# HARM: A Hint-Based Assertion Miner

Samuele Germiniani, *Member, IEEE*, and Graziano Pravadelli, *Senior Member, IEEE*

***Abstract*—This article presents HARM, a tool to generate linear temporal logic (LTL) assertions starting from a set of user-defined hints and the simulation traces of the design under verification (DUV). The tool is agnostic with respect to the design from which the trace was generated, thus the DUV source code is not necessary. The user-defined hints involve LTL templates, propositions, and ranking metrics that are exploited by the assertion miner to reduce the search space and improve the quality of the generated assertions. This way, the tool supports the work of the verification engineer by including his/her insights in the process of automatically generating assertions. The experimental results show real improvements with respect to the state-of-the-art in terms of assertion coverage and scalability.**

***Index Terms*—Assertion mining, assertion ranking, assertion-based verification (ABV), temporal assertions.**

## I. Introduction

ASSERTION-BASED verification (ABV) is a popular method to check the functional correctness of a design. Unfortunately, assertion definition is a time-consuming and error-prone task, which requires high expertise to reason in terms of logic formulas [1].

To overcome the severe limitations of manually defining assertions, starting with the pioneering work on specification mining proposed in [2], verification engineers have developed several assertion mining approaches to automatically extract assertions from the actual implementation of the DUV (see Section II).

There exist mainly two ways for automatically extracting assertions from the DUV implementation. A first possibility is to statically analyze the DUV source code searching for (and exploiting) cones of influence and control-flow graphs among variables. The alternative method consists of dynamically applying data mining techniques searching for association rules on a set of execution traces obtained by simulating the DUV.

While static mining pursues generalizations and abstractions, dynamic approaches can only extract *likely assertions*, i.e., formulas that represent solely the behaviors that are exposed in the considered execution traces. On the other hand, static methods have scalability issues while dynamic assertion mining is much more computationally efficient.

Even if the quality of likely assertions extracted by dynamic approaches strictly depends on the level of exhaustiveness exhibited by the considered execution traces (indeed, a common characteristic of any simulation-based method), dynamic mining has gained more and more consensus in the last decade, thanks to its higher scalability. In addition, it has been shown that dynamic approaches can be applied also when the source code of the DUV is not available and only a black-box implementation is at disposal (e.g., for checking the presence of malicious code on a third-party IP).

A popular way of extracting assertions from execution traces consists in using greedy-based heuristics with decision trees (DTs) and association rules to generate invariants that follow the template *always*(*antecedent* $\rightarrow$ *consequent*), where *antecedent* and *consequent* represent arbitrarily temporal behaviors. The heuristic way is fundamental to guarantee the scalability of the approach as checking long execution traces for every possible temporal behavior is not feasible for large designs. However, heuristic approaches can be unable of mining a satisfying set of assertions, as some interesting behaviors are not generated, while irrelevant assertions are extracted.

Thus, the majority of existing dynamic assertion miners exploit heuristic approaches to guarantee scalability at the cost of a poor set of generated assertions, while other tools sacrifice scalability to provide a more complete set of assertions.

An additional drawback of existing approaches, both static and dynamic, is related to the fact that they blindly extract hundreds of assertions without considering the designer's intent and the application domain. This negatively affects the post-mining analysis from the verification engineers, which lack an automatic way of ranking the mined assertions in terms of interestingness.

Overall, existing miners miss a method that allows the user to decide the desired tradeoff between scalability, completeness, and manual effort to be applied for each context of interest.

To handle the aforementioned tradeoff, in this article, we propose a new assertion miner, called HARM (Hint-Based AsseRtion Miner). Given a set of execution traces and a set of *hints*, HARM generates linear temporal logic (LTL) assertions in the form *always* (*antecedent* $\rightarrow$ *consequent*). The hints are represented by: 1) *template* formulas characterizing temporal behaviors of interest for the verification engineer; 2) *propositions* defining relations between variables that he/she wants to investigate; and 3) *metrics* to assess the interestingness of the generated assertions. In particular, templates are employed to model the temporal relations between the operands involved in the implication inside the "always" statement. They can be

Manuscript received 3 August 2022; accepted 3 August 2022. Date of current version 24 October 2022. This work was supported in part by the Italian Ministry of Education, Universities and Research (MIUR) through the Project Dipartimenti di Eccellenza 2018–2022. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2022 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding author: Samuele Germiniani.)*

The authors are with the Department of Computer Science, University of Verona, 37134 Verona, Italy (e-mail: samuele.germiniani@univr.it; graziano.pravadelli@univr.it).

Digital Object Identifier 10.1109/TCAD.2022.3197525

defined by using all LTL operators belonging to the property specification language (PSL). A proposition can be any kind of Boolean expression that can be constructed in C/C++ by connecting variables through Boolean, relational, and arithmetic operators. Metrics are indexes used to measure the quality of an assertion. The tool allows the definition of *contexts* to cluster and rank the generated assertions according to the given hints.

With respect to the existing approaches, the main contributions of this work are listed below.

1) A very fast miner engine (with respect to the state-of-the-art). HARM features a parallelized linear-time algorithm to evaluate an assertion and generate its contingency table, efficiently managing input traces that are millions of time units long.
2) An agnostic and general-purpose miner. HARM does not require the sources of the DUV as input. As a matter of fact, the tool does not even require the existence of the design implementation, but only its execution traces.
3) A customizable template-based miner. Most tools provide only one mining template, a few others allow the user to choose between a limited subset of templates. HARM provides *always*(*antecedent* $\rightarrow$ *consequent*) as a base template, the user is then allowed to customize the antecedent and consequent by using the full set of LTL temporal operators included in the PSL. While writing the right set of templates to mine high-quality assertions is not a trivial task, the users can start from a set of well-known generic templates, like those proposed in [3], and gradually refine their hints according to the effectiveness of mined assertions.
4) A generalized procedure to generate assertions through a template-based and entropy-based DT algorithm.
5) A context-based approach to single out interesting assertions according to the user requirements.

HARM is an open-source tool freely available at [4].

The remainder of this article is organized as follows. Section II reports the related work; Section III provides a set of preliminary definitions useful to understand the technical parts of the paper; Section IV describes the architecture of HARM; Sections V–VIII report a detailed description of each step of the methodology implemented in the tool; Section IX reports the experimental results; finally, in Section X we draw our conclusions.

## II. Related Work

Static and dynamic approaches have been proposed for assertion mining. The first focus on analyzing the source code of the DUV, while the second considers only the execution traces obtained by simulating the DUV by means of input stimuli. As static and dynamic analyses present complementary advantages and disadvantages concerning accuracy and scalability, some works employ mixed static and dynamic techniques.

Among the first works in the software domain, Lo and Maoz [5], [6] proposed scenario-based specification mining approaches where the source code is instrumented to mine linear sequence charts. However, these approaches are not aimed at discovering the complete behavior of the DUV, but only the collaboration among its components. Other works mine the specifications of the DUV in form of algebraic equation [7] or Hoare-style equations of pre and post-conditions [8], [9], but the temporal behaviors are not considered. The work in [10] is one of the first attempts of statically generating assertions by using a template-based technique. The employment of user-defined templates greatly increases the applicability of the approach as templates can be chosen according to the design's characteristics.

In the hardware domain, Hangal *et al.* [11] described IODINE, a tool to automatically extract likely design properties of hardware descriptions. It generates invariants by making hypotheses on one or more variables in the design and by analyzing its behavior over a set of inputs. Chen *et al.* [12] developed a new procedure called "Semantic Inference" with the specific goal of automatically translating the behavior of a cyber-physical system into a formal specification. Seshia *et al.* [13] proposed a gate-level approach that extracts assertions compliant with a predefined set of temporal templates. Vasudevan *et al.* [14], [15], [16] proposed Goldmine, a tool for extracting LTL assertions following the template $G\ (boolVar_1\ \&\ next[1](boolVar_2)\ \&\ \cdots\ \&\ next[N](boolVar_m) \rightarrow boolVar_k)$. It generates assertions by using static analysis and data mining. The authors have recently improved their ranking method by introducing complexity and importance metrics in [17]. Ghasempouri *et al.* [18] proposed a paper on the same topic where the interestingness of assertions is determined by using data mining metrics.

