A baseline framework for the qualification of LTL specification miners

Samuele Germiniani*†, Daniele Nicoletti‡, Graziano Pravadelli*
University of Verona, Department of Engineering for Innovation Medicine {*}
University of Verona, Department of Computer Science {‡}
University of Guglielmo Marconi {†}
Email: *samuele.germiniani@univr.it, †s.germiniani@unimarconi.it ‡daniele.nicoletti@univr.it, *graziano.pravadelli@univr.it,

***Abstract*—Over the past few decades, the verification community has developed several specification miners as an alternative to manual assertion definition. However, assessing their effectiveness remains a challenging task. Most studies evaluate these miners using predefined ranking metrics, which often fail to ensure the quality of the inferred specifications, especially when no fixed ground truth exists and the relevance of the specifications varies depending on the use case. This paper presents a comprehensive framework aimed at facilitating the evaluation and comparison of Linear Temporal Logic (LTL) specification miners. Unlike traditional approaches, which struggle with subjective analyses and complex tool configurations, our framework provides a structured method for assessing and comparing the quality of specifications generated by multiple sources, using both semantic and syntactic techniques. To achieve this, the framework offers users an easy-to-extend environment for installing, configuring, and running third-party miners via Docker containers. Additionally, it supports the inclusion of new evaluation methods through a modular design. Miner comparison can be based either on user-defined designs or on synthetic benchmarks, which are automatically generated to serve as a non-subjective ground truth for the evaluation of the miners. We demonstrate the utility of our framework through comparative analyses with four well-known LTL miners, illustrating its ability to standardize and enhance the specification mining evaluation process.**

***Index Terms*—Linear Temporal Logic, Specification Mining, SVA Generation, Assertion Mining, Behavior Detection**

I. Introduction

Formal specifications play a crucial role in the verification process of electronic systems, increasing the guarantees of their correctness and reliability. To simplify and quicken the tedious and error-prone process of manually writing formal specifications, researchers have proposed the use of "specification miners", i.e., tools that automatically infer formal specifications. These are mined, either as automata or formulas often expressed by means of temporal logics, from the system implementation [1], its execution traces [2]–[4], or its informal requirements [5], [6], even with the use of LLM-based solutions [7], [8], to infer the relevant behaviors of a design, detecting bugs or vulnerabilities, and automatically generating regression tests [9]. However, measuring the quality of assertion miners is still challenging. Most authors evaluate their tools by using predefined ranking metrics and through comparison with other miners. The employed metrics are hardly a guarantee of the quality of the inferred specifications, especially when there is no fixed ground truth, and the interestingness of the inferred specifications usually changes depending on the mining use case. For example, frequent specifications outlining the most common behaviors of a system might not be interesting if the use case involves detecting vulnerabilities, which are usually rare behaviors. Furthermore, fairness, while comparing different miners, is extremely hard to achieve as these tools require different inputs and configurations, return outputs in several formats, and are evaluated on different benchmarks. When comparing miners, even the researcher's personal bias comes into play, worsening the odds of a fair comparison, as the configuration of the miners is usually adjusted to the personal preferences of the tester, who might even lack the necessary knowledge to make a tool perform optimally. To make matters worse, most available tools are just prototypes, making them hard to install and set up for comparison. As a result, even though the literature is full of specification miners, it is hard to determine if new methodologies improve the state of the art. To address the abovementioned issues, we propose a framework to evaluate Linear Temporal Logic (LTL) specification miners presenting the following characteristics:

- It provides a ground truth through the automatic generation of benchmarks. In particular, the user can set the "mining difficulty" of the benchmarks in terms of what type and how many specifications should be mined, allowing to quickly determine the effectiveness and correctness of the target miners. User-defined benchmarks can be integrated and used too.
- It includes a novel method to determine the quality of the miner through syntactic, semantic, and coverage-based approaches.
- It exploits docker containers to provide a simple way to install and execute the target miners; the users can provide their own configurations to execute them. It is open-source and freely available at [10]. It already includes the orchestration of four well-known LTL specification miners, but given its modularity, the user can easily integrate further miners or directly provide specifications generated externally, for example, by an LLM-based tool, for their evaluation.

