg z$ and A3: $c \rightarrow \neg z$ are excluded in the first iteration as they can only increase the coverage by 25% above A1, respectively. For the second iteration, the coverage gain constraint is lowered to 0.25 (25%), and additional assertions that meet the new coverage gain constraint are identified. Assertion A2 covers an additional 25% of the input space and is included in the set of mining specifications; however, A3 is still excluded from the solution assertions since the coverage gain is only 12.5% above A1 and A2. Finally, in the second iteration, assertion A4: $a \wedge b \rightarrow z$ is discovered, which covers the remaining 25%.

D. DECISION TREE

Decision Tree (DT) mining is a technique that leverages execution traces to infer logical relationships between input variables and their effects on output variables. The fundamental idea is to build a DT that models the behavior of the design by recursively partitioning the input data into subsets based on the available feature values. In this section, we will describe three main approaches (and their evolution) implemented in three publicly available tools: Goldminer [164], HARM [152], and samples2LTL [165].

1) FUNDAMENTALS OF DECISION-TREE MINING

Vasudevan et al. introduced Goldminer in [62] and [63]; the tool utilizes DT-based mining and static analysis to automatically generate SVAs of the form $G\ (bv_1 \wedge$

$X[1](bv_2) \wedge \ldots \wedge X[N](bv_m) \rightarrow X[M](bv_k))$ where each bv_i is a boolean variable representing one bit of a variable/signal of the Register Transfer Level (RTL) design under verification, M and N are positive integers. The authors refer to bv_k as the "target signal". By leveraging simulation traces, the tool infers logical relationships between input/output variables. This work establishes the feasibility of using DTs for assertion generation and demonstrates the initial effectiveness of this approach.

As implied by the name, the algorithm employs a DT procedure where each decision consists of choosing a new split variable (a DUV variable) and adding a new proposition (either the split variable in its positive form or its negated version "$\neg variable$") to the DT expression. Split variables are chosen according to their Information Gain (IG), that is, the expected reduction in information entropy caused by adding them to the antecedent of the expression. Information entropy can be thought of as the amount of variance in a dataset. In our scenario of application, the entropy is maximum (1) when each time the antecedent is true, we have a 0.5 probability of having the consequent being also true; conversely, the entropy is minimum (0) when each time the antecedent is true, then the consequent is always true or always false. To be more precise, specification miners use "conditional entropy" often formalized with $H(Y \mid X) = \sum_{i=1}^{2} p(x_i)H(Y \mid X = x_i)$ where $H(Y \mid X)$ measures the entropy of a target variable Y (which represents the truth values of the consequent) given that we know the value of another variable X (which represents the truth values of the antecedent after adding the split variable); x_1 and x_2 are the positive and negated split variables, respectively.

The algorithm chooses the variable(s) that produces the highest reduction in entropy (the highest IG) to mine specifications by using as few variables as possible, avoiding over-constrained expressions.

We report a minimal but complete example below. Consider the following dataset S with two input variables a and b, and one output variable f.

```
a      0 0 0 1 1 1
b      0 1 1 0 1 1
f      0 0 1 1 0 1
time   0 1 2 3 4 5
```

First, we calculate the entropy $H(S) = -\sum_{i=1}^{2} p(s_i) \log_2 p(s_i)$ of the entire dataset (the entropy of the consequent). There are two classes (false: 0 and true: 1) for the output variable f. From the dataset, we have that $f\ (s_i = f)$ is true 3 times ($\frac{3}{6} = 0.5$) and $f\ (s_i = \neg f)$ is false 3 times ($\frac{3}{6} = 0.5$). Entropy of the entire dataset S is $H(S) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1$. The initial mined specification looks like $G(? \rightarrow f)$. Next, we split the dataset based on variable a and calculate the entropy of each subset. We show below the subset of the dataset where $\neg a$ is true.

```
a      0 0 0
b      0 1 1
f      0 0 1
time   0 1 2
```

The proportions of times in which antecedent true implies consequent false with $\neg a$ is $P(Y \mid X = \neg a) = \frac{2}{3} = 0.67$, and for $P(\neg Y \mid X = \neg a) = 1 - P(Y \mid X = \neg a) = 0.33$. Therefore, $H(S_{\neg a}) = -(0.67 \log_2 0.67 + 0.33 \log_2 0.33) = 0.915$

Below, we show the subset where a is satisfied.

```
a      1 1 1
b      0 1 1
f      1 0 1
time   3 4 5
```

The proportions of times in which antecedent true implies consequent false with a is $P(Y \mid X = a) = \frac{1}{3} = 0.33$, and for $P(\neg Y \mid X = a) = 1 - P(Y \mid X = a) = 0.67$. Therefore, $H(S_a) = -(0.33 \log_2 0.33 + 0.67 \log_2 0.67) = 0.915$. Finally, to determine the entropy after choosing split variable a, we need to do a weighted sum of $H(S_a)$ and $H(S_{\neg a})$. The weight is the $p(x_i)$ part of the conditional entropy formula; that is, the proportions of instances in which variable x_i is true in the dataset. The weighted entropy for split on a is $H(S|a) = \frac{3}{6}H(S_{a=0}) + \frac{3}{6}H(S_{a=1}) = 0.916$.

If we repeat the procedure for variable b, we obtain $H(S|b) = 1$. Finally, we compute the information gain. For splitting on variable a, the IG is $H(S) - H(S|a) = 1 - 0.916 = 0.084$, for splitting on b the IG is $H(S) - H(S|b) = 1 - 1 = 0$. Since the information gain for splitting on a is higher than for splitting on b, a is the better choice for splitting the dataset; therefore, the obtained expression after one choice is $G(a \rightarrow f)$

The DT approach was extended [64] by introducing the concept of Best-Gain Decision Forest (BGDF). DT algorithms select one feature variable with maximum information gain to partition the dataset. In contrast, the BGDF algorithm partitions the dataset using all variables with the best IG.

2) MINING WORD-LEVEL SPECIFICATIONS WITH DECISION TREES

Goldminer was extended in [68] to integrate static analysis more deeply into the assertion mining process with the main goals of I) better guiding the selection of features to reduce spurious assertions I) addressing the limitations of bit-level assertions by proposing a methodology for discovering word-level features.

In particular, static analysis focuses on the most relevant features within the source code of the RTL design that are likely to influence the behavior of the target signals. This process involves the following key aspects.

- Cone of Influence (COI) analysis: this process identifies the subset of signals that can affect one another during execution. This analysis allows us to preemptively infer chains of "cause-effect" patterns, greatly reducing the search space of the algorithm. For example, if the target signal is the output of a multiplier in the arithmetic unit,

the COI analysis would identify all input signals and intermediate signals (such as the results of preceding addition operations) that could influence this output. Signals outside this influence cone, such as unrelated control signals, would be excluded from consideration. From a technical point of view, this process requires building and analyzing the Control Data Flow Graph (CDFG) of the DUV.

- Use-Definition Chain (UD-Chain) analysis: for each signal within the COI, the tool traces its use-definition chain to single out all the definitions (assignments) of a variable that can reach a particular use without any intervening redefinitions. This helps in understanding the propagation of values through the design and identifying critical points where assertions can be generated. For example, in the arithmetic unit, if a register R holds an intermediate result used in the target multiplication, the UD-chain analysis would trace all assignments to R and ensure that only the relevant definitions (those that can reach the multiplication operation) are considered. If R is redefined in another part of the design, those definitions are excluded.
- Path condition analysis: the tool performs path condition analysis to determine the conditions under which different paths in the CDFG are executed. This analysis helps in understanding the logical conditions that lead to particular states or behaviors in the design. Path conditions are used to refine the selection of features by focusing on those that are relevant under specific conditions. For example, in the arithmetic unit, suppose there is a condition that determines whether the multiplier operates in signed or unsigned mode. Path condition analysis would identify this condition and ensure that assertions are generated considering both modes. For instance, assertions might specify that under the signed mode, certain input ranges produce specific output ranges.

The tool ranks the features in the DT according to their relevance and impact on the target signals: features more influential in determining the target signal behavior are given higher priority.

In single-bit assertion generation, the target boolean variable *bv* can have one of two values: 0 and 1. Therefore, the proposition in the consequent for any bit-level target variable can only be $b = 0$ or $b = 1$. However, at the word level, the variables can have many possible values (bit-vectors). Deciphering all these values may lead to too many assertions, many of which could be irrelevant. Hence, the word-level predicate itself is provided as a target for the learning algorithm. In other words, the word-level expression is given to the DT as an additional complex proposition.

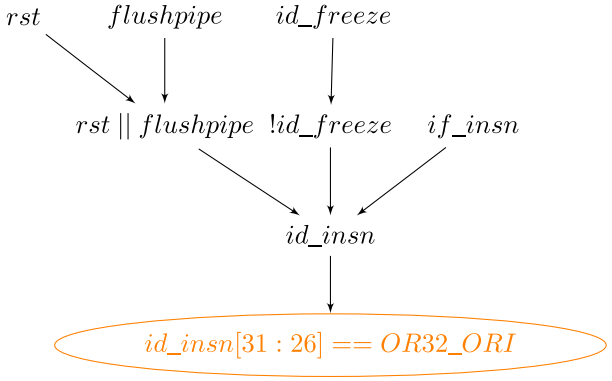
Here is how Goldminer performs word-level feature discovery. To simplify the exposition, we will refer to a running example involving the Verilog RTL design shown in Listing 1.

```
1  module or1200_ctrl(clk, rst, ex_freeze,
2     id_freeze, flushpipe, if_insn, alu_op);
3
4    input rst, clk, ex_freeze, id_freeze, flushpipe;
5    input [31:0] if_insn;
6    output [3:0] alu_op;
7    reg [31:0] id_insn;
8    reg [3:0] alu_op;
9
10   always @(posedge clk) begin
11     if (rst || flushpipe) begin
12       id_insn <= {6'h5, 26'h0410000};
13     end else if (!id_freeze) begin
14       id_insn <= if_insn;
15     end else begin
16       id_insn <= id_insn;
17     end
18   end
19
20   always @(posedge clk) begin
21     if (rst) begin
22       alu_op <= `ALU_NOP;
23     end else if (!ex_freeze &
24                  id_freeze | flushpipe) begin
25       alu_op <= `ALU_NOP;
26     end else if (!ex_freeze) begin
27       case (id_insn[31:26])
28         `OR32_J: alu_op <= `ALU_IMM;
29         `OR32_ORI: alu_op <= `ALU_OR;
30         `OR32_ADDI: alu_op <= `ALU_ADD;
31       endcase
32     end else begin
33       alu_op <= `ALU_NOP;
34     end
35   end
36 endmodule
```

LISTING 1: Source code of the Goldmine running example.

The procedure follows the steps reported below.

1) Identifying word-level targets. The first step involves identifying word-level output variables in the RTL code that are assigned constant values. These assignments are analyzed to determine word-level predicates that serve as targets for assertion generation. In the running example, variable *alu_op* is assigned constant values *ALU_NOP*, *ALU_IMM*, *ALU_OR*, and *ALU_ADD*; therefore, the following assignments could be used as word-level targets: *alu_op* == *ALU_NOP*, *alu_op* == *ALU_IMM* and so on.
2) Discovering word-level features. The process considers all word-level conditional expressions within the logic cone of the targets identified in the previous step. In the running example, consider the COI for the target *id_insn*[31 : 26] == *OR32_ORI*.



In this case, we would have to consider all conditions and signals that might affect the target, such as *rst* || *flushpipe*, !*id_freeze*, *if_insn*, and so on.

After that, instead of using all predicates in the COI, the process refines them using a simulation-guided Weakest Precondition (WP) computation. The purpose of identifying the WP is to efficiently determine the minimal set of conditions necessary for the target to occur. The WP computation refines the set predicates to those that are strictly necessary, leading to a simpler and more efficient analysis.

To compute the WP, first determine the initial conditions under which the target predicate is true. That would be $id\_insn[31:26] == OR32\_ORI$ in the running example. The paper proposes both a static and a simulation-based version to compute the WP. However, the static version suffers from scalability issues; therefore, we describe only the simulation-based version below.

First, run simulations on the RTL design using directed or random testbenches (in literature, symbolic simulation is often employed instead to quickly generate high-coverage tests) to generate simulation traces and record the concrete paths taken during the simulation. Each path consists of a sequence of executed statements. For example, consider the following path, where each number is the line of the executed statement in Listing 1.

```
Clock    First        Second
cycle    block        block
1        (11,12)      (21,22)
2        (11,13,14)   (21,23,26,32)
3        (11,13,16)   (21,23,26,27,29)
```

In particular, in the first cycle, the reset is high. In the second cycle, register *id_insn* is loaded with the value of input *if_insn* (line 16). Finally, *ex_freeze*, *id_free*, and *flushpipe* are all low, and line 29 is reached in the second block.

After that, for each simulation path that reaches the target predicate (line 29 in the running example), perform backward substitution to trace the conditions leading to the target predicate. That involves identifying the assignments to the variable in the predicate and substituting the assignments backward along the simulation path to express the predicate in terms of earlier states or inputs.

First, identify the initial predicate at cycle 3. The target predicate is $id\_insn[31:26] == OR32\_ORI$ which requires !*rst*, !(*id_freeze* || *flushpipe*) and !*ex_freeze* to be true. Next, perform backward substitution for cycle 2. *id_insnid* is assigned the value of *if_insnif* at line 14 if !(*rst* || *flushpipe*) and !*id_free* are true. Moreover, for $id\_insn[31:26] == OR32\_ORI$ to be true in cycle 3, $if\_insn[31:26] == OR32\_ORI$ must be true in cycle 2.

The procedure terminates here since the target predicate is completely justified using conditions from the inputs. The final conditions are

$$!(rst \,||\, flushpipe) \;\&\&\; !id\_freeze \;\&\&\; if\_insn[31:26] == OR32\_ORI$$

followed by

$$!rst \;\&\&\; !(id\_freeze \,||\, flushpipe) \;\&\&\; !ex\_freeze$$

in the next cycle.

3) Removing redundant propositions. After discovering word-level features, a post-processing step is applied to remove redundant propositions. This involves checking for mutually exclusive features and eliminating overlaps that might lead to over-constrained or meaningless assertions. For example, if features like $state[15:0] = S1$ and $state[15:0] = S2$ are mutually exclusive, the proposition $!(state[15:0] = S2)$ can be removed to simplify the assertion.

Finally, the simplified word-level features and targets are instrumented back into the RTL code, which is then re-simulated to generate new traces. These traces are fed into the DT algorithm to generate word-level assertions following the aforementioned approach. For example, if the target signal is *alu_op*, and the word-level feature discovered is $id\_insn[31:26] == OR32\_ORI$, after re-simulation, the new trace will contain an additional split variable reporting the values for each clock cycle of $if\_insn[31:26] == OR32\_ORI$.

