ARTmine: Automatic Association Rule Mining with Temporal Behavior for Hardware Verification

Mohammad Reza Heidari Iman, Gert Jervan and Tara Ghasempouri
Department of Computer Systems, Tallinn University of Technology, Tallinn, Estonia
{mohammadreza.heidari, gert.jervan, tara.ghasempouri}@taltech.ee

***Abstract*—Association rule mining is a promising data mining approach that aims to extract correlations and frequent patterns between items in a dataset. On the other hand, in the realm of assertion-based verification, automatic assertion mining has emerged as a prominent technique. Generally, to automatically mine the assertions to be used in the verification process, we need to find the frequent patterns and correlations between variables in the simulation trace of hardware designs. Existing association rule mining methods cannot capture temporal behaviors such as *next[N], until,* and *eventually* that hold significance within the context of assertion-based verification. In this paper, a novel association rule mining algorithm specifically designed for assertion mining is introduced to overcome this limit. This algorithm powers ARTmine, an assertion miner that leverages association rule mining and temporal behavior concepts. ARTmine outperforms other approaches by generating fewer assertions, achieving broader design behavior coverage in less time, and reducing verification costs.**

***Index Terms*—verification, assertion-based verification, automatic assertion mining, data mining, association rule mining**

I. Introduction

Functional verification ensures that a system's design meets its specification before manufacturing, by identifying and rectifying design errors [1]. Among the various methods to ensure system correctness, Assertion-Based Verification (ABV) has emerged as one of the most popular solutions for checking the design functionality [2].

Assertions are Boolean expressions that define the design's behavior [1]. Traditionally, verification engineers manually defined assertions [2]. However, the manual definition of assertions is a time-consuming task and requires human expertise and a profound understanding of the design's functionality [3]. Therefore, some studies have been conducted to automatically mine the assertions [4–9].

In the literature, there are several works on the automatic assertion mining of digital designs. The work in [4] proposed an assertion miner that employs dynamic dependency graphs to extract the design's signal relations and generate assertions. Another approach presented in [5] is a syntax-guided enumeration assertion miner. The proposed methods in [6] and [7] are techniques that extract assertions using several templates in the form of Finite State Machines (FSMs). HARM [10] and its extended version [11] are hint-based assertion miners that generate Linear Temporal Logic (LTL) assertions from simulation traces. The GoldMine introduced in [8] can generate assertions for a given Register-Transfer Level (RTL) design by leveraging formal verification and static code analysis. Finally, the method described in [12] extracts assertions by transforming sequential designs into pseudo-combinational designs.

This work was supported by the Estonian Research Council grants PSG837.

While existing assertion miners show promise, they do have some drawbacks that require improvement. Assertion miners in [4–6] suffer from redundancy (*i.e.,* assertions that describe the same design behavior) and inconsistency (*i.e.,* assertions that contradict each other), requiring additional tools like IMMizer [13] for resolution. Certain miners such as [5], [10], and [11] generate excessive assertions, necessitating tools like Dominance [14] and Shayan [15] for selecting the best assertions among all the generated ones. Readability issues arise with complex antecedents in miners like the work in [4]. Moreover, high execution times in the works such as [16] incur costs in the verification process.

On the other hand, association rule mining algorithms in the realm of data mining tend to neglect the incorporation of mining temporal information [17]. The primary focus of these algorithms such as FP-growth and Eclat lies in handling static data, which remains unchanged over time [18]. Nevertheless, they typically employ various data mining metrics to generate a limited yet accurate set of association rules [18]. These algorithms continue to demonstrate efficiency in swiftly extracting valuable information from big datasets within a short timeframe [18]. Furthermore, while some research has explored temporal association rule mining [19, 20], none of these approaches can capture temporal behaviors such as *next[N], until*, and *eventually* that are vital for ABV.

This paper introduces an innovative association rule mining algorithm to address the aforementioned drawbacks. This algorithm is capable of extracting rules that incorporate the concept of time, which is necessary in the context of ABV. Based on this algorithm, an automatic assertion miner called ARTmine is proposed to generate *next[N], until,* and *eventually* temporal behaviors from hardware designs. Remarkably, the data mining literature lacks dedicated algorithms for mining these temporal behaviors [20, 21], making the algorithm presented in this paper a unique and specific solution for assertion mining.

ARTmine efficiently produces readable assertions in significantly less time compared to other approaches. It generates a compact assertion set, minimizing redundancy and inconsistency while effectively covering the design behavior. As a result, ARTmine reduces verification costs and time.