More recently, system-level approaches that work also on non-Boolean data types have been presented by Danese *et al.* [19], [20]. In [19], a time-window-based approach, focused only on DUV control signals, is proposed to extract assertions related to the I/O protocol. A more generic tool is instead described in [20], but mined assertions are restricted to a subset of predefined temporal patterns. The only two approaches that generate temporal assertions considering arithmetic/logic expressions among the variables of the DUV are ODEN [21] and the work described in [22]. In [23] a tool called A-TEAM is introduced for template-based assertion mining. A-TEAM employs the Apriori algorithm [24] to extract high-frequency atomic propositions. Then, the propositions are used to instantiate the templates through a set of justification rules. Finally, the generated assertions are qualified in terms of fault coverage.

Commercial tools are also available for automatic assertion generation at RTL, e.g., Atrenta BugScope [25] and Jasper ActiveProp [26]. The first generates SVA or PSL assertions where only the *next* temporal operator is considered. The second generates both structural and behavioral SVA *next*-based assertions.

The above tools are effective in automatically generating LTL assertions; however, they present several limitations which are solved by HARM.

1) All approaches employing static analysis techniques require the source code of the DUV, making them unsuitable in all verification flows in which the DUV is not available. Furthermore, they are dependent from the implementation language of the DUV, greatly reducing their applicability.
2) The basic template for HARM is $G$(*antecedent* $\rightarrow$ *consequent*), however, the antecedent and the

```
template : G(implication)

implication : tformula -> tformula
        | tformula => tformula
        | {sere} |-> tformula
        | {sere} |=> tformula

tformula: proposition | placeholder | ..&&..
    | (tformula) | !tformula | tformula && tformula
    | tformula || tformula | tformula xor tformula
    | tformula U tformula | tformula W tformula
    | tformula R tformula | tformula M tformula
    | X [N..(N)?] tformula | X tformula
    | F tformula | {sere}

sere : proposition | (!)? placeholder | ..&&..
     | ..##N.. | ..#N&..  | (sere) | {sere}
     | sere | sere | sere & sere | sere && sere
     | sere;sere | sere:sere | sere[*N(..N)?]
     | sere[*] | sere[+] | sere[=N(..N)?]
     | sere[->N(..N)?] | ##N sere | ##[N(..N)?] sere
     | sere ##N sere | sere ##[N(..N)?] sere

placeholder: 'P' N
N: numeric
```

Fig. 1. Template grammar adopted in HARM.

consequent can be customized by the user through the grammar shown in Fig. 1, thus allowing the specification of a wide set of different LTL formulas. On the contrary, existing tools work on a narrower set of predefined templates. For example, IODINE [11] and DAIKON [9] can extract only invariant formulas, then no temporal operators are allowed. Goldmine [14], [15], [16] extracts assertions of the form $G(antecedent \rightarrow consequent$ but it supports only the use of the temporal operator next (i.e., X). ODEN [21] implements only $G(a \rightarrow next(b))$, $G(a\ alternating\ b)$ and $G(a \rightarrow a\ U\ b)$, where $a$ and $b$ are simple propositions. A-TEAM [23] is the only other tool supporting a set of templates similar to HARM, but it only partially supports the DT operator defined in Definition 4 of Section III, as it does not allow conjunction of *next* operators with holes in the temporal extension.

3) Goldmine performs assertion ranking by analyzing the source code of the DUV; [18] proposes ranking metrics employing the contingency table of the miner; A-TEAM proposes a minimal fault-coverage ranking technique. Instead, HARM is more general and does not require the source code of the DUV. Furthermore, these tools do not provide a completely configurable context-based approach to allow the generation of an interesting set of assertions for every domain of application.
4) None of the above tools provides a configurable and generalized entropy-based DT procedure allowing the use of a variety of DT operators (Definition 4) that can be instantiated in a templated formula according to the user's requirements.

## III. Preliminaries

In this article, we make use of well-known LTL operators, such as Next (X), Until (U), Release (R) and Always (G). Due to lack of space, we kindly refer the reader to [27] for a complete description of the corresponding semantics. In the remaining part of this section, we report a list of definitions useful to understand the remainder of this article.

*Definition 1:* Given a finite sequence of time units $\langle t_1, \ldots, t_n \rangle$ and a set of variables $\{v^1, \ldots, v^m\}$, an **execution trace** 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 $t_i$.

For the sake of compactness, the term *trace* will be used instead of "execution trace" in the rest of the paper.

*Definition 2:* A **proposition** is a Boolean expression that can be constructed by using Boolean operators (&&, ||, !) between Boolean expressions, or relational operators (<, >, >=, <=, ==, ! =) between numeric expressions. Numeric expressions are constructed by using arithmetic operators (+, −, ∗, /) or bitwise operators (&, |, ∼, >>, <<). Boolean constants and DUV variables are propositions. Numeric constants and DUV variables are numeric expressions.

A proposition is labeled with one of the following tokens "a," "c," "ac" to specify that it appears in the assertion, respectively, into the antecedent, the consequent or both. For example, in Fig. 3, $P$ contains four "a" propositions ($v1$, $v2$, $v3$, $v4$) and two "c" propositions ($v5$, $v6$).

*Definition 3:* A **placeholder** is a Boolean variable that can be substituted by a proposition.

Similarly to propositions, we will refer to three kinds of placeholders: 1) those who appear only in the antecedent ($aP$); 2) only in the consequent ($cP$); or 3) in both the antecedent and consequent ($acP$). Placeholders of kind $aP$, $cP$ and $acP$ can only be substituted by propositions labeled with "a," "c," and "ac," respectively. In the rest of this article, placeholders are always indicated in the form $PN$, where $N$ is a positive integer number. For example, template $t_1$ of Fig. 3 contains two placeholed of kind $cP$ (P1, P2) and one of kind $aP$ (P0).

*Definition 4:* A *DT operator* is a special type of temporal (or propositional) operator that can be instantiated by using a DT algorithm.

Currently, HARM implements three DT operators as shown in Fig. 1.

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

A DT operator can only appear once in the antecedent of each template. This is necessary to preserve the scalability of the approach. An uninstantiated DT operator is equal to the *true* Boolean constant.

*Definition 5:* A **template** is a temporal expression constructed by connecting propositions, placeholders and DT operators through PSL temporal operators in the form $G$ ($antecedent \rightarrow consequent$). A template is potentially instantiated (PIT) if all placeholders are substituted with propositions or if it does not contain any placeholders. A PIT may contain uninstantiated DT operators.

Templates define the initial temporal behavior (the temporal behavior can change while instantiating a DT operator) of the mined assertions. The grammar for the templates supported in HARM is reported in Fig. 1. It is a subset of the grammar

of the popular framework spotLTL [28]. The grammar also supports sequential extended regular expressions (SEREs).

*Definition 6:* An **assertion** is a PIT where all DT operators are instantiated with propositions or where there is no DT operator.

*Definition 7:* Given a trace $tr$ of length $n$ and an assertion $as$, an **evaluation** of $as$ with respect to $tr$ is the sequence of evaluation units $\langle e_1, \ldots, e_n \rangle$, where $e_i$ is the truth value of the assertion at instant $t_i$.

The concept of evaluation for propositions and potentially instantiated templates are defined similarly, by substituting "assertion" with either "proposition" or "potentially instantiated template" in Definition 7. The evaluation unit $e_i$ may be *true*, *false* or *unknown*. It assumes the *unknown* value if it depends on at least one instant $t_j$ where $j$ is greater than the length of the trace.

*Definition 8:* An assertion, a proposition, or a potentially instantiated template **holds** in a trace if and only if its evaluation does not contain any evaluation unit whose value is false.

*Definition 9:* Given a trace $tr$, and an assertion $as$, a **contingency table** is a $3 \times 3$ matrix displaying the frequency distribution of *true*, *false* and *unknown* evaluations units of the antecedent with respect to the consequent of $as$ in $tr$.

*Definition 10:* A **metric** is a numeric formula measuring the impact of an assertion's feature in the assertion ranking.

The more prominent the feature, the higher its impact on the final ranking of the assertion. The elements of the contingency table are examples of features of an assertion. For example, in Fig. 3, $M$ contains three metrics. Metric $m_2$ (frequency) results in a value closer to 1 as the assertion gets a higher value of ATCT (number of times in which the antecedent implies the consequent on the input trace).

*Definition 11:* A **context** is a set of propositions, templates, and metrics.

The user can provide HARM with his/her own context to guide the mining.