In the rest of the paper, Section II reports preliminary definitions, Section III describes the overall architecture of the framework, Sections IV and V provide details on its core aspects, Section VI show a use case, finally, Section VII, draws the conclusions.

II. Preliminaries

We focus hereafter on concepts useful for clearly understanding the rest of the paper: the definition of the logic used to formalize the specifications (output of the miners), its corresponding equivalent form in terms of automata (useful for their comparison), and the notion of trace (input for most of the miners).

Definition 1: The **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. Given a finite set of propositions P, the set of LTL formulas over P can be defined 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 ϕ_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 ϕ_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.

LTL is often extended with Sequential Extended Regular Expressions (SEREs), which are used to formalize sequences of events over time. This extension is commonly found in languages such as Property Specification Language (PSL) [11] and SystemVerilog Assertion (SVA) [12] with the reader referred to their respective standards for a comprehensive reference.

Definition 2: 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., by 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 specifications.

Definition 3: Given a finite sequence of time units $\langle t_1, ..., t_n \rangle$ and a set of variables $\{v^1, ..., v^m\}$ representing inputs and outputs of the Design Under Verification (DUV), an **execution trace** is a sequence of tuples $(t_i, v_i^1, ..., v_i^m)$ such that v_i^j is the value assumed by variable v^j at time t_i.

III. Framework Architecture

Our framework is designed to offer an objective and efficient assessment of LTL specification miners across a range of scenarios. Its architecture is shown in Fig. 1. The framework's input is a *test suite*, i.e., a list of tests to be executed. Each test T_i is a tuple $\langle MI, \{C^{m_1}, C^{m_2}, \ldots, C^{m_n}\}\rangle$, where MI denotes the *mining input*, and C^{m_j} is the configuration to be used for miner m_j in test T_i. Each test must have a configuration for each miner under evaluation. The *mining input* MI is the data being fed to each of the considered miners. It consists of a set of execution traces (see def. 3) and/or the DUV model. It can be either user-defined or automatically generated by our framework as reported in Section IV. The output of the framework consists of detailed reports that evaluate each miner against the mining inputs and miner configurations. These reports enable users to determine (i) if a miner works correctly (for debugging purposes), (ii) how well a miner works in isolation against a fixed baseline, and (iii) how each miner compares with the other being evaluated.

For each couple $\langle MI, C^{m_i}\rangle$ the evaluation process executes according to the three main sequential steps described below.

1) **Miner execution**. Miners are executed within Docker containers to provide a consistent and controlled environment. The framework utilizes shared Docker volumes to efficiently make the mining input available to each miner and to gather their outputs. Input and output adaptors play a critical role in managing the diversity of data formats and structures encountered in mining inputs and outputs. The input adaptor standardizes various trace formats (usually CSV or VCD) into a custom format accepted by a target miner. Conversely, the output adaptor converts the specifications generated (sometimes in a non-standard LTL format) by the miners to a standardized format accepted by our framework. At the time of writing, the framework supports the PSL [11], SystemVerilog Assertions [12], and SpotLTL [13] format. Every functionality of the framework is modular with good isolation, heavily simplifying its extendability and maintainability. In particular, adding a new miner requires only introducing new adaptors (if those already included are not suited) and performing its containerization.
2) **Miner evaluation**. This step involves evaluating the specifications generated by the miner following three main approaches:
   - comparison of the mined specifications against a set of golden specifications through semantic and syntactic methods;
   - computation of the fault coverage to determine how well the mined specifications are capable of identifying design regressions;
   - computation of performance metrics regarding how much resources are required by the miner to generate the specifications; this usually involves measuring how long it takes to complete the execution.
3) **Report generation**. The final phase involves organizing the results of the evaluation into human-readable reports. In particular, the framework generates a specific report for each test in the test suite and a general report comparing the miners to each other.