The contributions of this paper are listed as follows:

- A new association rule mining algorithm is proposed to create an assertion miner called ARTmine, which can automatically generate the most important temporal patterns in ABV, *i.e., next[N], until*, and *eventually*, in a shorter amount of time in comparison with the proposed methods in the literature;
- ARTmine integrates suitable data mining metrics (*e.g.,* min_supp, and min_conf) that aid it in generating

979-8-3503-4859-0/DATE24/© 2024 EDAA

fewer assertions, prioritizing only the most accurate and valid ones;
- ARTmine achieves broader design behavior coverage than the state-of-the-art through an efficient algorithm capable of analyzing all relationships among all hardware design variables with minimal overhead.

The paper is organized as follows: The preliminaries are presented in section II and the proposed method is elaborated in section III. Section IV presents the experimental results and finally, Section V concludes the paper.

II. Preliminaries

In this section, we briefly explain the definitions and concepts used in this paper.

Definition 1: An ***atomic proposition*** is a logic formula that does not contain logical connectives [6]. Examples of *atomic propositions* are such as a1 = True, and b1 = False.

Definition 2: A ***proposition*** is a composition of *atomic propositions* through logical connectives [6]. An example for *proposition* is a1 = True && b1 = False && c1 = True.

Definition 3: An ***assertion*** is a composition of propositions through temporal operators that must hold or must become true during the execution of the design [6]. Typically, an assertion is divided into two parts: the left side, named *antecedent*, and the right side, called *consequent* [6]. The general structure of an assertion in Property Specification Language (PSL) is like $always(antecedent \rightarrow consequent)$, which implies that the consequent will hold whenever the antecedent occurs [22].

Definition 4: A ***simulation trace*** consists of the values of the variables of hardware designs that have been stored as records of data for different time instants (clock cycles) during the execution of the designs [10].

Definition 5: ***Temporal pattern next[N]***: In PSL, *next[N]* temporal pattern will be in the form of: $always(antecedent \rightarrow next[N]\ consequent)$ [22]. This means that when antecedent occurs, after N time instant (clock cycle), consequent will occur [22]. N is an integer value and N > 0.

Definition 6: ***Temporal pattern until***: In PSL, *until* temporal pattern will be in the form of: $always(antecedent\ until\ consequent)$ [22]. This means that the antecedent is true and holds up until the time that the consequent happens [22].

Definition 7: ***Temporal pattern eventually***: In PSL, this pattern is in the form of: $always(antecedent \rightarrow eventually!\ consequent)$ [22]. This means that there exists a future time instant (clock cycle) where the consequent of assertion finally holds [22].

Definition 8: ***Frequent itemsets*** refer to a set of variables in simulation trace that occur with a frequency, indicating significant relations/associations between the variables.

Definition 9: An ***Association Rule (AR)*** is defined as an implication of the form X → Y where X, Y ⊆ I, with X ∩ Y = Ø, and I is a set of items [18, 20, 21]. X and Y are called *frequent itemsets*.

Definition 10: ***Support*** is a metric in association rule mining that indicates how frequently an itemset appears in the dataset [18]. This value is between 0 and 1. For the rule X → Y, the value of support is calculated with the following formula [18, 23]:

$$Supp(X \rightarrow Y) = P(X \cup Y) \quad (1)$$

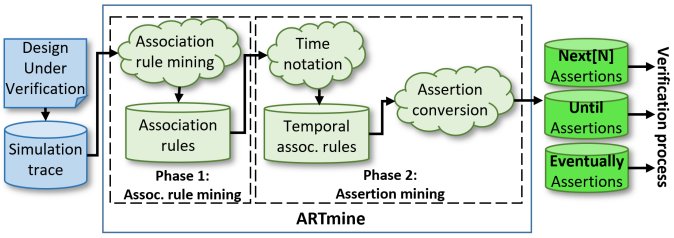

Figure 1: General flow of ARTmine

In (1), $P(X \cup Y)$ is the probability where $X \cup Y$ indicates that a record contains both X and Y, that is the union of itemsets X and Y.

Definition 11: The **min_supp** value is the threshold and a minimum value for support to decide whether an itemset is frequent (*i.e.*, occurs frequently in the simulation trace) or not [18]. If the frequency of the itemset is more than this threshold, the itemset is considered a frequent itemset [18]. A higher value of min_supp leads to generating commonly occurring (general) ARs, while a lower value of min_supp leads to generating rarely occurring ARs (corner cases) [18].