## IV. HARM Architecture

HARM takes as inputs a trace (Definition 1), obtained by concatenating the set of execution traces of the DUV,[1] and a set of contexts $C = \{c_1, \ldots, c_n\}$ (Definition 11). Each context $c_i$ is a tuple $(P, T, M)$, where $P$ is a set of propositions predicating over the variables belonging to the trace, $T$ is a set of templates, and $M$ is a set of metrics. The output of the tool is a ranked set of assertions according to $M$. The architecture of HARM is shown in Fig. 2 and it is composed of the following three main steps, which are executed sequentially.

1) *Instantiation of Placeholders:* in the first step of the methodology the placeholders in the templates $T$ are substituted by using propositions belonging to $P$ to generate PITs (Definition 5).

[1]The generation of the execution traces is outside the scope of the paper. They can be generated by means of a user-defined testbench or an automatic test pattern generator. Its quality definitely affects the quality of the mined assertions, as it happens in any other simulation-based verification approach. It is reasonably to assume that in a simulation-based verification flow a high-quality test set is available at the time assertion mining is executed.

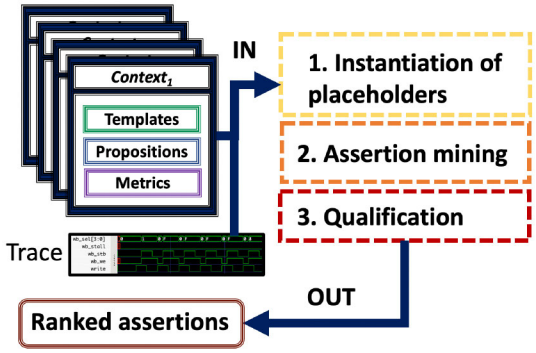


Fig. 2. Methodology overview.

*Trace*

| t | v1 | v2 | v3 | v4 | v5 | v6 | v7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 8 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 8 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 7 |
| 3 | 0 | 0 | 1 | 0 | 0 | 0 | 8 |
| 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 1 | 0 | 0 | 9 |
| 7 | 1 | 0 | 0 | 0 | 0 | 0 | 8 |
| 8 | 0 | 1 | 1 | 0 | 0 | 0 | 7 |
| 9 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 1 | 1 | 9 |
| 13 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 1 | 1 | 0 | 0 | 6 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |

*Context*

$P = \{(v_{1-4}, a), (v_{5-6}, c)\}$

$T = \{ \text{G}(\{ ..\#1\&.. ; P0\} \mid\rightarrow \text{X}(P1\,R\,P2)), \ t_1 \}$
$\{ \text{G}((P0 \,\&\&\, P1) \mid\mid (P2 \,\&\&\, P3) \rightarrow \text{X}(v7 > 5)), \ t_2 \}$

$M = \left\{ \left(causality, f, 1 - \frac{aftc}{traceLength}, 0.45\right), m_1 \right.$
$(frequency, s, \frac{atct}{traceLength}), \ m_2$
$\left. \left(reps, s, \frac{1}{pRepetitions*2+1}\right) \ m_3 \right\}$

Fig. 3. Running example.

2) *Assertion Mining:* in the second step, all PITs are used to generate assertions holding on the input trace. There are two scenarios: a) if a PIT does not contain a DT operator (Definition 4), then the tool directly generates an evaluation (Definition 7) for the PIT, and if the PIT holds (Definition 8) then it corresponds to a mined assertion and b) if a PIT contains a DT operator, then the tool invokes an entropy-based DT algorithm. The algorithm generates an assertion for each instantiation of the DT operator that makes the PIT hold on the trace.
3) *Qualification:* in the last step of the methodology, a context-based approach is applied to filter and rank the generated assertions according to their characteristics. This technique allows the user to single out the assertions that best fit all the features measured by the metrics of the considered context.

A detailed description of the methodology implemented in HARM is reported in the following sections. In addition, we give a practical demonstration of the various features of the tool by applying the methodology to a running example throughout the manuscript, whose inputs are shown in Fig. 3. In particular, the left side of Fig. 3 reports how the execution trace of the running example appears, while the right part describes the target context. According to Definition 1 the input trace is a sequence of values for the variables of the DUV at the varying of time. The time granularity depends on the DUV abstraction level. For example, at RTL the time granularity is based on the clock cycle, while at TLM it refers to the events associated with DUV transactions. Independently from this, the input trace can be informally seen as a matrix,

as shown in Fig. 3, where each row corresponds to a different time instant, and each column refers to a single variable of the DUV. From the operational point of view, HARM accepts the standard .vcd and .csv file formats to read the input trace. In addition, please note that, for the sake of simplicity, without lacking generality, in the running example, we consider only the propositions $vi$ = true (with $i \in [1, 6]$), and to simplify the writing we will refer to $vi$ = true with $vi$. Finally, $v1$, $v2$, $v3$, and $v4$ will be considered antecedent propositions, while $v5$ and $v6$ consequent propositions).

## V. Instantiation of Placeholders

In the first step of the methodology, we generate a set of PITs by instantiating the templates in $T$ with the propositions in $P$. For each template, HARM generates a set of proposition permutations with respect to the placeholders belonging to the template. Then, it substitutes each placeholder in the template with a proposition. Each permutation corresponds to a PIT. According to Definition 3, a consequent (antecedent) placeholder can be substituted only with a consequent (antecedent) proposition. For example, in Fig. 3, template $t_1$ has 2 placeholders in the consequent (P1, P2) that can be instantiated only with the consequent propositions $v5$ and $v6$.

To generate all the PITs of a template using the given set of propositions $P$ and the set of placeholders PH, we would have to generate $|P|^{|\text{PH}|}$ template instantiations, one for each permutation of propositions inside the placeholders. However, such a naive approach would not consider how templates are structured, forcing the miner to analyze several redundant permutations. For example, permutations generating the assertions $G((v1 \;\&\&\; v3) \;||\; (v2 \;\&\&\; v4) \rightarrow X(v7 > 5))$ and $G((v2 \;\&\&\; v4) \;||\; (v1 \;\&\&\; v3) \rightarrow X(v7 > 5))$ are equivalent because the || operator is commutative.

To solve the above issue, we have developed an algorithm that generates a reduced set of non redundant permutations.

The algorithm exploits the structural characteristics of the templates to avoid redundancy. In particular, we remove redundancy in two classes of operators: 1) commutative operators (COMs), such as && and || and 2) nonreflexive operators (NRs), which are binary operators of the form *left* $\Re$ *right*, where if *left* == *right* then *left* $\Re$ *right* is equivalent to *left* (or *right*). Examples of NR operators are the U (until) and R (release). Then, for COM operators the number of permutations generated by HARM is $\binom{\#\text{Propositions}}{\#\text{Operands}}$, while for NR operators they are #Propositions $*$ (#Propositions $-$ 1). For example, by considering the set of propositions $\{v0, v1, v2\}$ and the template $P0$ && $P1$, the resulting permutations are $\binom{3}{2} = 3$, i.e., $v0$ && $v1$, $v0$ && $v2$, $v1$ && $v2$, as && is a COM operator, while for the template $P0$ $UP1$ they would be $3 * (3 - 1) = 6$, as $U$ is an NR operator.

Algorithm 1 describes the GEN_PERMS function implemented in HARM to generate the reduced set of permutations according to the requirements mentioned above. The input of the function is the parameter $s$ of type FormulaStruct. This is a tree-like data structure generated from the abstract syntax tree of a template. Each node of $s$ is either an operator or a placeholder. The function considers three kinds of operators: 1) COM for commutative operators; 2) NR for NRs; and 3) $*$ for all the other types of PSL operators. Each node in $s$ is labeled

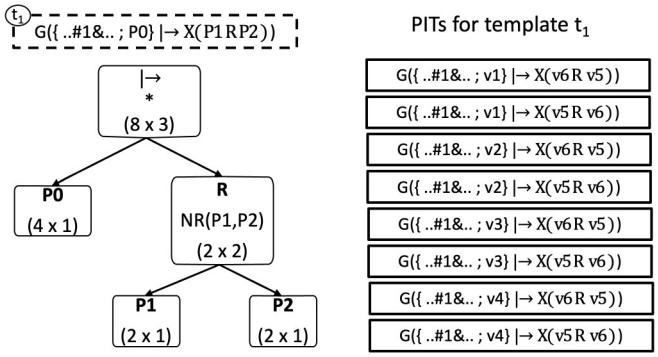


Fig. 4. Permutations in the running example.