In the following sections, we provide a detailed description of the main components of the framework: the generation of mining input and the miner evaluation, while details on the

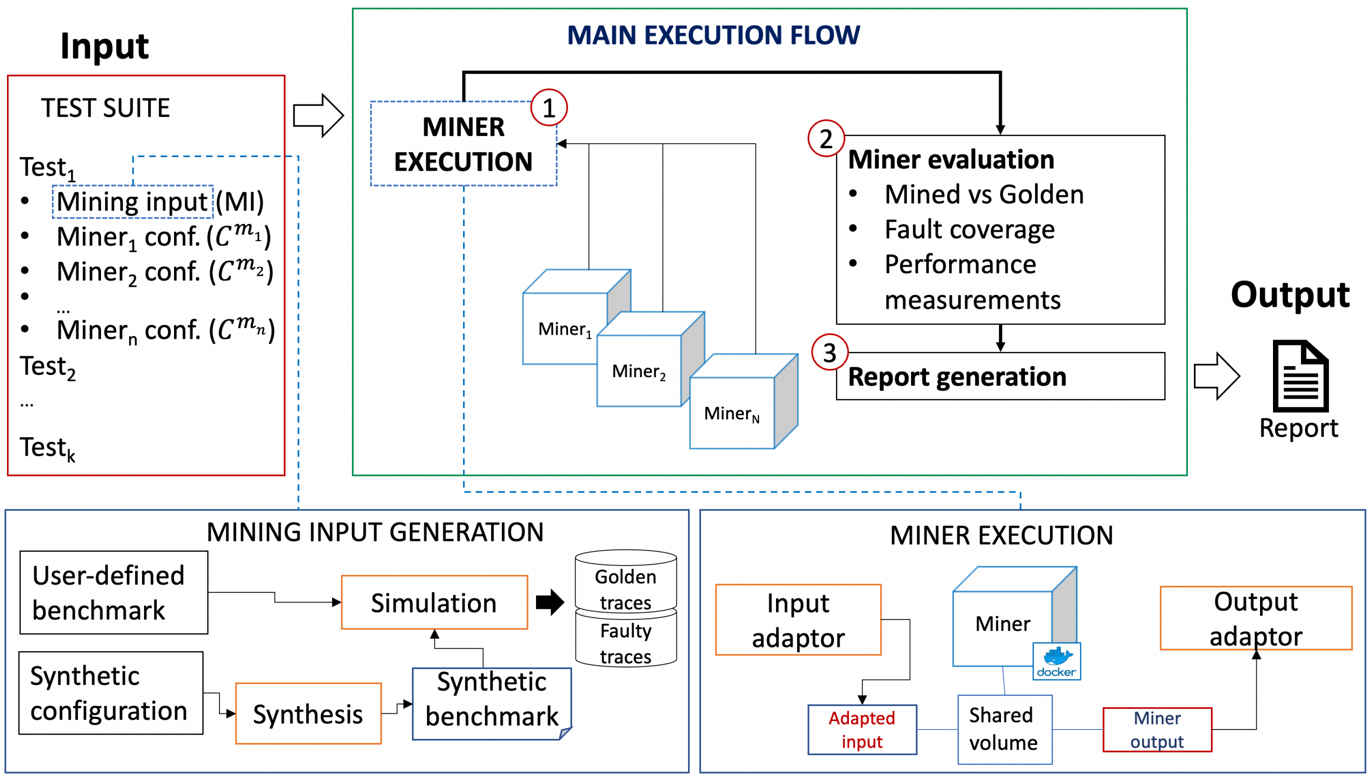


Figure 1: Framework architecture

containerized miner execution are omitted for lack of space and included in the tool manual [10].

IV. Mining input generation

This aspect of the framework (bottom-left box in Fig. 1) focuses on generating mining inputs by starting from a DUV and optionally a testbench. The framework employs a combination of synthetic and user-defined designs to ensure a comprehensive evaluation of specification miners. Synthetic designs, systematically generated from formal specifications, offer controlled and reproducible testing conditions. In contrast, user-defined designs, derived from real-world use cases, provide practical complexity and relevance, allowing for the assessment of miners in realistic scenarios. The framework simulates a candidate design to generate golden execution traces, which are used in the mining process. After that, the framework automatically injects faults in the form of mutants into the design to produce a set of faulty traces. These traces are used in the evaluation step to calculate the fault coverage of the mined specifications. We leveraged HIFSuite [14] to translate the Verilog design into an optimized-for-simulation C++ implementation capable of quickly executing hundreds of runs. At the same time, we employed the Muffin tool [15] to automatically inject faults into the HIFSuite design.