3) DECISION TREES AS OPERATORS

With respect to Goldminer, HARM [61] proposes a more flexible approach that allows the use of the DT algorithm in more complex and customizable templated scenarios. In HARM, the user can insert "DT operators" in the antecedent of a template of the form $G(antecedent \rightarrow consequent)$ where both the antecedent and the consequent can be customized using all LTL+SERE operators. A Decision Tree Operator (DTO) is a special type of temporal (or propositional) placeholder that can be instantiated by using a DT algorithm. Currently, HARM implements three DTOs:

- ..&&.. to generate an "and expression", e.g. $v1$ && $v2$ && ... && vn
- ..##N.. to generate a "chain of nexts", e.g. $v1$ ##1 $v2$ ##1 ... ##1 vn
- ..#N&.. to generate a "chain of nexts of and expressions", e.g. $v1$ && $v2$ ##1 $v3$ && ...##1 ....

DT operators are extremely flexible: for instance, the user can apply the DT algorithm by using a complex and partially instantiated template such as $G(\{prop_1[=1]$ ##7 $prop_2$ ##1 ..&&..$)[*3]\} \rightarrow con)$ where ..&&.. is the DTO that needs to be filled through a DT algorithm and each $prop_i$ are user-defined variables. Note that by using the template $G(..\#N\&.. \rightarrow X[k](target))$, HARM generates assertions of the same form as Goldminer. Furthermore,

HARM extended the concept of BGDF by introducing a user-defined "Effort" parameter to specify how many and which split variables to choose among those that provide the best gain.

HARM was extended in [65] to automatically generate specifications containing both the temporal dimension of LTL and the complexity of non-Boolean expressions. This is similar to the extension proposed in Goldminer to mine word-level assertions; however, since HARM doesn't employ the source code of the DUV, the method exploits a pure dynamic and unsupervised method based on clustering. Starting from a set of simulation traces of the DUV, the approach generates expressions of the form $c = ne$, $c \leq ne$, $c \geq ne$, $c_l \leq ne \leq c_r$, where c, c_l, c_r are constants of numeric type, and ne is a numerical expression predicating over the variables of the DUV. These expressions are then used as new features in the DT algorithm. The entire procedure involves a few steps: extracting Interesting Values (IVs) from the trace, clustering these values using k-means [166], and determining the optimal number of clusters with the elbow method. These steps enable the translation of numerical data into logical propositions that can be used in temporal logic assertions. Below, we provide a detailed explanation of each step using a minimal example.

Consider a trace with the following values for variable a:

```
time 0 1 2 3 4  5 6  7  8 9 10  11  12
a    0 0 0 0 10 0 11 13 0 0 201 199 203
```

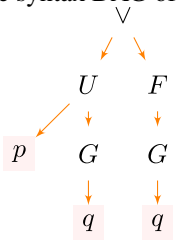
The first step is to extract IVs, which are numeric values from the trace that are crucial for forming meaningful propositions. IVs can be extracted based on their significance in the execution trace, especially focusing on points where changes in the variable a significantly impact the behavior of the system. Let us assume that from the given trace, the interesting values of a are {10, 11, 13, 201, 199, 203}.

After that, k-means is applied to partition the IVs into k clusters. Since the number of clusters is not typically known in advance, the approach employs the "elbow method" to determine it. This method involves running the k-means algorithm multiple times and measuring how the Within-Cluster Sum Of Squares (WCSS) decreases as k increases. The WCSS is usually computed as $\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in C_i} (x - \mu_i)^2$, where μ_i is the mean of the ith cluster C_i. The elbow point, where the reduction in variance plateaus, indicates a suitable number of clusters. A user-defined threshold is used to algorithmically identify this elbow; therefore, the process continues until the variance reduction falls below this threshold. In the example, we compute the WCSS for $k = 1, 2, 3$ and a threshold th equal to 0.5 (50%). For $k = 1$, all values are in one cluster. The centroid mean is $\approx$ 106.17 and the WCSS is $\approx$ 9262.03. For $k = 2$, the IVs are divided into two clusters: {10, 11, 13} and {201, 199, 203} and the WCSS is $\approx$ 12.64. For $k = 3$, the clusters are {10, 11}, {13} and {201, 199, 203} and the WCSS is $\approx$ 8.5. Reduction from $k = 1$ to $k = 2$ is significant: $\frac{54148.4 - 12.6467}{54148.4} \approx 0.9998\,(99.98\%)$; reduction from $k = 2$ to $k = 3$ is below the 50% threshold $\frac{12.6467 - 8.5}{12.6467} \approx 0.3279\,(32.79\%)$. Given the threshold th of 0.5, the reduction from $k = 2$ to $k = 3$ is not significant enough. Thus, the optimal number of clusters is $k = 2$, leading to the generation of propositions $10 \leq a \leq 13$ and $199 \leq a \leq 203$.

4) DECISION TREES AND SAT SOLVERS

Neider and Gavran [60] proposed a tool called samples2LTL/FLIE that integrates various supervised techniques to mine LTL properties from labeled traces, including SAT-based learning and DT approaches. The proposed tool tries to achieve the following goal: given S a set of positive and negative traces, learn an LTL formula ϕ that is consistent with S in the sense that all positive traces satisfy ϕ and all negative traces violate ϕ. Positive traces contain correct system behaviors, whereas negative traces represent unwanted behaviors. First, the tool reduces the mining problem into a series of satisfiability problems in propositional logic to find small LTL formulas. Then, the generated LTL formulas are used as features of a DT procedure to generate more complex specifications.

First, the procedure encodes the specification mining problem into a SAT problem. The goal is to construct the propositional formula Φ_n^S following two properties: (I) Φ_n^S is satisfiable if and only if there exists an LTL formula of size n (i.e., with n subformulas) that classifies the input traces correctly; (II) a model of Φ_n^S contains sufficient information to construct such an LTL formula. The key idea of Φ_n^S is to encode the syntax Directed Acyclic Graph (DAG) of an unknown LTL formula $\varphi^\star$ with n subformulas and then constrain the variables of Φ_n^S such that $\varphi^\star$ is consistent with the input traces in S. A syntax DAG represents the structure of an LTL formula, where each node in the DAG is a subformula or operator, and each edge represents the application of an operator to its arguments. As an example, we report below the syntax DAG of the LTL formula $(p\ U\ Gq) \vee (FG\ q)$.



Propositional variables are used to label nodes and define relationships between them. Formula labeling variables ($x_{i,\lambda}$) are true if node i is labeled with λ, where λ is an atomic proposition or a logical operator ($\neg, \vee, \wedge, X, U, F, G$). Child relationship variables ($l_{i,j}$, $r_{i,j}$) are true if node j is the left or right child of node i, respectively.

To ensure that the syntax DAG is valid, the authors impose several constraints:

- each node must have exactly one label;
- each node (except leaves) must have exactly one left and one right child;
- all leaves are labeled with an atomic proposition.

Finally, the author introduces additional constraints to implement the semantics of the used LTL operators. Once the mining problem is encoded, the approach starts with $n = 1$ and incrementally increases n until the SAT solver finds a satisfying assignment. When Φ_n^S is satisfiable, the satisfying assignment corresponds to an LTL formula consistent with the traces. The most interesting aspect of this approach is that the structure of the generated LTL formulas is decided by the SAT solver, freeing the user from the burden of defining templates.

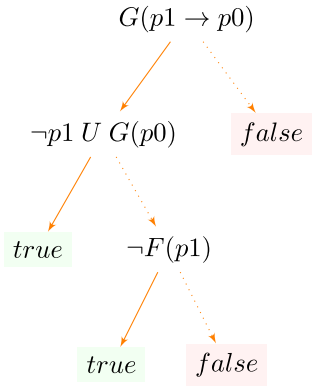
The SAT-based approach is extended with DT learning. In this context, a DT is a structure whose inner nodes are labeled with LTL formulas generated through the SAT approach, and whose leaves are labeled with either true or false. The LTL formula represented by such a tree is given by:

$$\psi_t := \bigvee_{\rho \in P} \bigwedge_{\varphi \in \rho} \varphi$$

where P is the set of all paths from the root to a leaf labeled with true and $\varphi \in \rho$ denotes that φ occurs on ρ (negated if the path follows a dashed edge).

To build the DT, apply a standard DT learning algorithm (e.g., using Gini impurity as a split heuristic). Each internal node of the tree represents a decision based on an LTL primitive. Each leaf node represents a classification (true or false).

An example DT might look like the one below.



This tree represents the following combination of LTL primitives: if $G(p1 \rightarrow p0)$ is false, then the overall formula is false. If $G(p1 \rightarrow p0)$ is true, then check $\neg p1\ U\ G(p0)$: if $\neg p1\ U\ G(p0)$ is true, the overall formula (conjunction of the two formulas) is true. If $\neg p1\ U\ G(p0)$ is false, then check $\neg F(p1)$: if $\neg F(p1)$ is true, the overall formula is true. If $\neg F(p1)$ is false, the overall formula is false. To be clear, the sub-tree on the left (in which the leaves are true) represents the formula

$$G(p_1 \rightarrow p_0) \wedge (\neg p_1 \cup G(p_0)) \\ \vee G(p_1 \rightarrow p_0) \wedge \neg(\neg p_1 \cup G(p_0)) \wedge \neg F(p_1)$$

The work above was extended in [66] and [67], where the authors employ a MaxSAT-based approach to explicitly address the challenge of noise in the data. This method uses a more generalized variant of MaxSAT, called Partial Weighted MaxSAT, which allows for the inclusion of *soft constraints* with weights. Given a propositional formula with weights assigned to clauses, MaxSAT solvers try to find a valuation that satisfies all the hard constraints and maximizes the total weight of the soft constraints that can be satisfied. This makes the approach robust against misclassifications in the dataset, a significant improvement over traditional SAT-based methods, which are not designed to handle noisy data effectively.

Riener [167] improved the foundational work of Neider and Gavran by introducing a topology-guided synthesis approach using partial DAGs. Riener's approach employs DAG topologies to direct the synthesis procedure. Rather than formulating the issue as a single, large SAT instance, this method divides it into several smaller, independent subproblems. Each subproblem corresponds to one possible DAG topology structure that is then solved independently and in parallel.

Other works proposed similar approaches employing SAT solvers to mine LTL specifications that exactly separate positive and negative examples. Camacho and McIlraith [168] have developed an SAT-based approach, utilizing counter-free Alternating Finite Automata (AFA) as a foundation. While AFA and LTL appear different on the surface, the subclass of counter-free AFA is equivalent to LTL in terms of expressive capabilities. This equivalence is carefully examined in their work.

For a more in-depth analysis of SAT-based approaches, we refer to [16].

E. NATURAL LANGUAGE PROCESSING

This section focuses on approaches for generating specifications by parsing and analyzing human language input. This involves using Natural Language Processing (NLP) techniques to understand and interpret text. By applying these techniques, NLP systems can transform unstructured text into structured data that can be used for various applications such as requirements engineering, document summarization, or data extraction. In this section, we report the main variants of this technique to generate LTL specifications.

1) NATURAL LANGUAGE PROCESSING USING LONG SHORT-TERM MEMORY (LSTM)

Patel et al. [76] presented a NLP approach that focuses on LSTM models. The methodology is designed to transform natural language instructions into LTL specifications for controlling robotic systems. It aims to translate movement commands, which start as natural language descriptions, into formal specifications based on a set of ground truth trajectories (accurate reference paths used to guide or train the system). For example, a command like "Walk straight until the intersection, then turn left" would be converted into an LTL expression. To achieve this, the approach utilizes an LSTM encoder-decoder model.

The **encoder** processes the input sentence to create contextualized representations, providing a compact input representation. A contextualized representation refers to how the model encodes the meaning of a word, token, or sequence based on its surrounding context. Imagine you're generating a sentence word by word, starting with "The cat". When predicting the next word, say "runs", the decoder forms a contextualized representation of "runs" by taking into account the context provided by "The" and "cat." This allows it to understand that "runs" in this context refers to the cat moving quickly, as opposed to other meanings of the word, such as operating a machine or managing a business.

The encoder is a bidirectional LSTM, typically used in the machine learning community to analyze data sequences, like text or time series. Unlike regular models that only look at data in one direction (from past to present), a bidirectional LSTM looks at the data forward and backward to help the model understand the context and make more accurate predictions.

The **decoder** generates tokens of the LTL expression sequentially (in practice, creating the complete LTL specification) by analyzing the output of the encoder. It is a feed-forward network with attention mechanisms. In a basic feed-forward network, information flows in one direction, from input to output. When combined with an attention mechanism, the network can identify and concentrate on the most relevant parts of the data. This ability to prioritize important information makes the model more effective at translating text and recognizing speech.

The **encoder-decoder** system learns to translate the natural language specifications into LTL formulas during the training phase: the input trajectories are used as ground truth to supervise the model's training by providing a reward if the model correctly generates tokens (i.e., parts of the LTL specification) that respect the trajectories.

Finally, once the model is trained, it can be used by the robot to learn and map between natural language and LTL specifications.

A more recent approach based on LSTM has been presented by Cherukuri et al. [92]. Here, contrary to the previously described approach, the LSTM encoder-decoder model is used to translate LTL to natural language sentences.

2) CHATBOT-BASED NLP

Keszocze and Harris [80] presented a Chatbot-Based NLP. Their methodology employs Google Dialogflow [169], a chatbot framework, to simplify extracting assertions from natural language hardware specifications. It leverages Dialogflow to train a model that recognizes various natural language expressions of properties and generates corresponding SystemVerilog assertions. The system's core is the **assertion generator**, which communicates with an agent running Dialogflow servers. This agent needs to be trained using a dataset of annotated sentences, which are natural language sentences labeled with the intent (e.g., equality, implication) and the parameters(e.g., signal names) of the corresponding LTL specification that are expected to be mined. The model then can generalize and recognize sentences that are not already known. The assertion generator sends natural language statements to the Dialogflow agent that breaks down each sentence into intents and parameters, which are then used to create the assertions. The different types of intents are as follows. The implication operator $\rightarrow$ is typically recognized by the words "if", "then", and "when". The conjunction operation && is recognized by the words "and" and "also". Equality, represented by ==, is recognized by phrases such as "is", "is the same as", and "must be". Inequality, expressed as ! =, is recognized by phrases like "is not" and "must not be". Stability, corresponding to the SystemVerilog function $stable(), is recognized by the phrase "must be stable". The expression == 1 is recognized by phrases like "must be asserted" and "is asserted". The SVA s_until operator is recognized by the word "until".

Finally, the assertion generator constructs the corresponding SVA based on the identified intent and parameters.

3) LLM-BASED NLP

Aditi et al. [82] utilize fine-tuned Large Language Models (LLMs) to automate the conversion of natural language specifications into validated SVAs. The first step of the methodology is to tune the CodeT5-Large-Python [170] and PLBart [171] models on a custom dataset of natural language specifications and corresponding SVAs. Once fine-tuned, the models generate multiple candidate SVAs from the input specifications. To ensure the accuracy of the generated SVAs, each is translated back to English using another set of fine-tuned models. The back-translated English statement is then compared with the original specification to assess its accuracy. If the two statements are sufficiently similar, the SVA is considered valid. To ensure consistency, the English statements from the back-translation are used to regenerate SVAs. The similarity between the original SVA and the regenerated SVA is computed using cosine similarity [172], measuring how closely the generated SVA matches the intended specification. This ensures that the final assertion is syntactically and semantically aligned with the original specification.