**Algorithm 1** Generate the Reduced Set of Permutations

```
Input: the structure of the formula
Output: matrix of permutations
1:  function GEN_PERMS(FormulaStruct s)
2:      switch s.type do
3:          case PH
4:              return makeDomainMatrix(s.dim.nRows)
5:          case NR
6:              return makeNRMatrix(
7:                  GEN_PERMS(s.ch[0]),
8:                  GEN_PERMS(s.ch[1]))
9:          case COM
10:             return makeComMatrix(
11:                 GEN_PERMS(s.ch[0]))
12:         case *
13:             ret ← GEN_PERMS(s.ch[0])
14:             for i ← 1 to s.ch.size do
15:                 ret *= GEN_PERMS(s.ch[i])
16:             end for
17:             return ret
18: end function
```

with the dimension (row $\times$ col) of the corresponding permutation matrix, where row is the number of available permutations and col is the number of placeholders to be instantiated. For example, in the left side of Fig. 4, the FormulaStruct of template $t_1$ contains 5 nodes, i.e., $\mapsto$, $P0$, $P1$ R $P2$, $P1$, and $P2$. The node corresponding to the operator $\mapsto$ is labeled with $(8 \times 3)$ to indicate that the 3 placeholders $P0$, $P1$, and $P2$, included in the template, can be replaced by 8 permutations, obtained by combining the 4 permutations related to node $P0$ (which can be replaced by any of the 4 antecedent propositions $v1$, $v2$, $v3$, $v4$) and the 2 permutations related to node $P1$ R $P2$ (which can be replaced by either $v5$ R $v6$ or $v6$ R $v5$). The output of the function is a matrix representing the set of candidate permutations.

GEN_PERMS is a recursive function generating the permutations of a node by compounding the permutations of its children. There are four cases depending on the type of operator.

1) *PH:* this is the base case of the function where placeholders (i.e., the leaves of FormulaStruct) are handled. It returns an $N \times 1$ matrix using function makeDomainMatrix (line 4). The matrix enumerates from 0 to N-1 the propositions of the corresponding domain (a, c, ac).

2) *NR:* it returns the candidate permutations for the children (*ch*[0] and *ch*[1]) of an NR by using the function makeNRMatrix (line 6).
3) *COM:* it returns the candidate permutations of a COM by using function makeComMatrix (line 10). Note that makeComMatrix requires as input only the permutations of the first child *ch*[0] of the operator, as all the other children must yield the same permutations by construction.
4) **:* it returns the combination of the children's permutations (lines 12–16).

The right side of Fig. 4 shows the full list of PITs for template $t_1$, obtained by instantiating the permutations extracted by GEN_PERMS. Through this algorithm, we avoid generating, for example, the useless permutations $v5$ R $v5$ and $v6$ R $v6$ for the consequent of $t_1$. The effectiveness of the approach is more evident if we consider the template $t_2$ of Fig. 3: in this case HARM generates only 15 nonredundant candidates instead of the full set of 256 permutations. The time complexity of this algorithm depends on the type of operators and the number of placeholders and propositions involved; in the worst case, where all permutations are generated through a "*" operator, the worst time complexity is $O(|\text{PH}|^P)$, given set of propositions $P$ and the set of placeholders PH.

## VI. EVALUATION FUNCTION

In this section, we describe how HARM generates an evaluation for propositions, PITS, and assertions (Definition 7). This is a necessary step to implement the assertion mining procedure described in Section VII.

To generate an evaluation we apply an evaluation function to the input trace. The evaluation function for a proposition is trivial as its truth values depend only on a single time unit. For example, the evaluation function for proposition $v1$ && $v2$ is a function returning true if both $v1$ and $v2$ hold on a certain time unit, false otherwise. However, a template includes also temporal operators. As a consequence, the corresponding evaluation function must usually consider several time units before returning a truth value. The evaluation function we implemented in HARM for templates is based on an automaton-based representation of the corresponding temporal formulas. Therefore, we employ the framework spotLTL to translate LTL templates to deterministic complete Büchi automata. The advantage of this approach is that automata can be optimized by using state-of-the-art minimization techniques, improving the performance of the whole evaluation process.

For each template, we generate two automata, one for the antecedent and one for the consequent. This is necessary for two reasons; first, the generation of the contingency table (Definition 9) requires knowing the truth values of both the antecedent and the consequent; second, this schema speeds up the evaluation process in scenarios in which only the antecedent (or the consequent) needs to be re-evaluated, such as in our DT procedure.

An automaton is composed of states and edges; it always contains a *root* state, an accepting state, and a rejecting state; each edge is associated with a proposition, if a proposition $p$ associated with an edge $e_k(s_{src}, s_{dst})$ connecting state $s_{src}$ with state $s_{dst}$ is true, and $s_{src}$ is the current state, then $s_{dst}$ is the next state. Given an LTL formula $f$, the corresponding Büchi automaton *aut*, and a trace *tr*, a trivial way to determine the truth value of $f$ at time $i$ consists of the following steps.

1) Current state is equal to the root state of *aut*.
2) Find edge $e_k(s_{cur}, s_{\text{next}})$ whose proposition is true at time $i$, if $s_{\text{next}}$ is an accepting (rejecting) state, then $f$ is true (false) at time $i$ and the procedure ends, else go to the next step.
3) Current state is now equal to $s_{\text{next}}$, if $i+1$ is greater than the length of *tr* then $f$ is unknown at time $i$, else repeat step 2 at time $i+1$.

By repeating the previous procedure for all the time units of the trace, we obtain the evaluation of $f$ with respect to *tr*. However, in the worst-case scenario, each execution of the above procedure starting at time $i$ might have to consider all the subsequent time units $i+1$, $i+2$,..., till the end of the trace to return a truth value; therefore, this algorithm is quadratic with respect to the length of the trace, which may result definitely inefficient for very long traces.

In Algorithm 2, we propose a more efficient linear-time function to generate an evaluation that we implemented in HARM. The idea of the algorithm is to group up evaluation units requiring the same operations and to execute such operations only once for the entire group. The inputs of function EVALUATE are a Büchi automaton *aut* where all the placeholders have been substituted with propositions, and a trace *tr*. *curr* and *next* are two vectors of type $[(0, ddl_0), (1, ddl_1), \ldots, (n, ddl_n)]$, where $(i, ddl_i)$ is a couple whose first element $i \in 0, 1, \ldots, n$ identifies one state of the automaton, while the second element $ddl_i$ is a double linked list (DLL) of unsigned integers. DLLs are fundamental for keeping the algorithm linear, as they allow us to append a DLL to another DLL in constant time. Each element $(i, DLL)$ contains the list of time units assigned to state $i$; *curr* contains the time units evaluated in the current time frame (*time*), while *next* contains the ones that will be evaluated in the following time frame (time + 1). Both vectors are initialized with a list (containing *aut.nStates* elements, that is, one element for each state of the automaton) of empty DLLs (lines 3–4). The algorithm is then, composed of three main parts repeated for each time unit of the trace (line 6).

1) In the first part (line 8), the current time unit (*time*) is appended to the DLL containing the time units of the root state (*aut.root* is the index of the root state). This is the engine of the algorithm, that is, where new time units are added to the data flow.
2) In the second part (lines 10–26), the algorithm repeats the following procedure for each element of *curr* (for each state of the automaton) with at least one time unit in the DLL. The algorithm finds an outer edge in which the corresponding proposition is true on the trace for the current instant of time (lines 13–14). The expression *aut*[*si.first*].*outEdges* returns the outer edges of the state with ID equal to *si.first*, which is the first element of the couple $(i, ddl_i)$. If the selected edge reaches an accepting (rejecting) state, then the time units in *si.second* are used to generate true (false) evaluation units (lines 16–18 and 19–21), otherwise, they are appended to the DDL of the element of *next* (lines 22–24) corresponding to the

**Algorithm 2** Linear-Time Evaluation Function

```
1: function EVALUATE(aut, tr)
2:     Evaluation ret
3:     curr ← init(aut.nStates)
4:     next ← init(aut.nStates)
5:
6:     for time ← 0 to tr.length do
7:
8:         append(curr[aut.root].second, time)
9:
10:        for all si ∈ curr do
11:            if isEmpty(si.second) then
12:                continue
13:            end if
14:            for all edge ∈ aut[si.first].outEdges do
15:                if edge.evaluate(time, tr) then
16:                    switch aut[edge.dst].type do
17:                        case Accept
18:                            for all tu ∈ si.second do
19:                                ret[tu] ← true
20:                            end for
21:                        case Reject
22:                            for all tu ∈ si.second do
23:                                ret[tu] ← false
24:                            end for
25:                        case default
26:                            append(next[edge.dst].second,
27:                                   si.second)
28:                    break
29:                end if
30:            end for
31:            clear(si.second)
32:        end for
33:
34:        for all si ∈ next do
35:            append(curr[si.first].second,si.second)
36:            clear(si.second)
37:        end for
38:    end for
39:
40:    for all si ∈ next do
41:        for all tu ∈ si.second do
42:            ret[tu] ← unknown
43:        end for
44:    end for
45:    return ret
46: end function
```

destination state of the selected edge. The expression *ret*[*tu*] ← *true* stores in evaluation *ret* that the corresponding formula is true on trace *tr* at time *tu*. After that, the DDL of *si* is cleared, as all the time units were either moved to *next* or used to generate evaluation units.