The framework supports the on-the-fly generation of synthetic Register Transfer Level (RTL) designs from a set of golden LTL specifications, implementing the "expected behaviors". These benchmarks are used to evaluate the miners' capability to capture predefined behaviors. The framework can generate synthetic benchmarks of increasing difficulty playing on the complexity of the corresponding golden specifications, i.e., changing the number of variables and the kind of LTL operators. The generation process is structured into two main steps, as outlined below.

1) **LTL specification to reactive controller**: The process begins with converting LTL golden specifications into a reactive controller. This is achieved using the `ltlsynt` tool [16]. Reactive control synthesis is a method used to automatically develop controllers that ensure a system adheres to specified LTL properties even in dynamic environments. This synthesis approach models the system as a strategic game between two players: the controller and the environment. The controller's role is to make decisions to meet the LTL specifications, while the environment acts as the adversary, providing external inputs or disturbances that could potentially disrupt the system. The goal of this synthesis is to devise a strategy for the controller that invariably satisfies the LTL specifications under any possible influence from the environment. The result of the synthesis is a logic representation of the controller, specifically encapsulated in an AIGER file. An AIGER file is a compact ASCII or binary format that encodes a combinational or sequential, synchronous circuit using And-Inverter Graphs. It essentially represents the logic functions and state transitions of the controller in a format that can be efficiently processed by verification tools.
2) **Verilog translation**: This phase involves translating the circuit design encapsulated within the AIGER file into

a Verilog module. It is performed using `Yosys` [17], a well-known framework for hardware synthesis.

V. Miner evaluation

This step evaluates the effectiveness and efficiency of specification miners. It involves assessing the performance of the miners and also the accuracy and utility of the specifications they generate. This evaluation is performed by (i) comparing the golden against the mined specifications and (ii) computing the Specification Fault Coverage (SFC). In both cases, the evaluation produces two scores.

In the case of SFC, the evaluation produces a value between 0 and 1 representing the ratio of faults covered by the mined specifications with respect to the total number of faults, and a positive integer indicating the minimum number of mined specifications covering the faults.

In the case of the comparison between the golden and the mined specifications, the evaluation produces the *Match Score (MS)* and the *Redundancy Score (RS)*. The *MS* represents the average similarity between the golden and the mined set. Formally, given the set of golden specifications $G = \{gs_1, gs_2, \ldots, gs_n\}$ and the set of mined specifications $M = \{ms_1, ms_2, \ldots, ms_m\}$, *MS* is defined as the average of the best similarity scores for each golden specification:

$$MS = \frac{1}{n}\sum_{i=1}^{n} \max_j f(gs_i, ms_j) \tag{1}$$

where $f(gs_i, ms_j)$ is the *similarity score* using method f between the golden specification gs_i and the mined specification ms_j, and it ranges from 0 (no similarity) to 1 (perfect similarity). The function $\max_j$ selects the highest similarity score reached by any mined specification for a given golden specification. We propose three methods for computing f: syntactic, semantic, and hybrid, as detailed in Section V-1.

The second score, *RS*, measures the proportion of mined specifications that did not contribute to determining MS. RS is calculated as follows:

$$RS = \frac{|M| - |U|}{|M|} \tag{2}$$

where $|U|$ is the number of mined specifications that were used at least once in calculating the *MS*, and $|M|$ is the total number of mined specifications. This score ranges from 0 (no redundancy, all mined specifications are useful) to 1 (high redundancy, no mined specifications contribute to the *MS*). A high *RS* indicates a significant number of mined specifications were unnecessary, pointing to potential underfitting, where the algorithm produces many irrelevant specifications. The *RS* is critical because it highlights one of the most common issues in the specification mining process.

1) Comparison between expected and mined specifications

This section is devoted to describing how the *similarity score* f of equation (1) is computed by adopting three different methods: syntactic, semantic, and hybrid, each addressing different facets of specification equivalence. Despite the method adopted to compute f, two essential preprocessing steps are initially undertaken:

1) **Parsing:** Specifications are parsed into syntax trees. This process is fundamental as it breaks down the specifications into their constituent syntactic elements, removing trivial syntactical differences (e.g., different versions of the same operators such as “G” and “always” or “X” and “nexttime”). Syntax trees help identify and organize the specifications’ components, such as operators and variables, which are crucial for both syntactic and semantic analyses.
2) **Simplification:** Subsequently, all variables and non-boolean expressions are transformed into boolean labels. This step is necessary for the following reasons:
   a) It aligns with the limitations and capabilities of most semantic equivalence checkers, which typically support only boolean expressions or a limited subset of non-boolean operators. Our framework extends this by supporting a wide array of operators found in programming languages like C/C++ and Verilog.
   b) It eliminates syntactic discrepancies that could incorrectly suggest differences between specifications that are semantically equivalent. For example, varying syntactic representations of the same logical expression could otherwise lead to false negatives in comparison results.