4) NLP THROUGH LIFTED TRANSLATION

Liu et al. [90] presented *lang2LTL*, a methodology to convert natural language navigational commands into LTL specifications. The approach consists of three main steps: referring expression recognition, referring expression grounding and lifted translation, each described hereafter.

- **Referring expression recognition**: The first step is identifying noun phrases and proper names that refer to specific landmarks or propositions. This is achieved by prompting GPT-4 with task descriptions and examples to recognize referring expressions. For example, consider

the command u = "Go to the store on Main Street but only after visiting the bank". This step would identify "the store on Main Street" and "the bank" as landmarks. These substrings are also known as referring expressions.

- **Referring expression grounding**: The second step maps each referring expression to known landmarks. This step is needed because the user could refer to the same landmark using different expressions in natural language. To achieve this, each referring expression previously extracted is associated with a proposition identifier by calculating the similarity of their embedding. The embedding, which is a numerical representation capturing the semantic information of a specific landmark, such as its street name and nearby amenities, is calculated by a LLM. The referring expression "the store on the main street" could, in this way, be associated with the proposition "Walmart".
- **Lifted translation**: Each mapped proposition is also associated with a placeholder string. The referring expression is then substituted in the natural language sentence by this placeholder. The relationship between the original proposition is unique, meaning that after translation, it is always possible to go back to the initial proposition. The lifted translation module uses T5-Base and GPT-3 models, fine-tuned with supervised learning, to generate LTL formulas in prefix format. After the substitution, command u would become "Go to A but only after visiting B" which the translation module will transform into formula $\phi = F(A) \wedge (A \; U \; B)$.

Once the translation has been completed, the placeholders are substituted again with the propositions through the aforementioned mapping, obtaining the final LTL formula representing the robot command.

5) INTERACTIVE LLM-BASED NLP

Cosler et al. [91] presented a methodology to translate unstructured natural language into LTL. The approach leverages LLMs to generate sub-translations, which are refined via "human-in-the-loop" interactions. One of the key components of this approach is the decomposition of the natural language input into "Sub-translations". These are smaller, meaningful fragments of the original sentence mapped to a logic subformula. For example, consider the sentence: "Globally, grant_0 and grant_1 do not hold at the same time until it is allowed", in this case, the phrase "do not hold at the same time" might be mapped to the subformula $\neg(g_0 \wedge g_1)$. The decomposition is automatically performed by the underlying LLM, which generates potential sub-translation starting from the input phrase. The user can correct errors or resolve ambiguities in the provided sub-translations by reviewing, editing, adding, or deleting them. The LLM is guided in the generation by few-shot prompting, where a small number of examples are provided within the prompt to demonstrate the desired pattern. The model generates sub-translations and the final LTL formula based on the prompt. The user can then reiterate the refinement process by modifying the sub-translations as described before until satisfied with the generated formula.

6) NATURAL LANGUAGE PROCESSING THROUGH GRAMMAR

Harris C. and Harris I. introduced GLAST (Grammar Learning for Assertion Translation) in [74]. GLAST is a machine-learning based approach that transforms natural language specifications into hardware assertions using an attribute grammar, a formal grammar that assigns attributes to the production rules of a context-free grammar [173]. These attributes capture the semantic meaning of the grammar's symbols and are critical for translating English sentences into SVAs.

The methodology starts with a learning set consisting of natural language sentences, their corresponding SVAs, and a list of signal names used in the design. For instance, the natural language specification "If rst is high, bvalid should be low" might be paired with its corresponding SVA $always(rst == 1 \rightarrow bvalid == 0)$.

The process begins by creating a token symbol for each unique word in the sample sentences, forming the starting grammar. Attribute values are then assigned to these tokens based on an initial mapping provided by the user, which could include attributes like signal names or specific functions. For example, "RESET" might be mapped to the signal name attribute [RESET].

After the unique words are tokenized and their attributes assigned, each sentence is converted into a top-level production by replacing the words with their corresponding tokens. At this stage, the sentence structure is represented in terms of tokens rather than words. Each top-level production is associated with an attribute that corresponds to the SVA of the original sentence.

Next, the attribute values are used to map the tokens in the SVA. Words or symbols in the SVA that match the attribute values of the tokens are replaced with appropriate attribute references. This allows the system to generalize the relationship between the natural language sentences and the corresponding SVAs, enabling the translation of unseen sentences using the same grammar.

For instance, consider the following initial grammar.

$$P0 \equiv \begin{cases} S0 \rightarrow S1\ S2\ S3\ S4\ S5\ S6\ S2\ S3\ S7 \\ S0.a = [\text{always}(S1.a\ S3.a\ S4.a \\ \qquad\qquad S5.a\ S6.a\ S3.a\ S7.a)] \end{cases}$$

$$P1 \equiv \begin{cases} S1 \rightarrow \text{RESET} \\ S1.a = [\text{RESET}] \end{cases}$$

$$P2 \equiv \begin{cases} S2 \rightarrow \text{should} \\ S2.a = [\,] \end{cases}$$

$$P3 \equiv \begin{cases} S3 \rightarrow \text{be} \\ S3.a = [==] \end{cases}$$

$$P4 \equiv \begin{cases} S4 \rightarrow \text{high} \\ S4.a = [\text{"1"}] \end{cases}$$

$$P5 \equiv \begin{cases} S5 \rightarrow \rightarrow \\ S5.a = [\rightarrow] \end{cases}$$

$$P6 \equiv \begin{cases} S6 \rightarrow \text{BVALID} \\ S6.a = [\text{BVALID}] \end{cases}$$

$$P7 \equiv \begin{cases} S7 \rightarrow \text{low} \\ S7.a = [\text{"0"}] \end{cases}$$

In this example, the top-level production attribute (S0.a) corresponds to the SVA generated from the sentence in the learning set. The grammar assigns specific attributes (e.g., ''=='', ''1'') to words like ''be'' and ''high'', which map to tokens like S3.a and S4.a. The original learning-set specification, ''RESET == 1 → BVALID == 0'', can be generated starting from the top-level production attribute (S0.a) by resolving the attribute references in the rules of the grammar, e.g., substituting S1.a with RESET from production P1.

Once the initial grammar is created, the search space is explored using a beam search technique [174]. During this process, a small set of candidate grammars is retained in a structure called a beam. The grammars are evaluated and pruned based on the Minimum Description Length (MDL) heuristic, which favors more compact grammars that can parse the learning set efficiently.

The process continues through three main stages: merge, chunk, and augment. In the merge stage, words that can be used interchangeably, like signal names, are combined into new non-terminal symbols, generalizing the grammar. In the chunk stage, commonly occurring sequences of words, such as ''should be'', are grouped into new non-terminals, reducing grammar specificity. Finally, in the augmentation stage, new symbols are added to further generalize the grammar.

The algorithm iterates through these stages, evaluating candidate grammars using the MDL heuristic and selects the best grammar based on its ability to compactly represent the learning set. The final grammar can then be used to translate new sentences from the specification into SVAs.

7) NATURAL LANGUAGE PROCESSING USING PARSERS

This technique integrates domain-specific glossaries, advanced parsing techniques, and semantic analysis to translate natural language into structured formal logic. The goal is the generation of accurate specifications by addressing ambiguities, identifying key semantic elements, and using predefined templates. The main idea is to use a parser to transform the natural language specifications into a tree-like structure used to create the final specifications. The basic steps of this technique are reported hereafter.

- **Preprocessing**: the process begins with preprocessing, where natural language sentences are standardized using a domain-specific glossary. This glossary includes commonly used terms and their synonyms to ensure consistency. A predefined list of terms for controlled languages aligns the natural language with the required syntax and semantics for formal specification. Controlled languages are a subset of natural languages obtained by restricting grammar and vocabulary to reduce or eliminate ambiguity and complexity.
- **Parsing**: in the parsing phase, sentences undergo syntactic analysis using tools like the Stanford parser [175]. This step identifies grammatical structures and syntactic dependencies, which are crucial for understanding the logical and temporal relationships within the specifications. Domain-specific optimizations are applied to resolve ambiguities, resulting in an accurate parse tree.
- **Intermediate Representation (IR) generation**: after parsing, the refined parse tree is transformed into a tree-like intermediate representation IR. This IR captures essential semantic information, distinguishing between triggers and effects and unary and binary temporal operators. Semantic parsing is performed in controlled languages using regular expressions or dependency graphs to create a structured representation.
- **Specification generation**: the IR is then used to generate the formal specification. Simple clauses in the IR are matched to predefined templates to create propositions. These propositions are combined using logical and temporal operators to form a complete specification. Logical fragments are synthesized into comprehensive logical forms, ensuring that all aspects of the natural language specification are accurately represented. The IR or logical forms are subsequently translated into formal specifications, such as LTL formulas or SVAs, using predefined syntax and templates.

Different variations exist to accommodate specific use cases; in [77], the technique is integrated with a glossary specific to smart home applications, allowing the methodology to achieve higher precision in synthesizing properties for this context. In [83], after the LTL formulas are generated from the IR, they are checked using a model-checking approach. This involves translating the formulas into Büchi automata and checking the emptiness of their intersection to ensure no contradictions exist among the requirements. In [86], the authors built a controlled word list, which is a collection of words that correspond to SVA syntax keywords in order to facilitate the translation into SVAs. In [88], the authors explore the problem of synthesizing LTL formulas to control the movement of robots. The parser-based approach is extended to provide feedback to the user. In fact, if the LTL specifications are not synthesizable (i.e., the robot cannot create a valid plan to execute the task with the provided constraints), the system uses SAT-solving to identify the minimum unsynthesizable cores, which represent specific

parts of the specification causing a failure. Once the critical parts are identified, the causes of the failure are reported to the user.

F. COMPOSITION

Composition is one of the most popular techniques across all reviewed papers. However, that is not surprising, considering that most specification mining algorithms naturally include compositional aspects, for example, to merge the states of an automaton or to build more complex specifications from simpler formulas. In this section, we report a relevant subset of approaches where the compositional aspect is the main or one of the main themes in the respective papers.

1) MODEL FISSION AND FUSION

Le et al. [47] propose a tool called SpecForge to combine the results of multiple automata specification miners through a technique called ''model fission and fusion''. The approach involves three main steps: model construction, fission, and fusion.

In the model construction phase, the process generates several FSAs using existing specification mining tools from the execution traces of the same DUV. In particular, SpecForge employs seven different specification miners: 1-tails and 2-tails, CONTRACTOR++, SEKT 1-tails, SEKT 2-tails, and TEMI (optimistic and pessimistic).

After that, the fission step breaks down each FSA into smaller LTL formulas. In particular, SpecForge employs 6 different binary LTL templates (e.g., $G(P0 \rightarrow XFP1)$) and checks all couples of trace events (using the SPIN model checker [176]) to determine which properties are satisfied by the FSAs. Then, the tool must single out the most relevant LTL constraints from the large set generated. To achieve that, SpecForge uses a heuristic that selects a constraint if it is satisfied by at least N FSAs: if $N = 1$, then the approach considers all constraints (Union), and if N is equal to the number of FSAs, then only the constraints satisfied by all FSAs are considered (Intersection).

Finally, in the fusion phase, the selected constraints are fused back together to create a new, more accurate FSA. In particular, the selected constraints are translated back into simple FSAs, which are then combined using automaton intersection to form a unified FSA.

2) NAIVE ITERATIVE COMPOSITION OF BASIC PROPERTIES

The paper in [48] presents a simple tool called Dianosis to iteratively generate complex specifications from HW traces. First, the tool generates basic properties by testing all combinations of DUV variables on the input traces using the well-known Open Verification Library (OVL) [177] assertion templates, such as the ''increment checker'' (analyzes whether a signal behaves as an increment counter) and the ''one-hot checker'' (identifies if a bus has only one bit set to ''1'' at any given time). Only candidates that hold true on the entire input trace are used in the subsequent step.

In the second phase, the methodology attempts to combine the validated basic properties to form more complex ones. These combinations consider temporal dependencies and other logical relationships between the basic properties. Starting with a set of validated basic properties $P = \{p_1, p_2, \ldots, p_k\}$ from the previous step, generate a set of complex properties C starting from step 1.

1) For each pair of basic properties p_i and p_j in P (where $i \neq j$): create a new property candidate c_{ij} by applying temporal dependency operators (e.g., conjunction $\wedge$, disjunction $\vee$, mutual exclusion, sequential order) to combine p_i and p_j. If the new candidate holds true on the input traces, add it to the set of complex properties: $C = C \cup \{c_{ij}\}$.
2) If new properties were added, then update the set of properties to be combined $P = C$ and go back to step 1. Otherwise, if no new properties were added, terminate the process as no further valid property combinations can be generated.

3) COMPOSITION OF MICROPATTERNS USING COMPOSITION RULES

Gabel and Su [42] propose a tool called Javert to compose micropatterns in the form of FSAs using logical inference rules. First, the tool generates a set of binary micropatterns using Binary Decision Diagrams (BDDs) (more on this in section VI-G) following two templates: alternating sequences (e.g., $(ab)*$) and resource usage patterns (e.g., $(ab * c)*$). After identifying the micropatterns, the next step is to combine them into more complex patterns. The process iteratively applies four composition rules: expansion, intersection, branching, and sequencing (the latter two are called ''inference rules'' in the paper). The entire process repeats in a loop (expansion, intersection, and inference rule application) until the set of patterns stabilizes, and no further compositions can be derived. The employed rules are summarized below.

- The *expansion rule* widens the detected micropatterns to include additional events that may occur in the trace but were not part of the original micropattern's alphabet: each micropattern's recognized language is expanded to account for additional symbols that may interleave between the events of the micropattern. For example, suppose we have a basic pattern recognized by the automaton A for the sequence $(open\ close)*$ with alphabet $\Sigma = \{\text{open, close}\}$. If the larger alphabet $\Sigma' = \{\text{open, close, log, connect}\}$, the expansion $E_{\Sigma'}(L(A))$ would allow interleaving ''log'' and ''connect'' events between the ''open'' and ''close'' operations. The expanded language might now include sequences like: $E_{\Sigma'}(L(A)) = \{\text{open log close, open connect close, open log connect close}\}$ while the core ''open close'' pattern is preserved.
- The *intersection rule* creates a new automaton by computing the intersection of their expanded languages. This

intersection corresponds to sequences in which both patterns are satisfied simultaneously. The intersection automaton is constructed by taking the product of the expanded automata. For example, consider two patterns, $(open\ read * close)*$, and $(open\ (write \mid read) * close)*$. The intersection of these patterns over the same trace could yield a combined pattern where both sequences must occur; the resulting pattern might be something like $(open\ read * close)*$.

- The *branching rule* combines patterns that share the same starting and ending states but differ in the internal sequence of events. If two patterns $(aL_1^*b)^*$ and $(aL_2^*b)^*$ are found, they can be combined into $(a(L_1 \mid L_2)^*b)^*$. For example, consider the patterns $(open\ close)*$ and $(open\ seek * close)*$. The branching rule allows these to be combined into a pattern representing either possibility: $(open\ (close \mid seek * close))*$. This captures the idea that after opening, you can either directly close or perform a seek before closing.
- The *sequencing rule* combines patterns that can be sequenced, where the end state of one pattern matches the start state of another. For patterns $(aL_1b)*$ and $(bL_2c)*$, the resulting composition is $(aL_1bL_2c)*$. For example, consider the following patterns, $(connect\ open)*$, and another where "open" is followed by "close" $(open\ close)*$. By applying the sequencing rule, these patterns can be combined into $(connect\ open\ close)*$, capturing the sequence where "connect", "open", and "close" occur in that order.