3) In the third part (lines 28–30), all DLLs of *next* are moved to *curr*, precisely, the *i*th DLL of *next* is appended to the *i*th DLL of *curr*. This step is meant to load in *curr* the time units accumulated in *next*. This way, the algorithm is ready for the next time frame. The DDL of each *si* of *next* is cleared as all the time units were moved to *curr* (line 30).

Finally, the remaining time units, for which a truth value can not be inferred (the evaluation goes beyond the length of the trace), are used to generate unknown evaluation units (lines 32–34).

The worst-case time complexity of the algorithm is $(V + E) * \text{traceLength}$, where $V$ and $E$ are the numbers of states and edges of the automaton. However, since in all meaningful practical scenarios $((V + E)/\text{traceLength}) \approx 0$ (which means that *traceLength* is orders of magnitude larger than $V + E$), we can conclude that the time complexity of the algorithm is linear with respect to the length of the trace.

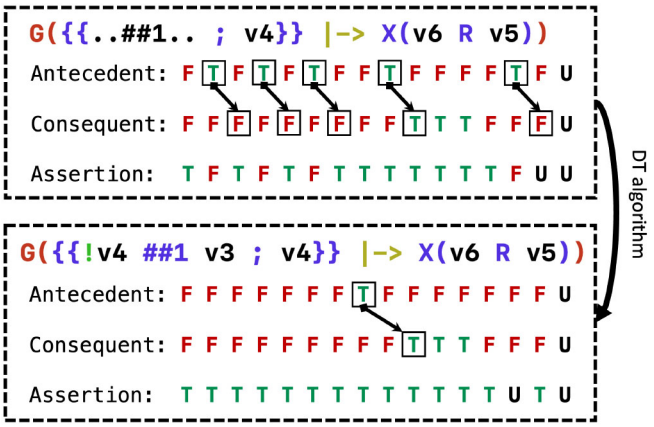


Fig. 5. DT narrowing.

## VII. Assertion Mining

In the second step of the methodology, we use PITs to generate assertions. The procedure differs depending on the content of the template.

If a PIT *pit* does not contain any DT operator (Definition 4) and the evaluation of *pit*, according to the EVALUATE function proposed in the previous section, does not contain any false values, then *pit* is an assertion that holds on the input trace.

If a PIT contains a DT operator, assertions are then mined through a DT algorithm as follows. As indicated after Definition 4, an uninstantiated DT operator is equivalent to *true*. Since DT operators appear only in the antecedent, an evaluation of a PIT including an uninstantiated DT operator usually contains several evaluation units in which the antecedent is true and the consequent is false. Then, instantiating a DT operator consists of narrowing the antecedent pool of true values to make the implication hold on the input trace. This is achieved by substituting the *true* constant with a more constrained expression by using the DT algorithm.

For example, in Fig. 5 we show a PIT containing an uninstantiated DT operator (..##1..) in the antecedent. The evaluation of the PIT (by considering the trace in Fig. 3) shows that there are four evaluation instants in which the antecedent is true and the consequent is false. By applying the DT algorithm, the *true* constant represented by the DT operator is replaced by !*v*4 ##1 *v*3 (!*v*4 and *v*3 are the operands of this expression) making the assertion hold on the trace. This concept can be applied to any LTL operator, as long as each time a new operand is added to the expression by the DT algorithm, the number of true values of the antecedent is reduced. For instance, the operator || would not be a suitable DT operator, because, being an "or" operator, each new operand would widen the number of true values of the antecedent. On the contrary, && (being an "and" operator) would work perfectly for the opposite reason. In HARM, we have generalized and formalized this idea by classifying the LTL operators complying with the above constraint into "Prop", "Temp," and "Mixed" DT operators.

1) *Prop* DT operators have only a propositional dimension; in our grammar we have ..&&.. $= \{o_1 \;\&\&\; o_2 \;\&\&\; \cdots \;\&\&\; o_n\}$ as a Prop DT operator. Each $o_i$ is a proposition.
2) *Temp* DT operators have only a temporal dimension; we have implemented ..##*N*.. $= \langle o_1 \#\#N \; o_2 \; \#\#N \; \cdots \; \#\#N \; o_n \rangle$ as a *Temp* DT operator. The user can define *N* to configure the temporal delay between $o_i$ and $o_{i+1}$.
3) *Mixed* DT operators have both the temporal and propositional dimension; we implemented ..#*N*&.. $= \langle (..\&\&..)_1 \; \#\#N \; (..\&\&..)_2 \; \#\#N \; \cdots \; \#\#N \; (..\&\&..)_n \rangle$ as a *Mixed* operator. This operator behaves like the sum of the propositional and the temporal dimension of the previous two DT operators.

Each DT operator is associated with a configuration (*Size*, *CType*, *Range*, *Offset*) involving several adjustable parameters.

1) *Size* is a tuple (*TempSize*, *PropSize*, *AllSize*) containing the maximum overall size *AllSize* (in terms of number of operands) of the generated expression, and the maximum number of *Temp* operands *TempSize* and Prop operands *PropSize*.
2) *CType* is a binary parameter stating if a DT operator with a temporal dimension must construct expressions following a sequential or a not ordered approach. To understand this, consider the DT operator ..##2.. with *TempSize* equal to 3; the resulting expression must follow the implicit template $o_1$ ##2 $o_2$ ##2 $o_3$; however, the order in which $o_1$, $o_2$, $o_3$ are substituted changes the outcome of the DT algorithm. A sequential DT operator substitutes the operands in order from $o_1$ to $o_3$ while a not ordered DT operator can substitute operands in any order. The first can only generate the expressions $o_1$, $o_1$ ##2 $o_2$, $o_1$ ##2 $o_2$ ##2 $o_3$, while the latter can generate expressions such as $o_1$ ##4 $o_3$ or ##4 $o_3$.
3) *Range* is a numeric parameter to adjust the number of candidates selected by the DT algorithm to split the search space.
4) *Offset* is a binary parameter stating if the algorithm must return the assertions belonging to the offset; such assertions are obtained by negating the consequent of an implication that is false each time the antecedent is true ($G(ant \rightarrow !con)$), making the implication always true on the trace.

As implied by the name, the DT algorithm employs a DT procedure where each decision consists of adding a new operand (proposition) to the DT expression. Operands are chosen according to their information gain (IG), that is, the expected reduction in information entropy caused by adding them to the DT 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 (onset) or always false (offset). The algorithm implemented in HARM always chooses the operand(s) that produces the highest reduction in entropy (the highest IG) in order to mine assertions by using as few operands as possible, avoiding over-constrained expressions. The computational cost of the DT algorithm is dependent of the number of propositions, the employed DT operators, the length of the trace, and the aforementioned parameters. The temporal complexity is summarized as follows. ..&&.. operator: $2^{\text{AllSize}} * \text{traceLength}$; ..##.. operator: $|P|^{\text{AllSize}} * \text{traceLength}$, where P is the number of "a" propositions in the context; ..#&.. operator: $2^{\text{TempSize}+\text{PropSize}} * \text{traceLength}$. Note that in any meaningful practical scenario, the worst-case scenario never occurs as the entropy-based heuristic only selects a small subset of the possible decisions for every level of the tree.

The main advantages of our approach with respect to earlier works are that:

1) we offer a variety of DT operators;
2) our DT algorithm is more configurable;
3) our DT operators are more flexible as they appear in a template-based context. For instance, the user can apply the DT algorithm by using a complex and partially instantiated template such as $G(\{\text{prop}_1[ = 1] \; \#\#7 \; \text{prop}_2 \; \#\#1 \; ..\&\&..)[*3]\} \rightarrow con)$.