To better understand the role of parsing and simplification, consider, for example, the following specifications:
$G(a + 1 > 10 \;\&\&\; Xb \to X[2](c \neq true))$
$always(a > 9 \;\&\&\; nexttime(b) \to X[2](!c))$
Initially, a syntactic comparison would flag these as different. However, after parsing, “G” becomes equivalent to “always” and “X” becomes equivalent to “nexttime”; then, after simplification, a sat solver can quickly determine that $a + 1 > 10$ is semantically equivalent to $a > 9$, and $c \neq true$ is equivalent to $!c$. Both specifications are simplified to the same boolean identifiers producing the same expression $G(blabel0 \;\&\&\; X(blabel1) \to X[2](blabel2))$, where $blabel0$ is the identifier for $a > 9$ and $a + 1 > 10$, $blabel1$ is the identifier for b, and $blabel2$ is the identifier for $!c$ and $c \neq true$. After that, even syntactic methods would recognize them as equivalent.

After these preprocessing steps, the similarity score is computed in one of the following three methods.

Syntactic comparison. We perform the syntactic comparison by computing the Levenshtein distance between two specifications; it measures the minimum number of operations required to transform one specification into another. We employ the opal library [18], which computes the Levenshtein distance through global pairwise sequence alignment [19], finding the minimum penalty of aligning two sequences. The alignment cost is normalized by the length of the longest specification to produce a score between 0 and 1.

Different from the classical approach of comparing individual characters, our method treats each variable, operator, and parenthesis as a single, indivisible element. This adjustment prevents the algorithm from inaccurately identifying two specifications as similar when they only share minor character sequences. For instance, without this modification, the specification $G(a \;\&\&\; b)$ could incorrectly appear similar to (G_f && c) merely because the variable G_f begins with the character "G". Our approach ensures that the comparison maintains the integrity of the expressions, avoiding coincidental character similarities.

For example, given $s1 : b\_0$ && $Xb\_1 \rightarrow Xb\_2$ and $s2 : \{b\_0$ ##2 $b\_1\}| \rightarrow b\_2$, the following best alignment is produced, where each $\wedge$ indicates an alignment penalty of 1.

```
  b_0  &&   X b_1    ->   X    b_2
{ b_0  ##2    b_1  } |->       b_2
^      ^    ^        ^  ^    ^
```

Consequently, the normalized syntactic similarity score for $s1$ and $s2$ is $1 - 6/7 = 0.142$, where 6 is the penalty corresponding to the 4 gaps and 2 mismatches, and 7 is the maximum possible penalty for this pair.

Semantic comparison. The semantic comparison is performed by comparing the languages of the specifications. We do that by employing the spotLTL library [13]. Given two LTL specifications $s1$ and $s2$, we consider three cases:

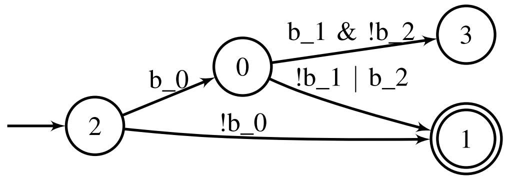
- $L(s1) = L(s2)$, where $L(s_i)$ is the language of specification s_i, then we consider the two specifications equivalent and we assign a score of 1;
- $L(s1) \subset L(s2)$ or $L(s2) \subset L(s1)$, then the specifications are considered partially equivalent (one covers/implies the other) and we assign a score of 0.5;
- else, we consider $s1$ and $s2$ not semantically related and we assign a score of 0.

For specifications $s1$ and $s2$ reported above, the semantic similarity score is then 0.