4) PATTERN CHAINING

Pattern chaining is a technique that sequentially combines simpler inferred patterns into more complex sequences. Imagine you're observing a sequence of events in a program's execution, like a series of method calls. Each method call might have a certain pattern, such as "whenever event A happens, event B must follow". Pattern chaining looks for such simple relationships and tries to link them together into a larger chain, creating a more comprehensive description of the program's behavior. In the context of the Perracotta tool [44], the process begins with the inference of simple two-event temporal properties using a pre-defined set of templates (see section VI-B). Once these simple properties are identified, the chaining process logically combines related two-event properties into a longer sequence or chain. Specifically, if the inference process identifies that event A must be followed by event B ($A \Rightarrow B$) and event B must be followed by event C ($B \Rightarrow C$), the chaining technique would combine these to form a longer chain $A \Rightarrow B \Rightarrow C$. This chaining not only reduces the number of properties that need to be considered but also provides a more complex and comprehensive understanding of the program's behavior.

5) COMPOSITION OF INVARIANTS

Several works employed the Daikon tool (or other approaches to generate invariant, see section VI-B) to generate a set of non-temporal specifications and use them to build more complex temporal or non-temporal specifications. Bertasi et al. [53] propose an approach that mines specifications of the from $G(antecedent \rightarrow consequent)$ by composing invariants. First, the process generates a set of potential antecedents by identifying initial conditions from the execution traces that could potentially lead to interesting properties. Antecedent collection involves analyzing subsequences of the execution trace, looking for invariants that hold true in one subsequence but not in the next. This step identifies points where significant changes occur in the system, marking them as potential antecedents. For example, if at one point in the execution trace, a variable x consistently equals 1, and this condition no longer holds in the subsequent part of the trace, the change (from $x = 1$ to $x \neq 1$) is flagged as a potential antecedent. After that, the process mines the consequent, specifying what conditions are expected to follow the mined antecedents. The paper defines three temporal patterns to mine the consequent: next, until, and alternating. The approach employs a series of algorithms to identify invariants in meaningful subtraces that can be used to fill the placeholders of the aforementioned templates. For each antecedent, the execution trace is examined to see if it matches any of the temporal patterns with a possible consequent. If a matching consequent is found, a property is generated.

For example, for the "next" template, consider the case in which we need to find a consequent for the partially instantiated template $G(x = 0 \rightarrow XP0)$, where $P0$ needs to be filled with invariants. First, we must identify the subtrace S' of the input execution trace, where the antecedent candidate $x = 0$ is consistently true. The miner then searches for a consequent that holds at each simulation instant immediately following the instants in S'. If, during the mining process, it is found that whenever $x = 0$ in the current instant, $x \geq y$ holds in the next instant, the property $G(x = 0 \rightarrow X(x \geq y))$ is generated. A similar approach generates specifications following the "until" and "alternating" templates.

G. OTHER TECHNIQUES

In this section, we discuss other notable mining techniques that do not fit neatly in the aforementioned categories and are underrepresented to warrant their own category.

1) MINING LTL SPECIFICATIONS WITH GRAPH NEURAL NETWORKS

Luo et al. [130] proposed a novel method for mining LTL specifications using Graph Neural Networks (GNNs), leveraging their natural ability to handle structured data in the form of graphs. These networks operate by leveraging a message-passing mechanism where each node aggregates information from its neighbors through successive layers, allowing the network to learn complex patterns.

The approach begins by transforming sequences into linear graphs. In this graph-based representation, each node corresponds to an individual token from the sequence. Each

token represents a discrete event or data point within the sequence, such as a specific action or system state. Edges in the graph denote the sequential progression from one token to the next, establishing a directional flow that mirrors the temporal order of the sequence. Each node in the graph is associated with a feature vector. This vector encodes both the logical properties and temporal relations pertinent to each token, enabling the GNN to effectively analyze and learn from the structure and dynamics of the sequence. The GNN is trained on a dataset labeled as $S = (P, N)$, where P consists of positive examples and N consists of negative examples. These examples are sequences whose tokens collectively satisfy or violate specific specifications, demonstrating desired or undesired behaviors, respectively. Through training, the GNN learns to classify new sequences by predicting whether their overall arrangement of tokens satisfies the given specifications, leveraging the encoded information within the nodes' feature vectors.

While Luo et al.'s method provides a novel way to apply GNNs to the problem of LTL specification mining, it faces significant limitations. The primary limitation is that the formulas derived from the model do not always accurately capture the behavior learned by the GNN, due to a disconnect between the neural network's pattern recognition capabilities and the semantics of LTL formulas.

To address these limitations, Wan et al. [131] introduced an enhanced GNN architecture that incorporates "faithful LTL encoding". This method involves placing parametric constraints on the network weights to ensure that the operations performed by the GNN can be directly and reliably interpreted as operations on LTL formulas. The faithful encoding guarantees that each component of the GNN's computation corresponds to a logical component of a formula, thus ensuring that the neural network's output can be translated back into an LTL formula without losing meaning or accuracy. This enhancement not only improved the faithfulness of the generated specifications but also backed the theoretical grounding of the approach, making it more robust against the inherent uncertainties in training neural networks.

2) BAYESIAN INFERENCE

In this section, we report methods from two different research teams to learn specifications from traces using a probabilistic technique called Bayesian inference.

The first group proposed three works to learn LTL specifications from task demonstrations (execution traces) using predefined LTL templates and Bayesian inference. In their first work [105], the authors describe the fundamentals of their methodology. The method can be divided into two main steps: 1) generating potential candidate specifications using LTL templates (mainly "and" combinations of G and F operators) and Bernoulli trials and 2) evaluating the candidates using the Bayes theorem to determine the most likely specifications. First, the process defines probabilities for each proposition; for example, for each task (e.g., "The robot must never touch the centerpiece") or subtask (e.g., "Place the dinner plate"), a probability of 50% and 70% could be defined, respectively. This probability represents the likelihood that a proposition will be included in the candidate specification in a Bernoulli trial. These probabilities are not derived from the data but are set arbitrarily or based on domain-specific knowledge. After that, the process performs a trial for all combinations of propositions/templates. All combinations passing the trial are considered specification candidates. The next step is to evaluate how well each candidate explains the observed task demonstrations using Bayesian inference. Each candidate specification has an initial prior probability $P(\varphi)$, which reflects its likelihood before considering the observed data. The prior probability can simply be the same probability used in the Bernoulli trials or can be modified by the user according to the structure of the tasks. The likelihood $P(D|\varphi)$ is the probability that the observed demonstrations D would occur given the candidate specification φ. This is computed by evaluating how well the demonstrations satisfy the logical and temporal constraints imposed by the candidate specification: if the demonstrations consistently adhere to these constraints (e.g., respecting the required order of actions and always meeting certain conditions), the likelihood is high; if they violate the constraints, the likelihood decreases. Using Bayes' theorem, the prior and likelihood are combined to compute the posterior probability $P(\varphi|D)$ of each candidate specification:

$$P(\varphi|D) = \frac{P(\varphi)P(D|\varphi)}{\sum_{\varphi' \in \Phi} P(\varphi')P(D|\varphi')}$$

where Φ is the hypothesis space or the set of all possible candidate specifications φ' that could be considered. The posterior probability reflects how likely a candidate specification is to be the correct one, given the observed data.

The authors built on this work in [52] to address the need for generating contrastive explanations, where the goal is to identify LTL specifications that are satisfied by one set of traces but not by another. This allows for a clearer understanding of what distinguishes different behaviors; the approach is useful for anomaly detection, system comparison, and user-group classification. Finally, the author extended the Bayesian inference approach again in [129] by incorporating supervision into the specification learning process: the innovation here is to refine the inference process by utilizing expert knowledge or additional training data, which is often available in practical applications.

The second research group proposed a different method [121] and a tool called BaySpec [178] that involves the use of Bayesian Networks (BNs); here, the focus is on mining specifications from imperfect traces to address the presence of noise in the data. A BN is a type of graphical model that represents a set of variables and their conditional dependencies using a DAG. It is like a roadmap that shows how different factors influence each other and allows you to calculate the probability of certain outcomes given prior information. Given a set of traces, the methodology segments

these traces into subtraces, each of which is associated with a specific functional process. This segmentation ensures that each subtrace corresponds to a distinct function, which is necessary because the specifications are meant to describe individual functions, not composite behaviors. For example, if a trace contains data for both a braking system and an indicator light system in a vehicle, the functional segmentation step would isolate the events related to braking into one subtrace and the events related to the indicator lights into another. A BN is trained for each segmented subtrace. The nodes in a BN represent events, and the edges represent the conditional dependencies between these events. The BN aggregates temporal and causal relationships observed in the data, allowing for a compact and probabilistically consistent representation of the system's behavior. For example, in the indicator light system, the BM might represent the sequence of events where a driver's action of turning on the indicator (event A) probabilistically leads to the light being activated (event B) with a certain probability. After that, the most likely sequences of events (paths) are identified within each network. These paths represent the most common behaviors observed in the traces and are potential candidates for specifications. In particular, the BN is converted into a Mining Graph (MG), a directed acyclic graph where each node represents a possible event state, and edges are weighted by the likelihood of transitioning from one event to another. The task is to find paths through this graph that maximize the average likelihood, indicating the most probable sequences of events. If the identified paths appear too strict (i.e., they might specify behaviors that are too specific and rare), they are loosened by merging similar ones, making the resulting specifications more general. Finally, the final paths are converted into LTL specifications.

3) BINARY DECISION DIAGRAMS

The work in [123] introduces a new algorithm based on BDDs to efficiently generate complex sequential patterns similar to those mined by Perracotta. The approach exploits BDDs to mitigate some of the scalability issues, efficiently handling patterns involving more than two components, such as $(ab + c)*$. The approach's core involves encoding the state space of possible assignments in a program trace using BDDs. A BDD symbolically represents the states of an automaton (the pattern being mined) as it processes a program trace, where each possible mapping from trace symbols to the automaton's alphabet is encoded as a Boolean function. In this context, a BDD represents the current state of all possible assignments of trace symbols to the automaton's alphabet, allowing the algorithm to efficiently update and track these states as it processes each symbol in the trace.

Consider the following example. If the input trace contains the symbols w, x, y, z and we want to mine patterns of the form $(ab)*$, the BDD will efficiently check whether any pair of symbols from the set $\{w, x, y, z\}$ forms a sequence that fits the pattern $(ab)*$. First, you can create pairs from the set of symbols, such as (wx), (xw), (wy), (yw), (wz), (zw), (xy), (yx), (xz), (zx), (yz), (zy). Each pair represents a possible mapping of the pattern $(ab)*$ to the symbols in the trace: the BDD will symbolically represent the state transitions of the automaton of each couple. The BDD does not explicitly check each pair one by one in a brute-force manner. Instead, it symbolically encodes the possible transitions and checks them in a compact and efficient way. As it processes the entire trace, the BDD updates the automaton's state whenever it encounters a symbol, verifying if any sequence of transitions in the trace satisfies the $(ab)*$ pattern for any of the symbol pairs. For example, if the trace is $wxyzwyxz$, the BDD would check pairs like $(wx)*$ to see if the trace contains sequences like $wxwxwx$, and $(xz)^*$, to see if the trace contains sequences like $xzxzxz$, among others. Once the BDD has processed the entire trace, the final BDD represents all possible combinations of the input symbols that follow the considered pattern (such as $(ab)*$) and either accept or reject the input trace. The specifications are then generated by identifying these combinations that lead to acceptance and mapping them back to the corresponding patterns.

4) LTL MINING THROUGH ANSWER SET PROGRAMMING

Ielo et al. [132] proposed a new approach to the passive learning of LTL formulae using Answer Set Programming (ASP) to efficiently solve hard combinatorial search problems. In particular, they propose a passive learning approach to infer formulas that correctly classify a given set of positive and negative traces, those that should and should not satisfy the formula, respectively. ASP is a declarative programming paradigm specifically designed to express complex problems and find solutions by defining rules that describe the problem. The solutions, or ''answer sets,'' are found by computing stable models (sets of interpretations that satisfy these rules), making it particularly effective for tasks involving logical reasoning and constraints. Ther methodology involves three main steps reported below.

1) **Problem encoding**. The first step involves encoding the passive learning problem within the framework of ASP. This includes the representation of both the LTL formulae and the system execution traces as logical entities within ASP. This is comparable with the SAT-based encoding proposed by Neider et al. [60].
2) **ILASP utilization**. Utilizing the ILASP system [179], to generate hypotheses about the LTL formulae, which are then tested against the execution traces. This process iteratively refines the hypotheses based on which traces they classify correctly or incorrectly.
3) **Optimization through example-based pruning**. Unlike traditional methods that evaluate hypotheses one trace at a time, Ielo et al.'s approach considers the full set of traces simultaneously. This collective consideration allows the system to prune larger portions of the search space at once, significantly enhancing the efficiency of the learning process.

5) MAXIMUM A POSTERIORI INFERENCE

The paper in [125] addresses the challenge of inferring task specifications from demonstrations (a trace of observed state-actions) provided by an agent operating in an uncertain and stochastic environment. This problem is particularly relevant in robotics and similar fields where tasks often decompose into sub-tasks with complex temporal dependencies. The authors focus on inferring past-time LTL specifications, which are logical properties that describe the desired behavior of the system based on past and present states and actions. The problem of learning task specifications is framed as a formulation of a Maximum a Posteriori (MAP) inference problem. This involves finding the most likely specifications (that maximize the posterior probability; the likelihood of demonstrations is modeled using a maximum entropy distribution) that explain the observed demonstrations within a probabilistic model: the system is modeled as a probabilistic automaton (essentially a Markov decision process without rewards), where the task is to infer a specification that best fits the observed traces.

6) SYMBOLIC SIMULATION

This section covers two works whose main theme revolves around generating assertions through symbolic simulation [180] techniques. The work in [51] introduces a tool called DOVE designed to identify security vulnerabilities in firmware formalized through assertions that highlight unlikely execution paths. The tool works in three main steps. First, symbolic simulation is applied to explore all potential execution paths of the firmware by considering symbolic values rather than concrete inputs. The simulation generates a symbolic tree, which traces the values of registers and memory locations as the firmware executes. Each node in the tree corresponds to a symbolic state, and edges represent transitions based on conditional statements in the firmware. For instance, if the firmware contains a condition like $a < 40$, the symbolic simulation will generate two branches in the symbolic tree: one where $a < 40$ is true and another where the condition is false. After that, the process assigns probabilities to each execution path in the symbolic tree based on how likely it is for the path to be taken during execution. Finally, the tool generates LTL assertions that describe the behavior of the firmware along the most unlikely paths, which are often the ones where vulnerabilities may hide.