### A. 3-Level Parallelization

The formalization of the mining problem adopted in HARM allowed us to heavily parallelize the generation of assertions. Particularly, we have implemented a 3-level parallelization algorithm capable of exploiting the additional cores of a CPU to speed up the computation. The idea behind this procedure is that the following are independent processes that can be parallelized.

*Level 1:* Generation of an evaluation by dividing the evaluation units among different threads. For example, in the running example one thread can handle evaluation units from $t_0$ to $t_7$ while another thread can operate from $t_8$ to $t_{15}$.

*Level 2:* Generation of assertions from different permutations of the same template.

*Level 3:* Generation of assertions from different templates.

The algorithm starts with an initial number of available threads. First, it divides them among the templates defined in the user context (level 3); the templates that received at least one thread are elaborated in parallel. After that, the distribution of threads continues for each template (level 2); a template with $n$ threads will generate assertions for $n$ permutations in parallel. If a template has more threads than permutations, the additional threads are distributed among the evaluation functions in each permutation (level 1). Each time the generation of assertions for a permutation or a template is concluded, the assigned threads are returned to the upper levels.

## VIII. Qualification

The last step of the methodology consists of filtering and measuring the quality of the generated assertions. Mined assertions are labeled with a ranking score by using the metrics provided by the user in his/her input context. After that, they are filtered and sorted in increasing order according to their scores. The employed metrics are user-defined numeric expressions; the user can define each metric by using various built-in assertion features. For example, in Fig. 3, the

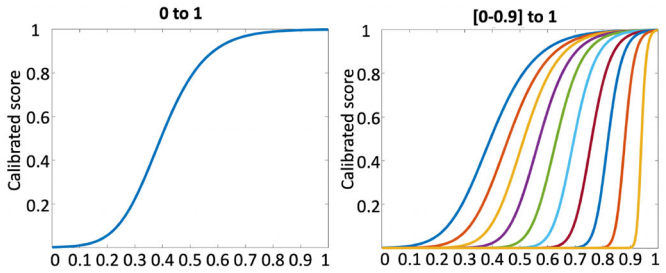


Fig. 6. Score functions.

metric $m_3$ combines arithmetic operators with *pRepetitions*, which is a built-in assertion feature measuring the number of repeated propositions. Through this mechanism, the user is free of defining the metrics by measuring the characteristics of an assertion he/she is interested in. In this process, metrics can be used either to filter or sort the assertions. The running example contains two sorting metrics ($m_2, m_3$) and one filtering metric ($m_1$). Filtering metrics are associated with a threshold; assertions with a score below the threshold of any filtering metric are directly discarded. In the running example, all assertions with a score of $m_1$ less than 0.45 are ignored.

Sorting metrics are used to perform the ranking. The ranking is computed according to an overall score. This is calculated, for each mined assertion $a$, through the following formula $\prod_{i=1}^{n} \text{calibrate}(sm_i(a)/sm_i(a_{\max\_i}))$, where $sm_i(a)$ is the score of $a$ by using the $i$th sorting metric, $a_{\max\_i}$ is the assertion that yields the maximum score by using metric $sm_i$, and *calibrate* is a procedure that "calibrates" the input score by using function $R = 1/(1 + e^{(z-kx)})^2$. In the chart on the left side of Fig. 6, we show the graphical representation of function $R$ with $z$ equal to 3.3 and $k$ equal to 10.62. This function is a modified version of the Richards' curve that ranges from 0 to 1. The intuition behind this ranking formula is that we want to allow the simultaneous employment of multiple sorting metrics in a single ranking procedure. Furthermore, we want to give more importance to assertions presenting a higher score for all sorting metrics, while penalizing assertions that score well only in a subset or in none of the given metrics. The *calibrate* function is capable of making lower scores greatly undermine the final ranking (1) while preventing high scores from compensating for lower scores (2).

The aforementioned values of $k$ and $z$ yield the standard *calibrate* function used in the miner; however, we allow the user to choose between 55 different configurations of $k$ and $z$, to adjust the effect of (1) and (2) on the final ranking. In the chart on the right side of Fig. 6, we show 10 configurations of the *calibrate* function, where the first function from the left returns values greater than 0 from 0 to 1, the second does the same thing from 0.1 to 1, the third from 0.2 to 1,..., the tenth from 0.9 to 1. Other configurations may include functions ranging from 0.3 to 0.8, from 0.5 to 0.6, etcetera. The computational complexity of performing the ranking is linear with respect to the number of ranked assertions.

Fig. 7 shows the final ranking of the assertions generated for the running example. Note that all assertions belonging to the offset of template $t_1$ were discarded by the filtering metric. As expected, assertions not presenting a good score for all

| N | Assertions (Context : default) | final | freq | pRep |
|---|---|---|---|---|
| 0 | G(((**v1** && **v2**) \|\| (**v3** && **v4**)) -> X(**v7** > **5**)) | 1.00 | 1.00 | 1.00 |
| 1 | G(((**v1** && **v3**) \|\| (**v2** && **v4**)) -> X(**v7** > **5**)) | 0.20 | 0.33 | 1.00 |
| 2 | G({{**v1** ##1 (**v2** && **v3**) ; **v4**}} \|-> X(**v6** R **v5**)) | 0.20 | 0.33 | 1.00 |
| 3 | G({{{(!**v3** && **v1**) ##1 **v3** ; **v4**}} \|-> X(**v6** R **v5**)) | 0.20 | 0.33 | 1.00 |
| 4 | G({{{!**v4** ##1 **v3** ; **v4**}} \|-> X(**v6** R **v5**)) | 0.20 | 0.33 | 1.00 |
| 5 | G(((**v1** && **v2**) \|\| (**v1** && **v3**)) -> X(**v7** > **5**)) | 0.19 | 0.67 | 0.33 |
| 6 | G(((**v1** && **v2**) \|\| (**v1** && **v4**)) -> X(**v7** > **5**)) | 0.19 | 0.67 | 0.33 |
| 7 | G(((**v1** && **v3**) \|\| (**v3** && **v4**)) -> X(**v7** > **5**)) | 0.19 | 0.67 | 0.33 |
| 8 | G(((**v1** && **v4**) \|\| (**v3** && **v4**)) -> X(**v7** > **5**)) | 0.19 | 0.67 | 0.33 |
| 9 | G(((**v2** && **v4**) \|\| (**v3** && **v4**)) -> X(**v7** > **5**)) | 0.19 | 0.67 | 0.33 |
| 10 | G(((**v1** && **v2**) \|\| (**v2** && **v4**)) -> X(**v7** > **5**)) | 0.04 | 0.33 | 0.33 |
| 11 | G(((**v1** && **v3**) \|\| (**v1** && **v4**)) -> X(**v7** > **5**)) | 0.04 | 0.33 | 0.33 |
| 12 | G(((**v1** && **v4**) \|\| (**v2** && **v4**)) -> X(**v7** > **5**)) | 0.04 | 0.33 | 0.33 |

Fig. 7. Running example final ranking.

sorting metrics have received a low ranking (assertions 1–12). Assertions 1–4 received the maximum ranking in one metric (pRepetitions), however, it was not enough to compensate for the low score in the other (frequency); furthermore, they received a final score very close to assertions 5–9 (0.20 vs. 0.19) even though having a far higher score with the frequency metric (1 vs. 0.67).

## IX. Experimental Results

Evaluating the effectiveness and efficiency of assertion miners is not a trivial task as the quality of the generated assertions is often a subjective matter; additionally, these tools are heavily influenced by their initial configuration, further complicating their overall evaluation. Nonetheless, there are objective measures we can exploit such as fault coverage and execution performances. Our experiments are divided into four parts; in the first and second parts we compare HARM against the well-known Goldmine and A-TEAM [23] miners, with respect to fault coverage and scalability; in the third part, we evaluate the behavior of HARM operating in a multithreading scenario; in the last part, we show the effectiveness of our context-based approach through a concrete use case.

The experiments have been carried out on a 3.5 GHz 16-core AMD Ryzen 3950× processor equipped with 32 GB of RAM (3600 MHz) and running Ubuntu 20.04 LTS.