Hybrid comparison. The hybrid comparison combines syntactic and semantic elements to provide a more comprehensive evaluation of the specifications. The need for this hybrid approach arises from the limitations of the above syntactic and semantic methods. While the syntactic comparison is effective at identifying structural similarities, it is unable to consider semantic nuances. Conversely, the semantic comparison, although proficient in assessing semantic equivalence, offers limited granularity with only three possible scores: 1, 0.5, and 0. The hybrid comparison leverages the strengths of both methods to provide a more nuanced evaluation of the specifications. The hybrid approach transforms each specification to an equivalent Büchi automaton (describing the semantics of the specification) and performs the Graph Edit Distance (GED) between the two automata. The GED is a measure of the minimum number of operations required to transform one graph into another. The operations include node/edge insertion, deletion, and substitution. The GED is then normalized by the number of nodes+edges in the largest automaton to produce a score between 0 and 1. We calculate the GED using the GEDlib library [20] and using the optimal A* algorithm to find the optimal GED.

For example, consider the Büchi automata of specifications $s1$ and $s2$ reported in Fig. 2. Remapping the automaton of $s1$ to $s2$ requires four operations of the same cost: 1) Removing edge $2 \rightarrow 0$, 2) adding a new node N_1 (corresponding to node 1 in the automaton of s2), 3) Adding the new edge $2 \rightarrow N_1$ with label b_0, 4) adding the new edge $N_1 \rightarrow 0$. The similarity is then calculated as $1 - (4/10)$, where 4 is the penalty cost

Büchi automaton of $s1 : b\_0$ && $Xb\_1 \;\rightarrow\; Xb\_2$



Büchi automaton of $s2 : \{b\_0$ ##2 $b\_1\}| \rightarrow b\_2$

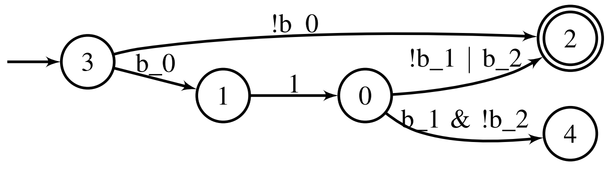


Figure 2: Büchi automata of speficications $s1$ and $s2$.

of the edits and 10 is the maximum number of possible edits for this pair of specifications (edge+nodes of $s2$ automaton).

We can observe that while semantic equivalence reported a similarity score of 0 and the syntactic similarity returned an underwhelmingly low score of 0.142 for two pretty similar specifications, the edit distance managed to mitigate the evaluation, providing a more realistic 0.6, which, in this instance, is more aligned with the user expectation.

2) *Specification fault coverage*

The SFC evaluation measures how effectively the mined specifications can detect known design regressions, which are critical for ensuring the robustness of designs against known bugs [21]. The process involves mapping the specifications generated by the miners against a predefined set of fault scenarios for the target DUV. This mapping helps to identify which specifications are capable of detecting specific faults. In particular, a fault f is detected (covered) by a specification s if s fails in the presence of f. Thus, the SFC is calculated by dividing the number of faults detected by the target set of specifications by the total number of injected (detectable) faults.

This evaluation also allows measuring the minimum number of specifications in the set that are necessary to cover all detectable faults, which is essential for maintaining verification efficiency [22]. Ideally, fewer specifications are preferable as they simplify the verification process and reduce computational costs. This challenge is analogous to the NP-hard set cover problem. To address it, we employ a greedy algorithm [23], which approximates the smallest set of specifications that achieves full SFC. This method effectively balances the trade-off between the number of specifications and the comprehensiveness of the SFC.

VI. Use case

In this section, we report the results for a meaningful synthetic use case to highlight the effectiveness of our approach. We orchestrated four well-known and open-source LTL specification miners: HARM [4], [24], Texada [25], [26], Goldmine [1], [27] and sample2LTL [28], [29]. The experiments have been carried out on a 3.5 GHz AMD Ryzen

Miner	N. Mined Specifications	Syntactic		Semantic		Hybrid		Specification Fault Coverage (190 faults)		Time to mine
		MS	RS	MS	RS	MS	RS	SFC	Minimum cov. subset	
Texada	123204	0.44	0.99	0.70	0.99	0.86	0.99	0.87	17	48m 5s
sample2ltl	20	0.36	0.83	0.00	1.00	0.32	0.83	0.05	10	17m 23s
Goldmine	60	0.71	0.83	0.50	0.83	0.79	0.83	1.00	20	2m 17s
Harm (MS)	10	0.70	0.00	1.00	0.00	1.00	0.00	0.79	10	1.50s
Harm (SFC)	70	1.00	0.86	0.74	0.86	1.00	0.86	1.00	20	1.60s

Table I: Summary report of comparing the four miners on the synthetic benchmark

3950x processor equipped with 32 GB of RAM (3600 MHz) and running Ubuntu 22.04 LTS.