The same authors build on DOVE in [127] by proposing a new tool named COME that automatically generates assertions to detect the sequences of instructions leading to vulnerabilities in firmware. The new approach takes as input a "vulnerability" as a non-temporal assertion, and then it leverages symbolic simulation to generate a set of bad or good execution paths of the firmware. Bad paths satisfy the vulnerability assertion, while good paths do not. After that, the generated symbolic tree is abstracted with a set of identifiers. This abstraction is done by assigning a unique identifier to semantically equivalent operations, converting the paths to sequences of identifiers. Finally, a pairwise alignment algorithm compares bad sequences with good sequences. The goal is to filter out the unnecessary operations in the bad sequences to find the minimal set of operations that are required to trigger the vulnerability. These minimal sequences are then translated back into the original firmware operations to generate temporal assertions that specify the exact conditions under which the vulnerability occurs.

7) SPECIFICATION MINING USING DEPENDENCY GRAPHS

This section focuses on two works that exploit Dependency Graphs (DGs) to generate specifications. Several papers (some of which are included in this review) exploit dependency graphs in one form or another; however, only a small fraction employed DGs as a main theme. The work in [126] exploits software DGs to generate malicious specifications. In this context, a DG is an acyclic graph where nodes are labeled with system calls observed in the execution traces, and the edges are labeled with logical formulas representing the dependencies between the system calls. The mined specification is a refined DG that represents the malicious behaviors. The approach takes as inputs a benign and a malware program. Then, it executes both the malware and benign programs in a controlled environment to capture their behaviors. This involves monitoring the system calls made during the execution of each program. After that, the process converts the collected execution traces into dependence graphs. Finally, the approach identifies the malicious specification by computing the differences between the dependence graph of the malware and those of the benign programs. The resulting specification is a minimal subgraph that captures only the behaviors unique to the malware.

The second work [124] presents a methodology for automatically generating SVAs. The main contribution of this paper is a technique that uses Dynamic Dependency Graphs (DDGs) to infer properties about hardware designs from a small number of simulation runs.

The DDG is created by simulating the design with a given set of test cases. During simulation, the design is instrumented to capture dependencies between signals at different time points. Vertices in the DDG represent expressions, statements, or signal values, while edges represent dependencies, such as data dependencies and control dependencies. For example, in a simple flip-flop design, the DDG might capture how the output *dataOut* depends on the input *dataIn* and the enable signal over multiple clock cycles. After that, the approach generates an initial set of basic properties by annotating vertices in the DDG that correspond to outputs. Each output vertex is traced back through the DDG to gather path conditions and create symbolic expressions that relate inputs to outputs. Similar properties are grouped together to generalize the design's behavior over different scenarios, reducing redundancy and capturing common patterns. However, if combining properties reduces their accuracy, they are split into smaller, more precise

properties. Finally, the properties are formally verified using a model checker. If the model checker finds that an abstracted property holds, it is retained; otherwise, the property is discarded or further refined.

8) TAINT AND TAG

The authors of [128] generate security SVAs by applying taint and tag analysis, which is a technique to identify paths within an RTL design where secure information might leak to non-secure outputs. This method is particularly effective in detecting vulnerabilities related to information leakage, where sensitive data could be inadvertently exposed due to flaws in the design. The first step of the approach involves modifying the RTL model to include additional signals (tags) that track the flow of sensitive data. These tags are propagated through the model alongside the original data. The tags are designed to follow the data flow, including control structures like if-else statements, ensuring that any flow of secure information is tracked comprehensively. As the RTL design is simulated, the taint analysis tracks the flow of tagged data. The propagation of taints is done both explicitly (through data dependencies) and implicitly (through control dependencies). The tags are updated at each step of the simulation to reflect the flow of data through the design. The tagged design is then used to generate security assertions. These assertions ensure that tagged (secure) data does not propagate to non-secure (public) outputs. The assertions are formalized as properties that must hold true during the RTL simulation. If any of these properties fail, it indicates a potential information leakage vulnerability.

The Taint and Tag approach is part of a broader topic called Information Flow Tracking (IFT). While this work is focused and tailored towards detecting specific types of vulnerabilities (where specific security properties are known and need to be enforced), the authors of [71] proposed a tool called Isadora which exploits general IFT. It is a more comprehensive and automated tool that aims to generate a wide range of security properties for hardware designs without requiring detailed prior knowledge from the user. It is better suited for broader security validation tasks, particularly in complex SoC designs, where the range of potential vulnerabilities is vast.

9) MINING PARTIAL ORDERS

The paper in [122] introduces a novel framework for mining API usage patterns directly from source code, focusing on extracting partial orders rather than simple sequential patterns. A partial order is a set of elements where not every pair of elements is necessarily comparable, but the order that is present follows certain rules. In the context of API calls, a partial order represents a set of APIs with some specified order constraints but does not require a strict linear sequence of all calls. The framework is designed to assist in API reuse and to check the correctness of API usage by identifying important orderings among API calls. The framework adapts a model checker to generate static traces of API usage from source code. Then, it develops an algorithm to separate different usage scenarios from the generated static traces. Finally, it mines frequent partial orders from the extracted scenarios. The mining process involves extracting Frequently Closed Partial Orders (FCPOs) from the scenario database. An FCPO is a partial order that is frequent across the different traces and closed, meaning it cannot be extended without reducing its frequency. The mining process systematically searches for these partial orders using a depth-first search through a set enumeration tree, pruning less frequent or redundant orders.

10) SPECIFICATION SKETCHING

The work in [94] proposes a new technique that addresses the challenge of creating formal specifications for system verification. The authors introduce a novel approach called specification sketching, where engineers can provide partial LTL formulas (sketches) that are completed automatically using examples of desired and undesired system behaviors. The sketch completion problem is encoded as a propositional logic problem, and then SAT solvers are used to find consistent substitutions to complete these sketches into full LTL formulas.

VII. QUALIFICATION

Qualifying the specifications is one of the most crucial aspects of the mining process, as miners tend to produce many valid specifications. Although most approaches only return specifications compliant with the input (e.g., the mined property holds on the input trace), not all specifications might be equally interesting to the user. Therefore, developing effective methods to navigate the mined specifications, such as filtering and ranking, becomes fundamental.

In this section, we focus on two main aspects: I) the qualification of the mined specifications II) and the empirical or theoretical evaluation proposed by the authors to validate the effectiveness of their methodology. The difference between the two is that (I) focuses on filtering and ranking to reduce the number of mined specifications, with the goal of keeping only those meaningful for the user, while (II) aims to assess the effectiveness of a proposed methodology, comparing it to similar approaches or employing other evaluation metrics.

A. FILTERING AND RANKING OF SPECIFICATIONS

In this section, we report several approaches to filter and rank specifications.

1) METRIC-BASED QUALIFICATION

The most popular qualification approach involves associating each specification with a score. The score is calculated according to metrics that capture certain characteristics of the mined specifications. Metrics are often used to perform both ranking and filtering. The ranking process orders the set of specifications based on the score, where the highest-scoring specifications are the most ''interesting''.

Filtering is usually associated with a threshold that allows discarding the uninteresting properties: those with a score lower than the specified threshold.

In the reviewed papers, several qualification metrics have been proposed, whereas specification mining approaches traditionally have relied heavily on two key measures: support and confidence. In [181], the authors recognize that these metrics often produce a high number of false positives and may not be the most effective in distinguishing correct specifications from incorrect ones. The authors test 38 additional different metrics to determine their utility. They show that certain measures statistically outperform the traditional metrics and could be more effective in reducing false positives in specification mining. They also highlight the need to look beyond conventional measures and consider alternative interestingness measures for more accurate specification mining.

Below, we report the metrics used in the papers selected for this review.

The metric most encountered to qualify specifications in this review is *support*, found in 18% of the accepted papers, commonly found in data mining and association rule learning. This metric captures the relative frequency of an itemset in a dataset; in other words, if the support is high, there are a lot of instances in which the itemset appears in the trace. The support metric is used in more complex metrics such as *confidence*. This is usually employed to assess the quality of specifications of the form $X \implies Y$. Formally, it is the estimate of the conditional probability $P(Y|X)$; if the rule has 100% confidence, it means that there is a complete coincidence between X and Y: X always implies Y on the input data. This metric is used in 13% of the reviewed papers.

The *precision* and *recall* metrics, employed in [41], are often used to evaluate the quality of inferred FSAs. They are computed by comparing the language accepted by a mined FSA and the language accepted by a ground truth FSA. *Precision* refers to the proportion of sentences accepted by the inferred model that the ground truth model also accepts. *Recall* refers to the proportion of sentences that are accepted by the ground truth model that the inferred model also accepts. The two metrics are combined in [47] to form the *F-Measure*. It is defined as the harmonic mean of precision and recall: $F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$. The use of the F-Measure is justified by the inherent trade-off between precision and recall. For instance, a less detailed FSA that accepts a broader language will have higher recall, but this comes at the expense of lower precision. Therefore, the F-Measure serves as a balanced metric, summarizing whether an increase in precision compensates for a decrease in recall or vice versa.

The *complexity* metric is used to rank and filter the specifications based on their structure. The metric yields a score based on parameters such as the temporal depth and the number of variables used in an LTL specification, or the number of states/edges in an FSA. In [62], the complexity is calculated using the depth of the design captured by the mined assertions, where each depth level represents a step of logic that the computation has to traverse during the execution of the design. To clarify the concept, consider the Verilog code in Listing 2.

```
1  always @(posedge clk) begin
2      if (enable) begin
3          temp <= data_in;
4      end
5  end
6
7  always @(posedge clk) begin
8      if (temp > threshold) begin
9          valid <= 1;
10     end
11 end
```

LISTING 2: Example of Complexity.

An assertion with a complexity equal to 3 could be

$$(enable \;\&\&\; data\_in > threshold) \;|->\; \#\#1\; valid == 1$$

At depth 1, the assertion checks that when *enable* is high and *data_in* is greater than *threshold*. At depth 2, if those conditions are satisfied, *temp* will be set to *data_in* (during the following clock cycle). Finally, at depth 3, *valid* will be set to 1 if *temp* is greater than *threshold*. Each of these steps represents a distinct level of logic; the assertion effectively checks a chain of three conditions that unfold over consecutive clock cycles.

In [21], the authors present a mutation-based metric called *fault coverage* that allows filtering assertions based on their ability to detect faults (mutants), thereby ensuring that the final set of assertions provides high coverage of potential faulty behaviors introduced in new versions of the DUV. To calculate that metric, a set of faults is injected into the DUV. Then, each mined assertion is evaluated depending on the faults it covers. An assertion covers a fault if it holds in the original DUV and fails in the mutated DUV (with the injected faults). The coverage of an assertion is measured as the number of faults detected in the mutated DUV. The goal is usually to identify the minimum set of assertions that cover all the faults. This metric aims at preserving assertions that enhance fault detection, thereby improving the effectiveness of the verification process. Additionally, if the mined assertions are used to synthesize and deploy runtime checkers, the objective is often to minimize the number of checkers to prevent the runtime verification process from saturating the available computational resources [182], and to reduce energy consumption [183] in low-power settings.

Ghasempouri and Pravadelli showed in [184] that the qualification process can be optimized using metrics from the data mining domain using support and correlation coefficient. The correlation coefficient measures the strength of the relationship between the antecedent and consequent of an assertion of the form $G(antecedent \rightarrow consequent)$, indicating how closely their occurrences are related in simulation traces. The authors claim that their approach is

significantly faster than mutation-based techniques while maintaining similar accuracy.

In [61], the authors introduce a novel method to *combine multiple arbitrary ranking metrics*. In this approach, mined specifications are assigned a ranking score based on user-defined metrics. These metrics are arithmetic expressions that can be constructed by the user using various built-in elementary parameters such as support, confidence, complexity, etc. The user-defined metrics can then be used either for filtering or sorting.

Sorting metrics are used to rank assertions based on an overall score, which is calculated using the formula:

$$\prod_{i=1}^{n} calibrate\left(\frac{sm_i(a)}{sm_i(a_{max\_i})}\right),$$

where $sm_i(a)$ is the score of assertion a based on the i-th sorting metric, a_{max_i} is the assertion with the maximum score for metric sm_i, and *calibrate* is a function that adjusts the score according to the formula:

$$R = \frac{1}{(1 + e^{(z-kx)})^2}.$$

This approach enables the use of multiple sorting metrics in a single ranking process. Assertions that score high across all metrics are rewarded, while those performing well in only a few or none of the metrics are penalized. The calibration function plays a crucial role in ensuring that low scores heavily impact the final ranking while preventing high scores in some metrics from compensating for poor performance in others.

2) OTHER QUALIFICATION TECHNIQUES

In this section, we report other techniques not associated with metrics.

- **Model checking**. When mining specifications from a set of partial traces, the extracted formulas are not guaranteed to hold in the DUV. In this case, formal methods, such as model checking, allow the filtering of the formulas that do not hold onto the design [63]. The specifications are checked for correctness by systematically exploring all states of a model representing the system from which the specifications have been extracted. This approach, however, usually presents scalability issues.
- **Reachability**. As described in [44], reducing the set of mined specifications is possible using reachability analysis. The objective is to discard formulas that contain variables with obvious relation among them. The reachability analysis for a property $P \implies Q$ involves analyzing the call graph, a structure that maps a software program's calling relationships between functions or methods, to check whether Q is reachable from P; if so, the property is uninteresting.
- **Name similarity** [44]. This heuristic provides a score based on the similarity of the elements composing the specification. This means that if a specification is composed of events with a similar name, the metric score will be higher. This metric is justified by the fact that developers tend to choose similar names for related functions while coding to keep the code more readable and easier to understand and maintain.
- **Subsumption** [69]. In logic, a formula or a statement subsumes a second if the truth value of the first guarantees the truth of the second. This is particularly useful for filtering redundant specifications to produce a smaller, meaningful set of mined specifications. For example, consider the properties $topOfStack <= theArray.length - 1$ and $topOfStack < theArray.length$ and assume that both are true. It is clear that if the first one is true, the second is also true. This means that keeping the first property makes the second redundant.
- **Null hypothesis test**. As with any dynamic analysis or machine learning technique, there is a possibility of overfitting by reporting properties that are true on the training runs but not in general. For example, if there is an inadequate number of observations of a particular variable, patterns observed over it may be mere coincidence. This problem can be mitigated by computing the *null hypothesis test* [69], that is, the probability that a property would appear by chance with a random input. A mined specification is not discarded if its probability is smaller than a user-defined confidence parameter.

B. EVALUATION STRATEGIES

This section describes how authors demonstrate the effectiveness of their methodology.