### A. Fault Coverage

The first set of experiments compares the fault coverage achieved by assertions mined using HARM, Goldmine and A-TEAM on five RTL designs. These designs are used by the developers of Goldmine to showcase the tool. They are available at [29]. For each design, we have inserted a set of mutants and resimulated the mutated design by activating only one mutant for each simulation. This way, each generated faulty trace is affected at most by a single fault. Note that we have kept only the faulty traces in which the fault was observable, that is, in which the values on the outputs were different compared to the original unaltered trace. The considered mutants are: 1) "bit flip" for bit-vector variables, where a bit of a variable is negated and 2) "operator swap" for relational, arithmetic, Boolean, and bit-wise operators, where the

TABLE I
COMPARISON BETWEEN GOLDMINE, A-TEAM AND HARM: FAULT COVERAGE

| Design | Complexity | | | Assertions | | | Coverage | | | Min subset | | | AVG Coverage | | | Time to mine | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lines | I/O | Faults | H | G | A | H | G | A | H | G | A | H | G | A | H | G | A |
| (1) arb2 | 28 | 6 | 7 | 12 | 6 | 6 | 100% | 100% | 100% | 2 | 2 | 2 | 3.5 | 3.5 | 3.5 | 0.3s | 2.4s | 2s |
| (2) id_stage | 484 | 82 | 546 | 3160 | 2803 | 2405 | 82% | 41% | 68% | 182 | 102 | 201 | 2.5 | 2.2 | 1.8 | 8.3m | 23.5m | 13.4m |
| (3) decoder | 97 | 4 | 368 | 701 | 431 | 501 | 52% | 39% | 40% | 54 | 39 | 58 | 3.5 | 3.7 | 2.5 | 3s | 19s | 5s |
| (4) controller | 459 | 57 | 602 | 4909 | 1010 | 2079 | 84% | 48% | 56% | 132 | 79 | 93 | 3.8 | 3.7 | 3.6 | 5.8s | 12m | 17s |
| (5) multdiv | 230 | 15 | 1780 | 1722 | 682 | 1255 | 92% | 51% | 79% | 387 | 222 | 344 | 4.2 | 4.1 | 4.1 | 22s | 1.3m | 49s |

original operator is changed with a compatible one. For example, the original expression $v1$ && $v2$ is changed to $v1$ || $v2$ to generate a mutant.

We measure the fault coverage by counting how many faults are covered by the mined assertions (an assertion covers a fault if it fails on the corresponding faulty trace). To enable a fair evaluation of the two tools, we have run the experiments under the following constraints, which are due to the characteristics of Goldmine.

1) Goldmine can extract only assertions compliant with the template $G(\{..\#1\&..\}| \rightarrow P0)$. Thus, the same template has been provided to HARM and A-TEAM.
2) Goldmine does not support non-Boolean data types, thus only Boolean variables have been considered;
3) The maximum depth of the assertion has been set to 3 cycles, and the maximum number of propositions in the antecedent has been set to 5 for all tools.
4) The tools are all single-threaded.

Table I reports the results for the five benchmarks. The columns on the left of the table characterize the benchmarks by showing their names (*Design*), the number of code lines (*Lines*), the total number of primary inputs/outputs ($I/O$) and the number of injected observable faults (*Faults*). Then, the second part of the table shows the results achieved, respectively, by HARM (H), Goldmine (G) and A-TEAM (A), in terms of the number of generated assertions (*Assertions*); the percentage of faults covered by the mined assertions (*Coverage*); the minimum number of assertions covering the maximum number of faults (*Min subset*); the value ((N_faults * Coverage)/ Min_subset), higher is better, to describe how effective the mined assertions are at covering the faults (*AVG coverage*); the time spent by the tools for mining the assertions (*Time to mine*). For all benchmarks, traces composed of 1 000 clock cycles have been provided to the tools. Overall, HARM consistently managed to reach a higher fault-coverage for all benchmarks except for (1), where all tools achieved the same coverage. In terms of *AVG coverage*, HARM generally produced an equal or better-covering set. This suggests that HARM is generally able of generating a less constrained set of assertions with higher coverage. One noticeable difference is that HARM is faster than the other tools at mining the assertions.

## B. Scalability

In this section, we compare the scalability of the three tools with respect to the length of the input trace. The tests are executed under the same constraints defined in the previous section but with a progressively increased length of the input trace; in particular, we executed all five designs by using four traces that are 1 000, 10 000, 100 000, 1 000 000 clock cycles long. We have set 12 h (43200s) as the time limit. The chart in Fig. 8 reports the results of this evaluation. The $x$-axis reports the four lengths of the trace while the y-axis reports the time in seconds (logarithmic scale).

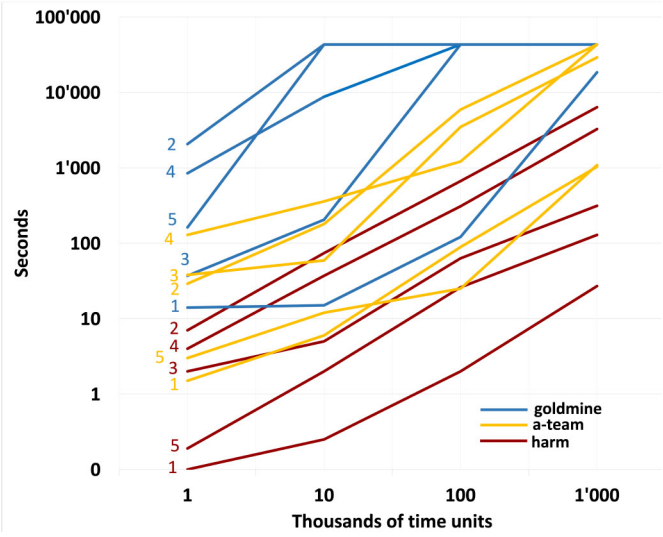


Fig. 8. Comparison between Goldmine, A-TEAM and HARM: scalability.

It is clear by analyzing the data that HARM is remarkably faster than the other tools. *arb*2 was the only design on which Goldmine terminated before the time limit by considering a 1 million clock cycle-long trace, taking over 5 h to complete, where HARM took only 34 s. A-TEAM appears to behave as a middle ground between the faster HARM and the slower Goldmine. Overall, HARM proves to be extremely scalable with respect to the length of the trace while Goldmine and A-TEAM look reasonably fast only for short traces.

We do not report results in terms of memory usage because they are dominated by the amount of memory required to hold the input trace; besides, the amount of memory required by HARM can not cause scalability issues as it is always a linear function of the input.

## C. Multithreading Evaluation

In this section, we analyze the speed-up guaranteed by applying the 3-level parallelization described in Section VII-A. Since this procedure has three dimensions, we have analysed the speed-up provided by each level separately.

In Fig. 9, we report the average results of parallelizing all the designs. The $x$-axis reports the number of threads available in each test while the $y$-axis reports the speed up of the test.

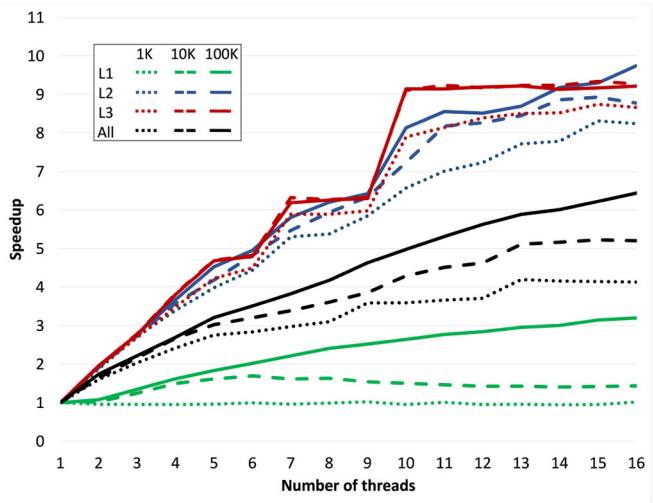


Fig. 9. Speedup of the 3-level parallelization.

The green, blue and red lines correspond to the results of applying levels one, two, and three, respectively. Since, the achieved speedup is also dependent on the length of the trace, we have included the results of using a 1 k, 10 k, and 100 k long traces, these correspond to the dotted, dashed, and solid lines, respectively. It is important to note that all speedups start deteriorating after 16 threads as all the tests have been executed on a 16-core machine.