The main challenges to orchestrating the miners in our framework are reported below.

- **Implementing the input adaptors**: Goldmine and HARM already accept input traces in the form of VCD or CSV files; therefore, they require no extra work. On the other hand, Texada and sample2ltl require the implementation of adaptors to convert the traces from VCD to the custom format they expect. Furthermore, samples2ltl requires remapping the variables to a set of unique identifiers of the form $x1$, $x2$,..., xn where xi is a unique identifier for the variable i.
- **Implementing the output adaptors**: HARM already returns specifications in one of the standards recognized by our framework; Goldmine, Texada and sample2ltl required simple adaptors to substitute non-standard operators, for example, the $=$ for equivalence, instead of $==$, and the $[]$ for always, instead of G.
- **Miner configuration**: for each miner, we prepared a custom configuration file specifying the parameters to be used during the mining phase. To ensure fairness across the options offered by the miners, we proceeded as follows. We configure the miners to generate the specifications following the template $G(antecedent \rightarrow consequent)$ and to run in single-thread mode. For HARM, Goldmine and samples2ltl, we set the temporal depth of the specifications to 3; for Texada, we enforced the same temporal depth by customizing the antecedent and consequent of the template. Since Goldmine restricts the antecedent to only include variables representing the inputs of the design and the consequent to only include the outputs (by analyzing the cone of influence of the design), we applied this constraint to HARM as well; however, Texada and sample2ltl do not support this configuration.

This use case was created using the automatic generation procedure explained in section IV. For this particular example, we generated a module starting from 10 specifications of the form $G(\{P0 \ \#\#1 \ P1\} \rightarrow \{\#\#1 \ P2 \ \&\& \ P3\})$ The procedure generated the Verilog design, the traces, and the corresponding golden specifications. We show in Table I the results of the mining evaluation according to the methods presented in Section V. It shows the number of mined specifications, the metrics used to evaluate the miners, namely the *Match Score* (MS), the *Redundancy Score* (RS), the *Specification Fault Coverage* (SFC), the minimum number of specifications that cover all the faults (*Minimum coverage subset*), and the execution time of each miner (*Time to mine*). The table is also organized as a heatmap, where the colors represent the performance of the miners in the different metrics. The colors range from green (best performance) to red (worst performance). At first glance, we can notice that "HARM (MS)" is the fastest and performs best in all MS and RS metrics except for the SFC. This is not surprising since, in this case, HARM was configured to produce as few redundancies as possible with the downside of missing some faults. When HARM is instead configured differently ("HARM SFC"), it can reach 100% of SFC, but with a higher redundancy (0.86), as reported in the last row. After that, we can observe how the introduction of different metrics allows the user to have a more comprehensive view of the miners' performance. In particular, the pure semantic comparison produces a simplistic value of 0 in the case of sample2ltl; however, the hybrid and syntactic metrics are capable of better pinpointing the miner's capabilities, producing a value of 0.36 and 0.32, respectively. Furthermore, RS allows the user to better interpret apparently good results in terms of MS but with high redundancy, making the miner less effective, as in the case of Texada that generated over 100k specifications to match just 10 specifications and covering 165 faults. It is evident that one configuration might not be able to optimize all use cases, corroborating the idea that for any given test, the input is far from being the only variable; use case and relative optimal configurations are as critical to ensure a fair and truthful comparison.

VII. Conclusions

In this paper, we introduced a framework to evaluate and compare LTL specification miners. Our approach leverages Docker containers to streamline the installation and execution of these miners, ensuring consistent evaluations across various configurations. We enhanced the evaluation process by using synthetic benchmarks as a reliable ground truth and incorporated semantic, syntactic and hybrid methods for a meaningful evaluation. Our comparative studies with four renowned LTL miners confirm the framework's efficacy in standardizing evaluations, thus advancing the understanding and application of specification mining.