- **Comparison**. The most popular approach to show the effectiveness of a new methodology is to compare the quality of the generated specifications against the quality of specifications generated with other existing approaches. In this review, 29% of the papers applied this approach.
- **Time and Memory**. A new methodology is evaluated according to the performance of the proposed tool or algorithm. The most popular performance metrics are *time-to-mine* and *memory*. The time-to-mine metric measures the time required to perform the mining. The authors use this metric to show how their approach scales with respect to the input size (e.g., length of the input trace, size of the DUV, etc.). The memory metric measures the peak working memory used by the approach to mine the specifications. 29% of the reviewed papers employ one or both of these metrics.
- **Between manual and automatic**. Several authors evaluate the effectiveness of their mining approaches by comparing mined specifications against a set of manually predefined ones. In this context, a human operator defines a set of specifications or informal requirements representing expected behaviors. The mining tool is

then used to extract specifications, and the results are compared by the human operator against the manual set. The mined specifications are considered optimal if they fully cover the manual specifications (capturing all intended behaviors of the DUV) and do not show erroneous or unexpected behaviors. This approach is applied in 28% of the reviewed papers.

VIII. CHALLENGES AND FUTURE DIRECTIONS

In this section, we address the main challenges of mining LTL specifications and explore possible future research directions.

A. MAIN CHALLENGES OF SPECIFICATION MINING

Even though the literature presents a staggering number of different techniques to generate specifications, the challenges faced by most mining approaches are predominantly analogous.

1) CHALLENGES RELATED TO THE KIND OF INPUT

The first set of challenges is related to the kind of input the technique requires. In particular, the techniques described in this paper take as input the execution traces, the source code of the DUV, or a set of informal specifications written in English.

1) **Only execution traces**. In our experience, this is usually the most difficult case, often reserved for scenarios where the design is unavailable. The first challenge is the absence of an initial clustering of the events (or DUV variables) in the trace: all events or variables usually appear in "list form" without hinting whether a group of events should or should not appear together in a formal specification. For example, consider the following execution trace.

```
bl_ack, bl_re, core_in, core_out,...
0 0 1 0 ...
1 0 1 0 ...
1 0 1 0 ...
0 0 0 1 ...
```

That trace does not directly indicate how to group the variables meaningfully. Should we infer that `bl_ack` and `core_in` belong together in a specification or that they are independent events occurring in this trace? A possible solution would be to group together events whose names follow certain patterns (e.g., the name similarity metric of Perracotta [44] or the approach in [27]), exploiting the natural tendencies of programmers to add prefixes or suffixes to related variables. However, even if we managed to cluster related events, there is no information on the legal sequential ordering in which events should appear in a formal specification. For example, consider the LTL template $G(P0 \rightarrow P1)$. How can we know which events should represent the trigger or the effect of this implication? The aforementioned approach could be applied to retrieve hints on the legal order of events. For example, in RTL hardware representations, there is a tendency to add the prefix "in_" or "out_" to input and output ports, respectively; therefore, in the previous example, you could exploit that information to determine which signals should go in the antecedent (in) and which ones should go into the consequent (out). Regardless, that can hardly be considered a reliable approach; at the same time, several authors (especially those appearing in section VI-A) greatly emphasize the importance of preemptively clustering the input events.

The second challenge is the tight complexity bounds of specification mining from examples. Learning FSAs from examples is a well-known NP-hard problem proven in [185]. Moreover, Gabel and Su showed that the pattern-based specification mining problem is NP-complete in its general form. More recently, Fijalkow et al. [186] investigated the computational complexity of learning minimal LTL formulas from examples, proving NP-hardness for several fragments; however, the NP-hardness of the entire LTL remains still an open problem.

Finally, in our experience, when mining from execution traces, generating minimal complex properties (simpler specifications are often preferred to improve readability and avoid noise) is usually difficult due to the relation between the complexity of a specification and the number of examples required to justify its complexity. Consider a specification s_1 of the form $G(v_1 \,\&\&\, v_2 \rightarrow v_3)$ that holds on the only input trace tr. This specification is minimal only if $s_2 : G(v_1 \rightarrow v_3)$ and $s_3 : G(v_2 \rightarrow v_3)$ does not hold on tr. Otherwise, if either s_1 or s_2 holds on tr, they represent a smaller and more readable (and possibly of higher support) variant (that represents a simpler trigger of the consequent) of s_1. Therefore, to justify mining the more complex minimal specification s_1, the trace must contain two counterexamples in tr to make s_1 and s_2 fail.

2) **With the source code of the design**. If the mining approach takes as input the source code of the design, then some of the above problems are automatically solved. In particular, we can exploit the modularity of the design to group together related events; then, we can generate the COI of any target to determine a sequencing order (or a cause-effect relation) among the events (see sections VI-A and VI-D). Nonetheless, taking the design as input introduces new issues. Above all, the technical pitfall of instrumenting the design with annotations required to execute the methodology. In academic publications, this process is usually performed manually by the authors and is considered a minor technical issue [35]. However, in our experience, that is usually not the case due to the huge number of different models (languages) available to implement HW and SW systems, each requiring a parser to

interpret the code and a custom instrumentator to add the annotations. Moreover, these ad-hoc elements must be kept updated with new versions of the models. In practice, open-source tools support only a small subset of a language, making them unusable in most industrial settings.

If the DUV is the only input and the mining approach is fully static (does not execute the design), then it is limited to what can be inferred from the model, which may result in missing contextual or emergent behaviors observed only during execution (e.g., race conditions, stack overflows, etc.). As a consequence, when the approach requires the model of the DUV, it also requires a set of meaningful testbenches to explore its dynamic behaviors. Some authors mitigated the issue by employing automatic approaches to generate random testbenches [64], or used symbolic simulation approaches to generate high-coverage tests [180]; however, these solutions exacerbate the issue of generating additional software to handle the simulation aspects to generate significant tests.

3) **Informal specifications**. If the mining approach takes as input a set of informal specifications with the goal of generating formal specifications, the following limitations apply.

Ambiguity of natural language: human language is inherently ambiguous, with phrases often having multiple meanings or interpretations; therefore, the available techniques may struggle to resolve ambiguities and translate imprecise or vague instructions into specific, actionable formal specifications. Conversely, translating natural language into LTL can lead to the loss of nuanced meanings in the original text, omitting critical details that can not be formalized in LTL.

Inconsistencies in input: informal specifications often contain inconsistencies or contradictions, forcing the mining technique to make arbitrary decisions on which parts of the text to prioritize or ignore.

Difficulty with Highly Structured Input: NLP models tend to perform better with continuous and somewhat flexible inputs (like narratives) but may struggle with highly structured data, such as technical specifications that require rigid and formal representations.

Variability in Linguistic Expressions and Non-deterministic Output: NLP models may not handle linguistic variability well. Different phrasings of the same intent might not be consistently understood or translated into the same specification, leading to inconsistent outputs. Furthermore, due to the stochastic nature of those models, slight variations in the input can lead to different results, even when the underlying meaning is the same. This can make it difficult to ensure repeatability or reliability of the mined specifications.

Domain-specific nature: most NLP techniques must be customized or tuned to a particular domain, which limits their general applicability across diverse fields: what works well for one type of text or specification might not work for another.

2) OTHER CHALLENGES

In this section, we report a comprehensive list of other challenges of the specification mining techniques reported in this paper.

- **Preprocessing overhead**. The preprocessing stage adds significant time and resource overhead. Furthermore, any mistakes or inaccuracies in the preprocessing can propagate through the learning process, resulting in erroneous specifications.
- **Template dependency**. The technique's effectiveness depends highly on the pre-defined templates. If templates do not cover certain behaviors, relevant properties may never be inferred.
- **Scalability**. Most mining techniques face combinatorial explosion as the number of permutations grows exponentially with the number of variables or events. This happens because the mining process does not scale well unless greedy algorithms are used to reduce the search space. Conversely, using heuristics prevents exploring all possible solutions, leading to missed properties. Consequently, many specification mining techniques struggle to handle large systems, impacting their applicability to real-world scenarios. Furthermore, long or complex traces can make the mining process computationally expensive, while window-based techniques may miss long-range dependencies or patterns that span multiple windows. Finally, most mining techniques require storing the entire trace in memory, which can become a bottleneck for large datasets.
- **Overfitting and generalization**. Balancing between underfitting (too general) and overfitting (too specific) the model to the traces. Over-generalized models miss important specific behaviors, while overly specific models fail to cover broader behaviors. Moreover, mining techniques may find patterns that are specific to the training data but do not generalize well to new data.
- **Overflow**. The mining tool might generate too many specifications, which can be overwhelming and difficult to manage. Therefore, complex ranking techniques are needed to filter redundant assertions and prioritize the most important ones. However, identifying an effective ranking mechanism is still an open problem.
- **Human interpretation and effort**. The mined specifications are often in a form that requires expert interpretation. Translating learned models into actionable insights or understandable specifications for non-experts can be difficult. Complex specifications might be difficult to interpret even for experts.
- **Threshold sensitivity**. Setting appropriate minimum support, confidence, or other thresholds is often tricky. Too high and meaningful patterns may be missed; too low and noise may be considered as patterns. While

the support-confidence framework is widely used, it can miss useful but infrequent patterns or overemphasize frequent but trivial ones.

- **Noise in data**. Irrelevant events or noisy data can affect the mining process, leading to inaccurate or incomplete specifications. Mining techniques might need robust filtering to avoid generating misleading specifications.
- **Trace coverage and incompleteness**. The inferred properties depend on the quality and completeness of the program traces. Specification mining often relies on the assumption that the execution traces cover all behaviors. In practice, traces are often incomplete, making the inferred process miss critical behaviors.
- **Non-deterministic, probabilistic, and parallel behavior**. If the system exhibits behavior that does not fit into regular languages, the inferred specification will be inadequate. Non-deterministic behaviors (such as race conditions or parallelism) are hard to model with deterministic finite automata. LTL may struggle to represent systems with inherent non-deterministic or concurrent behaviors. This may limit its applicability to concurrent or parallel systems, where interactions occur in non-sequential patterns. Furthermore, many techniques fail to account for uncertainty or probabilistic behaviors in the system; this results in models that are too rigid to handle their stochastic nature.

B. FUTURE DIRECTIONS

In this section, our experience meets the knowledge in the reviewed paper to outline a set of opportunities for future research directions.

- **Putting everything together**. Future research should focus on integrating existing mining methods to create a unified framework. These methods should complement one another by fully exploiting the input data, whether it is source code, execution traces, or informal system requirements. A holistic approach will ensure that no valuable information is overlooked, and results can be more accurate and comprehensive.
- **Customizability and adaptability**. A critical aspect of future research is making LTL mining tools customizable to adapt to diverse use cases. As reported in section V-A, most use cases are fundamentally overlapping while most techniques could be applied to most use cases; however, the lack of customization of the available tools is a major barrier to achieving that goal. Moreover, even when using a tool specialized for a certain use case, users should be able to tweak parameters based on their specific settings, heavily increasing the quality of the generated specifications.
- **Testability and evaluation**. A robust and standard way to test and evaluate LTL miners is essential. As of now, there is little consensus on how to determine if new approaches are superior to the state-of-the-art. A set of standard test cases and evaluation metrics should be developed, covering a wide spectrum of real-world systems and theoretical benchmarks. This could help bridge the gap between research prototypes and practical, reliable tools. Furthermore, there is still limited comparative analysis between the various mining techniques. An extendable and customizable evaluation framework should be developed to allow a fair comparison, assessing their actual performance under different conditions.
- **Ease of use**. The user experience of mining tools is paramount. These tools should not introduce a significant overhead compared to manually writing assertions: using the tool should not be harder than directly hiring a verification engineer to formalize the specifications manually.
- **Mining assisted by Artificial Intelligence (AI)**. With the recent advancements in machine learning and natural language processing, developing AI assistants that can help generate LTL templates would be a promising future direction. Such an assistant could guide users through common patterns, offer suggestions based on system behavior, and help with customization, significantly speeding up and simplifying the process. This approach could be applied to any aspect of the mining procedure.
- **Maturity of tools**. Currently, many LTL mining tools are in a prototype phase and lack the maturity for widespread industrial adoption. A concerted effort towards improving their robustness and usability would ensure these tools can transition from research artifacts to dependable industrial applications.
- **Qualification**. Qualification of mined LTL specifications remains an unsolved challenge. It's unclear how to assess the correctness or completeness of specifications extracted by mining algorithms. The only clear aspect is that qualification highly depends on the mining context [61].
- **Interoperability with existing workflows**. Integration with existing verification and testing environments should be a priority. Research should explore how LTL mining tools can seamlessly integrate with prevalent Hardware Description Languages (HDLs), model checkers, or software testing frameworks, ensuring interoperability and facilitating adoption by practitioners who already have established toolchains.
- **Scalability**. As systems grow in complexity, the ability to mine specifications from large-scale systems, both in hardware and software, will become increasingly critical. This includes handling more complex models, dealing with distributed systems, and efficiently processing large amounts of log data. Research should focus on improving the scalability of existing mining algorithms and adapting them for high-performance computing environments.

IX. CONCLUSION

In this paper, we explored the landscape of mining LTL specifications in both hardware and software systems. The field has shown significant progress in the last decade, with a variety of approaches developed to extract specifications from diverse sources of information, such as system traces, source code, and formal models. Looking ahead, there is considerable potential for growth and innovation in this area. As discussed in the future directions section, the emphasis should be placed on enhancing the testability, scalability, and ease of use of LTL mining tools to ensure they meet the demands of real-world settings.