For the first level (green lines), we have executed HARM with a single template producing a single permutation, making the evaluation function the only element benefiting from multithreading. The results at this level are heavily affected by the length of the trace: at 1 k (dotted line), we obtain a negative speed up ($< 1$) as the overhead of multithreading is not compensated by dividing the work among several threads; at 10 k (dashed line) the overhead starts to stabilize and we obtain a 1.5× speedup; at 100 k (solid line) the overhead completely stabilizes and we obtain a 3× speed-up with 16 threads. The results of applying the second and third levels are quite similar (blue and red lines) as their parallelization follows the same principle. In this case, we have executed the tests with a single template generating 16 permutations for level 2, and with 16 templates generating a single permutation for level 3. As shown in the chart, the best results are obtained at 100 k (blue and red solid lines) where we achieve a 9× speed-up with 16 threads. We also include the results of using the "fault-coverage configuration" (black lines) as a more practical scenario where we use all three levels together.

### D. Applying the Context-Based Approach

In this section, we show the effectiveness of applying our context-based ranking approach to a concrete use case. The use case consists of mining assertions for an RTL design containing a hardware trojan (HT). Our objective is to show that the assertions identifying the behavior of the HT are ranked higher than the other assertions if the user provided the correct metrics. To perform this test, we have selected the designs listed in Table II implementing the RSA encryption algorithm. These designs are available at [30]. According to the HT taxonomy, *rsa*100 and *rsa*300 contain a "leak-information" HT while *rsa*200 contains a "denial of service" HT; all three Trojans are activated through the user input. The trace used to mine the assertions is generated by simulating the design with a test bench performing thousands of encryptions; additionally, the generated trace contains at least one activation of the HT. To rank the generated assertions, we have provided the following sorting metrics.

1) *Causality:* $(1 - AFCT/traceLength)$.
2) *Frequency:* $(1 - ATCT/traceLength)$.
3) *Complexity:* $(complexity)$.

The first metric is to reward assertions representing a real correlation between antecedent and consequent, as assertions with a high *AFTC* in the contingency matrix, correspond to random correlations or partial behaviors. The second and third metrics reward assertions with low frequency and high complexity, respectively, where *complexity* is the number of variables in the assertion.

In this context, we assume that an assertion identifying an HT would have low frequency and high complexity as it shows a rare and hidden behavior with several constraints. We used template $G(..\&\&.. \rightarrow P0)$ for the first and second designs; template $G(..\#\#100..| \rightarrow P0)$ was employed for the third design. In the first and second designs, we mine the assertions by using 931 propositions, while in the third we have selected 212 propositions. In this scenario, we have a huge amount of propositions; this happens because all variables of bit-vector type are split into many single-bit variables, allowing the miner to generate complex expressions corresponding to bit configurations. HARM was capable of generating assertions catching the behavior of the injected HT in all three designs. Furthermore, as shown in Table II, such assertions are ranked in the first position by using our context-based approach, greatly simplifying the work of the verification engineer, who no longer has to read thousands of assertions to identify a single interesting assertion.

TABLE II
RESULTS OF APPLYING THE CONTEXT-BASED APPROACH

| Design | Trace length | N assertions | Position of intresting assertion | Time to mine |
|---|---|---|---|---|
| rsa100 | 834k | 689 | 1st | 19m |
| rsa200 | 1074k | 1440 | 1st | 116m |
| rsa300 | 840k | 227 | 1st | 30m |

## X. CONCLUSION

In this article, we presented HARM, an efficient and flexible hint-based assertion miner. Its main characteristics include a customizable template-based procedure to mine assertions, efficient algorithms for instantiating assertion templates and evaluating if they hold on the input trace, a 3-level parallelization methodology that further speeds up the mining by fully exploiting the computing cores, and a context-based approach to rank the mined assertions. The experimental results show the efficiency of the tool and the quality of the generated assertions in comparison with two state-of-the-art miners. The scalability and effectiveness of HARM have been thoroughly

analysed too. Finally, a concrete use case has been presented to highlight HARM's capability of ranking interesting assertions. The tool is open source and freely available at [4].

## REFERENCES


[1] C. Wang, F. He, X. Song, Y. Jiang, M. Gu, and J. Sun, "Assertion recommendation for formal program verification," in *Proc. IEEE COMPSAC*, 2017, pp. 154–159.

[2] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proc. ACM POPL*, 2002, pp. 4–16.

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

[4] "HARM source code." Accessed: Aug. 16, 2022. [Online]. Available: https://github.com/SamueleGerminiani/harm

[5] D. Lo and S. Maoz, "Specification mining of symbolic scenario-based models," in *Proc. ACM PASTE*, 2008, pp. 29–35.

[6] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *Proc. ACM KDD*, 2007, pp. 460–469.

[7] J. Henkel and A. Diwan, "Discovering algebraic specifications from Java classes," in *Proc. ECOOP*, 2003, pp. 431–456.

[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[9] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 35–45, 2007.

[10] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *Proc. IEEE ATS*, 2005, pp. 34–39.

[11] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proc. ACM/IEEE DAC*, 2005, pp. 775–778.

[12] G. Chen, M. Liu, and Z. Kong, "Semantic inference for Cyber-physical systems with signal temporal logic," in *Proc. IEEE CDC*, 2019, pp. 6269–6274.

[13] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. ACM/IEEE DAC*, 2010, pp. 755–760.

[14] L. Liu, C.-H. Lin, and S. Vasudevan, "Word level feature discovery to enhance quality of assertion mining," in *Proc. IEEE ICCAD*, 2012, pp. 210–217.

[15] 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. ACM/IEEE DATE*, 2010, pp. 626–629.

[16] S. Vasudevan, D. Sheridan, and V. Athavale, "Automatic generation of assertions from system level design using data mining," in *Proc. IEEE MEMOCODE*, 2011, pp. 191–200.

[17] D. Pal, S. Offenberger, and S. Vasudevan, "Assertion ranking using RTL source code analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 8, pp. 1711–1724, Aug. 2020.

[18] T. Ghasempouri and G. Pravadelli, "On the estimation of assertion interestingness," in *Proc. IFIP/IEEE VLSI-SoC*, 2015, pp. 325–330.

[19] 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*, Y. Shin, C. Y. Tsui, J.-J. Kim, K. Choi, and R. Reis, Eds. Cham, Switzerland: Springer, 2016, pp. 193–221.

[20] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Proc. ACM/IEEE DATE*, 2015, pp. 67–72.

[21] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli, "Dynamic property mining for embedded software," in *Proc. ACM/IEEE CODES+ISSS*, 2012, pp. 187–196.

[22] M. Bertasi, G. Di Guglielmo, and G. Pravadelli, "Automatic generation of compact formal properties for effective error detection," in *Proc. ACM/IEEE CODES+ISSS*, 2013, pp. 1–10.

[23] A. Danese, N. D. Riva, and G. Pravadelli, "A-TEAM: Automatic template-based assertion miner," in *Proc. ACM/IEEE DAC*, 2017, p. 37.

[24] R. Srikant and J. F. Naughton, "Fast algorithms for mining association rules and sequential patterns," Ph.D. dissertation, Dept. Comput. Sci., Univ. Wisconsin, Madison, WI, USA, 1996.

[25] "BugScope." Accessed: Aug. 16, 2022. [Online]. Available: http://www.atrenta.com/about-bugscope.htm5

[26] "Jasper Activeprop." Accessed: Aug. 16, 2022. [Online]. Available: http://www.jasper-da.com

[27] *Standard for Property Specification Language (PSL)*, IEEE Standard 1850–2010, pp. 1–184, 2012.

[28] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0–A framework for LTL and $\omega$-automata manipulation," in *Proc. ATVA*, vol. 9938, 2016, pp. 122–129.

[29] "Goldminer." Accessed: Aug. 16, 2022. [Online]. Available: https://bitbucket.org/debjitp/goldminer/src/master/example/

[30] "Hardware trojan benchmarks." Accessed: Aug. 16, 2022. [Online]. Available: https://trust-hub.org/#/benchmarks/chip-level-trojan


**Samuele Germiniani** (Member, IEEE) received the master's degree in computer science from the University of Verona, Verona, Italy, in 2019, where he is currently pursuing the Ph.D. degree.

His main research interests are related to embedded security, with emphasis on edge-cloud software verification.

**Graziano Pravadelli** (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2004.

He has been a Full Professor of Information Processing Systems with the Computer Science Department, University of Verona, since 2018. In 2007, he co-founded EDALab s.r.l., an SME working on the design of IoT-based monitoring systems. His main interests focus on system-level modeling, simulation and semi-formal verification of embedded systems, as well as on their application to develop virtual coaching platforms for people with special needs.

Prof. Pravadelli is an IFIP 10.5 WG Chair.