References

[1] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," *Proc. Des. Autom. Test Eur. DATE*, 2010.

[2] G. Ammons, R. Bodík, and J. Larus, "Mining specifications," *Conf Rec Annu ACM Symp Princ Program Lang*, 2002.

[3] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci Comput Program*, 2007.

[4] S. Germiniani and G. Pravadelli, "Harm: A hint-based assertion miner," *IEEE Trans Comput Aided Des Integr Circuits Syst*, 2022.

[5] C. Harris and I. Harris, "Glast: Learning formal grammars to translate natural language specifications into hardware assertions," *Proc. Des., Autom. Test Eur. Conf. Exhib., DATE*, 2016.

[6] J. Zhao and I. Harris, "Automatic assertion generation from natural language specifications using subtree analysis," *Proc. Des., Autom. Test Europe Conf. Exhib., DATE*, 2019.

[7] F. Aditi and M. S. Hsiao, "Validatable generation of system verilog assertions from natural language specifications," *Proc. - Int. Conf. Transdiscipl. AI, TransAI*, 2023.

[8] O. Keszocze and I. Harris, "Chatbot-based assertion generation from natural language specifications," *Proc. Forum Specif. Des. Lang., FDL*, 2019.

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

[10] https://github.com/SamueleGerminiani/usm-t. Accessed: 15-01-2025.

[11] "Standard for property specification language (PSL)," *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pp. 1–184, 2012.

[12] "Ieee standard for systemverilog–unified hardware design, specification, and verification language - redline," *IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline*, pp. 1–1346, 2009.

[13] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko, "From Spot 2.0 to Spot 2.10: What's new?" in *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, ser. Lecture Notes in Computer Science, vol. 13372. Springer, Aug. 2022, pp. 174–187.

[14] N. Bombieri, G. Di Guglielmo, F. Michele, F. Fummi, G. Pravadelli, F. Stefanni, and V. Alessandro, "Hifsuite: Tools for hdl code conversion and manipulation," *EURASIP Journal on Embedded Systems*, vol. 2010, 01 2010.

[15] E. Fraccaroli, D. Quaglia, and F. Fummi, "Simulation-based holistic functional safety assessment for networked cyber-physical systems," in *2018 Forum on Specification & Design Languages (FDL)*, 2018, pp. 5–16.

[16] T. Michaud and M. Colange, "Reactive synthesis from ltl specification with spot," in *Proceedings Seventh Workshop on Synthesis, SYNT@CAV 2018*, ser. Electronic Proceedings in Theoretical Computer Science, vol. xx, 2018, p. xx.

[17] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:202611483

[18] https://github.com/Martinsos/opal. Accessed: 15-01-2025.

[19] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283670900574

[20] D. B. Blumenthal, S. Bougleux, J. Gamper, and L. Brun, "Gedlib: A c++ library for graph edit distance computation," in *Graph-Based Representations in Pattern Recognition*, D. Conte, J.-Y. Ramel, and P. Foggia, Eds. Cham: Springer International Publishing, 2019, pp. 14–24.

[21] A. Fedeli, F. Fummi, and G. Pravadelli, "Properties incompleteness evaluation by functional verification," *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 528–544, 2007.

[22] S. Brait, F. Fummi, and G. Pravadelli, "On the use of a high-level fault model to analyze logical consequence of properties," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, 2005, pp. 221–230.

[23] https://paal.mimuw.edu.pl/docs/w_set_cover.html. Accessed: 14-1-2025.

[24] https://github.com/SamueleGerminiani/harm. Accessed: 25-9-2024.

[25] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining," *Proc. - IEEE/ACM Int. Conf. Autom. Softw. Eng., ASE*, 2016.

[26] https://github.com/ModelInference/texada. Accessed: 25-9-2024.

[27] https://bitbucket.org/debjitp/goldminer. Accessed: 25-9-2024.

[28] D. Neider and I. Gavran, "Learning linear temporal properties," *Proc. Conf. Formal Methods Comput.-Aided Des., FMCAD*, 2019.

[29] https://github.com/ivan-gavran/samples2LTL. Accessed: 25-9-2024.