ACRONYMS

AFA Alternating Finite Automata. 30
AI Artificial Intelligence. 44
API Application Programming Interface. 4, 5, 11, 15, 16, 19, 39
ARM Association Rule Mining. 5
ASP Answer Set Programming. 37
BDD Binary Decision Diagram 33, 37
BGDF Best-Gain Decision Forest. 26, 28
BN Bayesian Network. 36, 37
CDFG Control Data Flow Graph. 26, 27
CFG Control Flow Graph. 24
COI Cone of Influence. 26, 27, 42
CPS Cybe-Physical System. 5
CPU Central Processing Unit. 6
CTL Computation Trex Logic. 4, 7
DAG Directed Acyclic Ciraph. 29, 30, 36
DDG Dynamic Dependency Ciraph. 38
DG Dependency Graph. 38
DT Decision Tree. 25–30
DTO Decision Tree Operator. 28
DUV Design Under Verification.1, 3, 6, 7, 11, 16, 17, 26, 29, 34, 40–42
FCPO Frequently Closed Partial Onter. 39
gFSA Guarded Finite Stak Automaton. 14, 15
GNN Giraph Neural Network. 35, 36
GPU Graphics Processing Unit. 20–22
GSP Generalined Sequential Pattern. 22, 24
HDLs Hardware Description Languager. 44
HW Hardware. 2-4, 6, 19, 21, 34, 42
IFT Information Flow Tracking. 39
IG Information Gain. 26
IR Intermediate Representation. 33
IVs Inveresting Values. 29
LLM Large Language Model. 31, 32
LSTM Long Short-Term Memory. 30, 31
LTL Linear Temporal Logic. 1-5, 7-9, 14, 16-20, 23, 28-44
LTLI Linear Temporal Logic on finite traces. 7
MDL Minimum Description Length. 33
NFSA Non-Deterministic Finice State Automaton. 8, 12
NLP Natural Language Processing. 30, 31, 43
PFSA Probabilistic Finite State Automaton.12-14
PRISMA Preferred Reporting Items for Systematic Reviews and Meta-Analyses. 2, 4, 5
PSL Property Specification Language. 3, 7
PTA Irefix Tree Acceptor. 17
PTLTL Past-Time Linear Temporal Logic. 18
RTL Register Transfer Level. 25-28, 38, 39, 42
SAT Satisfiability. 11, 29, 30, 33, 39
SERE Sequential Extended Regular Expression. 3, 4, 7, 8, 18, 28
SoC System on a chip. 5, 39
STL Signal Temporal Logic. 4, 6
SVA SystemVerilog Assertion. 3, 7, 23, 25, 31-33, 38
SW Software. 2-4, 6, 11, 13, 16, 42
UD-Chain Use-Definition Chain. 26
VHDL VHSIC Hardware Description Language. 6
WCSS Within-Cluster Sum Of Squares. 29
WP Weakest Precondition. 27, 28

REFERENCES

[1] C. Wang, F. He, X. Song, Y. Jiang, M. Gu, and J. Sun, "Assertion recommendation for formal program verification," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2017, pp. 154–159.
[2] M. J. Page et al., "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews," *BMJ*, vol. 372, p. 71, Mar. 2021.
[3] *Scopus*. Accessed: Feb. 18, 2025. [Online]. Available: www.scopus.com
[4] *Web of Science*. Accessed: Feb. 18, 2025. [Online]. Available: https://www.webofscience.com
[5] *IEEE Xplore Digital Library*. Accessed: Feb. 18, 2025. [Online]. Available: https://ieeexplore.ieee.org/Xplore/home.jsp
[6] (2025). *Google Scholar*. Accessed: Feb. 18, 2025. [Online]. Available: https://scholar.google.com
[7] *Scopus Search Tips Guide*. Accessed: Feb. 18, 2025. [Online]. Available: schema.elsevier.com/dtds/document/bkapi/search/SCOPUSSearchTips.htm
[8] (2025). *Rayyan AI*. Accessed: Feb. 18, 2025. [Online]. Available: https://new.rayyan.ai/
[9] *Scopus API Documentation*. Accessed: Feb. 18, 2025. [Online]. Available: https://dev.elsevier.com/
[10] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, May 2013.
[11] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 178–189.
[12] S. M. Ghafari and C. Tjortjis, "A survey on association rules mining using heuristics," *WIREs Data Mining Knowl. Discovery*, vol. 9, no. 4, Jul. 2019, Art. no. e1307.
[13] M. J. Mashhadi and H. Hemmati, "An empirical study on practicality of specification mining algorithms on a real-world application," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, May 2019, pp. 65–69.
[14] M. R. Aliabadi, H. Haghighi, M. V. Asl, and R. G. Meybodi, "Challenges of specification mining-based test Oracle for cyber-physical systems," in *Proc. 11th Int. Conf. Inf. Knowl. Technol. (IKT)*, Dec. 2020, pp. 1–7.
[15] M. R. Ahmed, H. Zheng, P. Mukherjee, M. C. Ketkar, and J. Yang, "A comparative study of specification mining methods for SoC communication traces," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2021, pp. 31–36.
[16] D. Neider and R. Roy, "What is formal verification without specifications? A survey on mining LTL specifications," in *Principles of Verification: Cycling the Probabilistic Landscape*. Cham, Switzerland: Springer, 2025.

[17] E. Bartocci, C. Mateis, E. Nesterini, and D. Nickovic, "Survey on mining signal temporal logic specifications," *Inf. Comput.*, vol. 289, Nov. 2022, Art. no. 104957.

[18] *Standard for Property Specification Language (PSL)*, IEEE Standard IEC 62531, 2012.

[19] *IEEE Standard for Systemverilog–unified Hardware Design, Specification, and Verification Language—Redline*, IEEE Standard 1800-2009, 2009.

[20] M. R. Heidari Iman, G. Jervan, and T. Ghasempouri, "ARTmine: Automatic association rule mining with temporal behavior for hardware verification," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2024, pp. 1–6.

[21] A. Danese, N. D. Riva, and G. Pravadelli, "A-TEAM: Automatic template-based assertion miner," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[22] D. Lo, G. Ramalingam, V.-P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," *Sci. Comput. Program.*, vol. 77, no. 6, pp. 743–759, Jun. 2012.

[23] E. El Mandouh and A. G. Wassal, "Automatic generation of hardware design properties from simulation traces," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2012, pp. 2317–2320.

[24] L. Liu, D. Sheridan, V. Athavale, and S. Vasudevan, "Automatic generation of assertions from system level design using data mining," in *Proc. 9th ACM/IEEE Int. Conf. Formal Methods Models Codesign (MEMPCODE)*, Jul. 2011, pp. 191–200.

[25] X. Cheng and M. S. Hsiao, "Simulation-directed invariant mining for software verification," in *Proc. Design, Autom. Test Eur.*, Mar. 2008, pp. 682–687.

[26] P. K. Mahato and A. Narayan, "MINTS: Unsupervised temporal specifications miner," in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Dec. 2021, pp. 841–851.

[27] H. J. Kang and D. Lo, "Adversarial specification mining," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 1–40, Apr. 2021.

[28] Y. Gao, M. Wang, and B. Yu, "Dynamic specification mining based on transformer," in *Theoretical Aspects of Software Engineering*. Cham, Switzerland: Springer, 2022.

[29] T.-D.-B. Le and D. Lo, "Deep specification mining," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2018, pp. 106–117.

[30] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proc. 19th ACM SIGSOFT Symp., 13th Eur. Conf. Found. Softw. Eng.*, Sep. 2011, pp. 267–277.

[31] X. Ning, N. Zhang, Z. Duan, and C. Tian, "PPTL specification mining based on LNFG," *Theor. Comput. Sci.*, vol. 937, pp. 85–95, Nov. 2022.

[32] N. Zhang, X. Yuan, and Z. Duan, "Propositional projection temporal logic specification mining," in *Combinatorial Optimization and Applications*. Cham, Switzerland: Springer, 2020.

[33] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *Proc. Int. Workshop Dyn. Syst. Anal.*, May 2006, pp. 17–24.

[34] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proc. 19th Int. Symp. Softw. Test. Anal.*, Jul. 2010, pp. 85–96.

[35] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proc. Conf. Rec. Annu. ACM Symp. Princ. Program. Lang.*, Jan. 2002, pp. 4–16.

[36] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning," in *Runtime Verification*. Cham, Switzerland: Springer, 2014.

[37] L. Mariani, M. Pezzè, and M. Santoro, "GK-Tail+ an efficient approach to learn software models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 8, pp. 715–738, Aug. 2017.

[38] S. Huang and R. Cleaveland, "Temporal-logic query checking over finite data streams," *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 3, pp. 473–492, Jun. 2022.

[39] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 651–666, Sep. 2008.

[40] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, p. 501.

[41] D. Lo and S.-C. Khoo, "SMArTIC: Towards building an accurate, robust and scalable specification miner," in *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2006, pp. 265–275.

[42] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2008, pp. 339–349.

[43] R. Raha, R. Roy, N. Fijalkow, and D. Neider, "Scalable anytime algorithms for learning fragments of linear temporal logic," in *Tools and Algorithms for the Construction and Analysis of Systems*. Cham, Switzerland: Springer, 2022.

[44] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proc. 28th Int. Conf. Softw. Eng.*, May 2006, pp. 282–291.

[45] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. 47th Design Autom. Conf.*, Jun. 2010, pp. 755–760.

[46] L.-C. Wang, M. S. Abadir, and N. Krishnamurthy, "Automatic generation of assertions for formal verification of PowerPC microprocessor arrays using symbolic trajectory evaluation," in *Proc. 35th Annu. Design Autom. Conf.*, 1998, pp. 534–537.

[47] T. B. Le, X. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 115–125.

[48] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Automatic generation of complex properties for hardware designs," in *Proc. Design, Autom. Test Eur.*, Mar. 2008, pp. 545–548.

[49] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli, "Dynamic property mining for embedded software," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2012, pp. 187–196.

[50] C. Deutschbein and C. Sturton, "Evaluating security specification mining for a CISC architecture," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Dec. 2020, pp. 164–175.

[51] A. Danese, V. Bertacco, and G. Pravadelli, "Symbolic assertion mining for security validation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1550–1555.

[52] J. Kim, C. Muise, A. Shah, S. Agarwal, and J. Shah, "Bayesian inference of linear temporal logic specifications for contrastive explanations," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 5591–5598.

[53] M. Bertasi, G. Di Guglielmo, and G. Pravadelli, "Automatic generation of compact formal properties for effective error detection," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2013, pp. 1–10.

[54] T. Ghasempouri, J. Malburg, A. Danese, G. Pravadelli, G. Fey, and J. Raik, "Engineering of an effective automatic dynamic assertion mining platform," in *Proc. IFIP/IEEE 27th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2019, pp. 111–116.

[55] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2010, pp. 387–396.

[56] D. Lo and S. Maoz, "Specification mining of symbolic scenario-based models," in *Proc. 8th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, Nov. 2008, pp. 29–35.

[57] M. O. Kayed, M. Abdelsalam, and R. Guindi, "Synthesizable SVA protocol checker generation methodology based on TDML and VCD file formats," in *Proc. IEEE Int. High Level Design Validation Test Workshop (HLDVT)*, Oct. 2016, pp. 1–8.

[58] X. Huo, K. Hao, L. Chen, X.-S. Tang, T. Wang, and X. Cai, "A dynamic soft sensor of industrial fuzzy time series with propositional linear temporal logic," *Expert Syst. Appl.*, vol. 201, Sep. 2022, Art. no. 117176.

[59] M. Miyamoto and K. Hamaguchi, "Extracting hardware assertions including word-level relations over multiple clock cycles," in *Proc. 19th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2018, pp. 244–250.

[60] D. Neider and I. Gavran, "Learning linear temporal properties," in *Proc. Formal Methods Comput. Aided Design (FMCAD)*, Oct. 2018, pp. 1–10.

[61] S. Germiniani and G. Pravadelli, "HARM: A hint-based assertion miner," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4277–4288, Nov. 2022.

[62] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 6, pp. 952–965, Jun. 2013.

[63] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "GoldMine: Automatic assertion generation using data mining and static analysis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2010, pp. 626–629.

[64] S. Vasudevan, L. Liu, and S. Hertz, "A comparative study of assertion mining algorithms in GoldMine," in *Machine Learning in VLSI Computer-Aided Design*. Cham, Switzerland: Springer, 2019.
[65] S. Germiniani and G. Pravadelli, "Exploiting clustering and decision-tree algorithms to mine LTL assertions containing non-Boolean expressions," in *Proc. IFIP/IEEE 30th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2022, pp. 1–6.
[66] J.-R. Gaglione, D. Neider, R. Roy, U. Topcu, and Z. Xu, "MaxSAT-based temporal logic inference from noisy data," *Innov. Syst. Softw. Eng.*, vol. 18, no. 3, pp. 427–442, Sep. 2022.
[67] J. R. Gaglione, D. Neider, R. Roy, U. Topcu, and Z. Xu, "Learning linear temporal properties from noisy data: A MaxSAT-based approach," in *Automated Technology for Verification and Analysis*. Cham, Switzerland: Springer, 2021.
[68] L. Liu, C.-H. Lin, and S. Vasudevan, "Word level feature discovery to enhance quality of assertion mining," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2012, pp. 210–217.
[69] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 35–45, Dec. 2007.
[70] C. Lemieux, "Mining temporal properties of data invariants," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, May 2015, pp. 751–753.
[71] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton, "Isadora: Automated information-flow property generation for hardware security verification," *J. Cryptograph. Eng.*, vol. 13, no. 4, pp. 391–407, Nov. 2023.
[72] T. Xie and D. Notkin, "Mutually enhancing test generation and specification inference," in *Formal Approaches To Software Testing*. Cham, Switzerland: Springer, 2004.
[73] C. Wang, Y. Cai, Q. Zhou, and H. Wang, "ASAX: Automatic security assertion extraction for detecting hardware trojans," in *Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2018, pp. 84–89.
[74] C. B. Harris and I. G. Harris, "GLAsT: Learning formal grammars to translate natural language specifications into hardware assertions," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2016, pp. 966–971.
[75] S. Katz and A. Rashid, "From aspectual requirements to proof obligations for aspect-oriented systems," in *Proc. 12th IEEE Int. Requirement Eng. Conf.*, Jun. 2004, pp. 43–52.
[76] R. Patel, E. Pavlick, and S. Tellex, "Grounding language to non-Markovian tasks with no supervision of task specifications," in *Proc. Robot., Sci. Syst.*, Corvalis, OR, USA, 2020.
[77] S. Zhang, J. Zhai, L. Bu, M. Chen, L. Wang, and X. Li, "Automated generation of LTL specifications for smart home IoT using natural language," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 622–625.
[78] J. Zhao and I. G. Harris, "Automatic assertion generation from natural language specifications using subtree analysis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 598–601.
[79] R. Krishnamurthy and M. S. Hsiao, "Controlled natural language framework for generating assertions from hardware specifications," in *Proc. IEEE 13th Int. Conf. Semantic Comput. (ICSC)*, Jan. 2019, pp. 367–370.
[80] O. Keszocze and I. G. Harris, "Chatbot-based assertion generation from natural language specifications," in *Proc. Forum Specification Design Lang. (FDL)*, Sep. 2019, pp. 1–6.
[81] X. Pi, J. Shi, Y. Huang, and H. Wei, "Automated mining and checking of formal properties in natural language requirements," in *Knowledge Science, Engineering and Management*. Cham, Switzerland: Springer, 2019.
[82] F. Aditi and M. S. Hsiao, "Validatable generation of system verilog assertions from natural language specifications," in *Proc. 5th Int. Conf. Transdisciplinary AI (TransAI)*, Sep. 2023, pp. 102–109.
[83] J. Shivamurthy, D. Vidyarthi, and T. Uppal, "Natural language processing based auto generation of proof obligations for formal verification of control requirements in safety-critical systems," *IFAC-PapersOnLine*, vol. 57, pp. 1–6, Jan. 2024.
[84] J. Pan, G. Chou, and D. Berenson, "Data-efficient learning of natural language to linear temporal logic translators for robot task specification," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2023, pp. 11554–11561.
[85] S. J. Frederiksen, J. Aromando, and M. S. Hsiao, "Automated assertion generation from natural language specifications," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2020, pp. 1–5.
[86] S. Liu, D. Li, Y. Chen, and G. Yang, "(Semi) automatic assertion generation from controlled Chinese natural language: A practice in aerospace industry," in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Dec. 2021, pp. 778–782.
[87] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: Automatic requirements specification extraction from natural language," in *NASA Formal Methods*. Cham, Switzerland: Springer, 2016.
[88] C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit, "Provably correct reactive control from natural language," *Auto. Robots*, vol. 38, no. 1, pp. 89–105, Jan. 2015.
[89] A. P. Nikora and G. Balcom, "Automated identification of LTL patterns in natural language requirements," in *Proc. 20th Int. Symp. Softw. Rel. Eng.*, Nov. 2009, pp. 185–194.
[90] J. X. Liu, Z. Yang, I. Idrees, S. Liang, B. Schornstein, S. Tellex, and A. Shah, "Lang2LTL: Translating natural language commands to temporal robot task specification," 2023, *arXiv:2302.11649*.
[91] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," in *Computer Aided Verification*. Cham, Switzerland: Springer, 2023.
[92] H. Cherukuri, A. Ferrari, and P. Spoletini, "Towards explainable formal methods: From LTL to natural language with neural machine translation," in *Requirements Engineering: Foundation for Software Quality*. Cham, Switzerland: Springer, 2022.
[93] N. Bombieri, F. Busato, A. Danese, L. Piccolboni, and G. Pravadelli, "Mangrove: An inference-based dynamic invariant mining for GPU architectures," *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 606–620, Apr. 2020.
[94] S. Lutz, D. Neider, and R. Roy, "Specification sketching for linear temporal logic," in *Automated Technology for Verification and Analysis*. Cham, Switzerland: Springer, 2023.
[95] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Detecting the behavior of design patterns through model checking and dynamic analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 4, pp. 1–41, Oct. 2017.
[96] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 81–92.
[97] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, May 2010, pp. 15–24.
[98] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *Proc. 14th Asian Test Symp. (ATS)*, 2005, pp. 34–39.
[99] J. Shi, J. Xiong, and Y. Huang, "General past-time linear temporal logic specification mining," *CCF Trans. High Perform. Comput.*, vol. 3, no. 4, pp. 393–406, Dec. 2021.
[100] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proc. 42nd Design Autom. Conf.*, 2005, pp. 775–778.
[101] A. Bunker, G. Gopalakrishnan, and K. Slind, "Live sequence charts applied to hardware requirements specification and verification: A VCI bus interface model," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 341–350, Aug. 2005.
[102] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science), vol. 3440. Berlin, Germany: Springer, 2005, pp. 461–476.
[103] V. Iyer, D. Kim, B. Nikolic, and S. A. Seshia, "RTL bug localization through LTL specification mining (WIP)," in *Proc. 17th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design*, Oct. 2019, pp. 1–5.
[104] E. Ghiorzi, M. Colledanchise, G. Piquet, S. Bernagozzi, A. Tacchella, and L. Natale, "Learning linear temporal properties for autonomous robotic systems," *IEEE Robot. Autom. Lett.*, vol. 8, no. 5, pp. 2930–2937, May 2023.
[105] A. Shah, P. Kamath, J. Shah, and S. Li, "Bayesian inference of temporal task specifications from demonstrations," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, Jan. 2018, pp. 3804–3813.

[106] T. Ghasempouri, A. Danese, G. Pravadelli, N. Bombieri, and J. Raik, "RTL assertion mining with automated RTL-to-TLM abstraction," in *Proc. Forum Specification Design Lang. (FDL)*, Sep. 2019, pp. 1–8.

[107] J. Yang and D. Evans, "Dynamically inferring temporal properties," in *Proc. 5th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, Jun. 2004, pp. 23–28.

[108] M. O. Kayed, M. Abdelsalam, and R. Guindi, "A novel approach for SVA generation of DDR memory protocols based on TDML," in *Proc. 15th Int. Microprocessor Test Verification Workshop*, Dec. 2014, pp. 61–66.

[109] A. Narayan and S. Fischmeister, "Mining time for timed regular specifications," in *Proc. IEEE Int. Conf. Syst., Man Cybern. (SMC)*, Oct. 2019, pp. 63–69.

[110] N. Zhang, B. Yu, C. Tian, Z. Duan, and X. Yuan, "Temporal logic specification mining of programs," *Theor. Comput. Sci.*, vol. 857, pp. 29–42, Feb. 2021.

[111] A. Narayan, N. Benann, and S. Fischmeister, "Mining specifications using nested words," in *Proc. 6th Int. Workshop Softw. Mining (SoftwareMining)*, Nov. 2017, pp. 9–16.

[112] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2007, pp. 465–468.

[113] D. Lo, S. C. Khoo, and C. Liu, "Efficient mining recurrent rules from a sequence database," in *Database Systems for Advanced Applications*, J. R. Haritsa, R. Kotagiri, and V. Pudi, Eds., Berlin, Germany: Springer, 2008, pp. 67–83.

[114] A. Danese, F. Filini, T. Ghasempouri, and G. Pravadelli, "Automatic generation and qualification of assertions on control signals: A time window-based approach," in *VLSI-SoC: Design for Reliability, Security, and Low Power*. New York, NY, USA: Springer, 2016.

[115] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Proc. 9th ACM/IEEE Int. Conf. Formal Methods Models Codesign (MEMPCODE)*, Jul. 2011, pp. 43–50.

[116] M. Valizadeh, N. Fijalkow, and M. Berger, "LTL learning on GPUs," in *Computer Aided Verification*. Cham, Switzerland: Springer, 2024.

[117] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2007, pp. 460–469.

[118] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 123–134, Jun. 2007.

[119] M. R. Heidari Iman, J. Raik, M. Jenihhin, G. Jervan, and T. Ghasempouri, "An automated method for mining high-quality assertion sets," *Microprocessors Microsystems*, vol. 97, Mar. 2023, Art. no. 104773.

[120] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Proc. 15th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2010, pp. 607–612.

[121] A. Mrowca, M. Nocker, S. Steinhorst, and S. Günnemann, "Learning temporal specifications from imperfect traces using Bayesian inference," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2019, pp. 1–6.

[122] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Sep. 2007, pp. 25–34.

[123] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, p. 51.

[124] J. Malburg, T. Flenker, and G. Fey, "Property mining using dynamic dependency graphs," in *Proc. 22nd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2017, pp. 244–250.

[125] M. Vazquez-Chanlatte, S. Jha, A. Tiwari, M. K. Ho, and S. A. Seshia, "Learning task specifications from demonstrations," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, Jan. 2018, pp. 5367–5377.

[126] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. 1st India Softw. Eng. Conf.*, Feb. 2008, pp. 5–14.

[127] S. Germiniani, A. Danese, and G. Pravadelli, "Automatic generation of assertions for detection of firmware vulnerabilities through alignment of symbolic sequences," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 2, pp. 728–739, Apr. 2022.

[128] H. Witharana, A. Jayasena, A. Whigham, and P. Mishra, "Automated generation of security assertions for RTL models," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 19, no. 1, pp. 1–27, Jan. 2023.

[129] A. Shah, P. Kamath, S. Li, P. Craven, K. Landers, K. Oden, and J. Shah, "Supervised Bayesian specification inference from demonstrations," *Int. J. Robot. Res.*, vol. 42, no. 14, pp. 1245–1264, Dec. 2023.

[130] W. Luo, P. Liang, J. Du, H. Wan, B. Peng, and D. Zhang, "Bridging LTL inference to GNN inference for learning LTLf formulae," in *Proc. 36th AAAI Conf. Artif. Intell.*, 2022, pp. 9849–9857.

[131] H. Wan, P. Liang, J. Du, W. Luo, R. Ye, and B. Peng, "End-to-end learning of LTLf formulae by faithful LTLf encoding," in *Proc. AAAI Conf. Artif. Intell.*, vol. 38, Mar. 2024, pp. 9071–9079.

[132] A. Ielo, M. Law, V. Fionda, F. Ricca, G. D. Giacomo, and A. Russo, "Towards ILP-based LTL$_f$ passive learning," in *Inductive Logic Programming*. Cham, Switzerland: Springer, 2023.

[133] R. Roy, J.-R. Gaglione, N. Baharisangari, D. Neider, Z. Xu, and U. Topcu, "Learning interpretable temporal properties from positive examples only," in *Proc. 37th AAAI Conf. Artif. Intell., AAAI*, vol. 37, Jun. 2023, pp. 6507–6515.

[134] S. Germiniani, D. Nicoletti, and G. Pravadelli, "Invited talk: Pros and cons of assertion mining," in *Proc. IEEE 25th Latin Amer. Test Symp. (LATS)*, Apr. 2024, pp. 1–2.

[135] A. Bosio, S. Germiniani, G. Pravadelli, and M. Traiola, "Exploiting assertions mining and fault analysis to guide RTL-level approximation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–2.

[136] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Comput.*, vol. C-21, no. 6, pp. 592–597, Jun. 1972.

[137] A. Raman, J. Patrick, and P. North, "The sk-strings method for inferring PFSA," in *Proc. 14th Int. Conf. Mach. Learn. (ICML)*, 1997, pp. 1–7.

[138] J. Wang and J. Han, "BIDE: Efficient mining of frequent closed sequences," in *Proc. 20th Int. Conf. Data Eng.*, 2004, pp. 79–90.

[139] C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby, "Compression by induction of hierarchical grammars," in *Proc. IEEE Data Compress. Conf. (DCC)*, Mar. 1994, pp. 244–253.

[140] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, 1970. [Online]. Available: https://doi.org/10.1016/0022-2836(70)90057-4

[141] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction To Cluster Analysis*. New York, NY, USA: Wiley, 1990.

[142] *Synoptic GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/ModelInference/synoptic/

[143] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.

[144] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 468–478.

[145] P. Cousot and R. Cousot, "Abstract interpretation," in *Proc. Conf. Rec. Annu. ACM Symp. Princ. Program. Lang.*, Jan. 1977, pp. 238–252.

[146] *Dice Github Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/kanghj/DICE

[147] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, Sep. 2011, pp. 416–419.

[148] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[149] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, Jun. 2017, pp. 5998–6008.

[150] K. M. Olender and L. J. Osterweil, "Cecil: A sequencing constraint language for automatic static analysis generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 3, pp. 268–280, Mar. 1990.

[151] *Texada GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/ModelInference/texada

[152] *Harm GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/SamueleGerminiani/harm

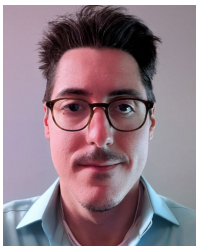
[153] *Perracotta GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/ModelInference/perracotta

[154] *Learning Ltl With GPUs GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/MojtabaValizadeh/ltl-learning-on-gpus?tab=readme-ov-file

[155] *Daikon GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/codespecs/daikon

[156] *Mangrove GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/mangrove-univr/Mangrove
[157] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th Int. Conf. Very Large Data Bases*. San Mateo, CA, USA: Morgan Kaufmann, Sep. 1994, pp. 487–499.
[158] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2000, pp. 1–12.
[159] R. K. Agrawal and R. Srikant, "Mining sequential patterns," in *Proc. 11th Int. Conf. Data Eng.*, Nov. 2002, pp. 3–14.
[160] M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Mach. Learn.*, vol. 42, nos. 1–2, pp. 31–60, 2001.
[161] *Artmine GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/MohammadRezaHeidariIman/ARTmine
[162] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A performance study of mining maximal frequent itemsets," Github Repository, Tech. Rep., 2003.
[163] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "FreeSpan: Frequent pattern-projected sequential pattern mining," in *Proc. 6th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, New York, NY, USA, Aug. 2000, pp. 355–359, doi: 10.1145/347090.347167.
[164] *Goldminer Bitbucket Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://bitbucket.org/debjitp/goldminer
[165] *Samples2ltl GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/ivan-gavran/samples2LT
[166] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. IT-28, no. 2, pp. 129–137, Mar. 1982.
[167] H. Riener, "Exact synthesis of LTL properties from traces," in *Proc. Forum Specification Design Lang. (FDL)*, Sep. 2019, pp. 1–6.
[168] A. Rivas and S. A. McIlraith, "Learning interpretable models expressed in linear temporal logic," in *Proc. Int. Conf. Automated Planning Scheduling (ICAPS)*, vol. 29, Jul. 2019, pp. 621–630.
[169] *Dialogflow*. Accessed: Feb. 18, 2025. [Online]. Available: https://dialogflow.cloud.google.com
[170] *Codet5 Large (Python Version)*. Accessed: Feb. 18, 2025. [Online]. Available: https://huggingface.co/Salesforce/codet5-large-ntp-py
[171] *Plbart Model Documentation*. Accessed: Feb. 18, 2025. [Online]. Available: https://huggingface.co/docs/transformers/en/modeldoc/plbart
[172] D. Harman, *Information Retrieval Evaluation*, 1st ed., San Rafael, CA, USA: Morgan & Claypool, 2011.
[173] D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory*, vol. 2, no. 2, pp. 127–145, Jun. 1968.
[174] B. P. Lowerre and B. R. Reddy, "Harpy, a connected speech recognition system," *J. Acoust. Soc. Amer.*, vol. 59, no. S1, p. S97, Apr. 1976, doi: 10.1121/1.2003013.
[175] M.-C. d. Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *Proc. 5th Int. Conf. Lang. Resour. Eval. (LREC)*, May 2006, pp. 449–454.
[176] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.
[177] (2025). *Open Verification Library (OVL)*. Accessed: Feb. 18, 2025. [Online]. Available: https://www.accellera.org/downloads/standards/ovl
[178] *Bayspec GitHub Repository*. Accessed: Feb. 18, 2025. [Online]. Available: https://github.com/arturmrowca/bayspec
[179] M. Law, A. Russo, and K. Broda, "Logic-based learning of answer set programs," in *Reasoning Web. Explainable Artificial Intelligence*. Cham, Switzerland: Springer, 2019.
[180] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surveys*, vol. 51, no. 3, pp. 1–39, May 2019.
[181] T. B. Le and D. Lo, "Beyond support and confidence: Exploring interestingness measures for rule-based specification mining," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 331–340.
[182] S. Aldegheri, N. Bombieri, S. Germiniani, F. Moschin, and G. Pravadelli, "A containerized ROS-compliant verification environment for robotic systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 222–225.
[183] H. Witharana, S. Sanjaya, and P. Mishra, "Dynamic refinement of hardware assertion checkers," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–6.
[184] T. Ghasempouri and G. Pravadelli, "On the estimation of assertion interestingness," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2015, pp. 325–330.
[185] E. M. Gold, "Complexity of automaton identification from given data," *Inf. Control*, vol. 37, no. 3, pp. 302–320, Jun. 1978. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995878905624
[186] N. Fijalkow and G. Lagarde, "The complexity of learning linear temporal formulas from examples," in *Proc. Mach. Learn. Res.*, Jan. 2021, pp. 237–250.

SAMUELE GERMINIANI (Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Italy, in 2023. He is currently a Research Consultant and a Contract Teacher with the Department of Engineering for Innovation Medicine, University of Verona. In 2024, he became a Researcher with the University of Guglielmo Marconi. His primary research interests include semi-formal verification and embedded security for cyber-physical systems.



DANIELE NICOLETTI received the master's degree in computer science from the University of Verona, Italy, in 2023, where he is currently pursuing the Ph.D. degree. His main research interests include system verification, specifically assertion mining techniques for cyber-physical systems.

GRAZIANO PRAVADELLI (Senior Member, IEEE) received the Ph.D. degree in computer science. He is currently the IFIP 10.5 WG Chair and a Full Professor of information processing systems with the Department of Engineering for Innovation Medicine, University of Verona, Italy. In 2007, he co-founded EDALab s.r.l., an SME working on the design of IoT-based monitoring systems. His main interests include system-level modeling, simulation and semi-formal verification of embedded and cyber-physical systems, and their application to human activity recognition and telemedicine.