Definition 12: ***Confidence*** is an indication of how often the rule has been found to be true [24]. For the rule X → Y, this value is calculated with the following formula [18, 24]:

$$Conf(X \rightarrow Y) = P(Y|X) \quad (2)$$

It evaluates the degree of certainty of the detected association rule. This is taken to be the conditional probability $P(Y|X)$, that is the probability that a record containing X also contains Y. This value is between 0 and 1.

Definition 13: The **min_conf** is the minimum value for confidence [18]. The higher value of min_conf leads to fewer but more accurate and valid association rules [18].

III. Proposed Methodology

Fig. 1 depicts the general flow of ARTmine, taking a simulation trace (Definition 4) of the Design Under Verification (DUV) as input. The resulting output comprises a set of temporal assertions, encompassing *next[N]* (Definition 5), *until* (Definition 6), and *eventually* (Definition 7). These temporal assertions are then utilized in the verification process. As illustrated in Fig. 1, the first phase of ARTmine is *Association rule mining*. Consequently, phase 2 is *Assertion mining* which includes two sub-phases named *Time notation* and *Assertion conversion*.

During the *Association rule mining* phase, at first, the simulation trace of the hardware design undergoes preprocessing to prepare the data. Subsequently, a procedure is applied to the preprocessed data to mine all association rules (Definition 9) derived from the simulation trace.

In the second phase, *Assertion mining*, the association rules obtained from the first phase are passed to the *Time notation* step. Here, these extracted association rules are integrated with the concept of time to generate appropriate time-integrated rules (temporal association rules). These rules serve as the input for the *Assertion conversion* step. Consequently, the *Assertion conversion* step transforms the rules from the previous step into assertions, rendering them ready for utilization in the verification process. In the following subsections, each phase of the method has been discussed in more detail.

A. Association Rule Mining

The primary objective of this phase is to first preprocess the simulation trace and second, mine the association

rules from the preprocessed data. Conventional association rule mining algorithms (*e.g.*, Apriori, and FP-growth [18]) typically lack the capability to incorporate temporal considerations required for extracting crucial temporal patterns in ABV. To address this limitation and introduce a time-aware approach for association rule mining in ABV, we propose Algorithm 1. This algorithm outlines the entire procedure in the *Association rule mining* phase.

Lines 1 to 10 of the Algorithm 1 handle its initialization. 'N' denotes the time instant used for mining the *next[N]* pattern, and $\mathcal{ST}$ represents the simulation trace from which we aim to mine association rules. The minimum support threshold is min_supp (Definition 11) and the minimum confidence threshold is min_conf (Definition 13). In this algorithm, $\mathcal{FI}$ signifies a set of frequent itemsets (Definition 8). α and β represent the values of a variable at consecutive time instants in the simulation trace (*e.g.*, time instants t1 and t2). The output of the algorithm is a set of association rules ($\mathcal{R}$) (Definition 9). Subsequently, lines 11 to 14 encompass the preprocessing steps for mining association rules for the *next[N]* pattern. In Algorithm 1, lines 16 to 24 handle the preprocessing for the *until* pattern, while lines 26 to 30 accomplish the same for the *eventually* pattern. After preprocessing the $\mathcal{ST}$ for each pattern, lines 32 to 49 perform the tasks of identifying frequent itemsets in the simulation trace and then mining the association rules.

Preprocessing of Simulation Trace – next[N]:

In Algorithm 1, lines 11 to 13 demonstrate the preprocessing of the simulation trace for *next[N]* pattern. To clarify the algorithm's hypothesis, consider a rule in the form of $antecedent \rightarrow next[N]consequent$. In order to prepare the simulation trace for mining this temporal pattern, all the output of the simulation trace is moved N records above its original position. However, the inputs of the simulation trace remain as they are. The simulation trace is modified in this way since the corresponding output of input variables in a sequential hardware design may occur in the simulation trace N time instants later. This ensures correct alignment of outputs with their corresponding inputs, allowing for accurate temporal analysis and also mining patterns for different N time instants (clock cycles). In line 14, the modified simulation trace is stored for mining the *next[N]* pattern using lines 32 to 49 of the Algorithm 1.

Fig. 2 illustrates an example of preprocessing for *next[2]* pattern. The simulation trace in Fig. 2.1 is preprocessed by moving the output parts 2 time instants above their original positions, resulting in the modified simulation trace shown in Fig. 2.2. The figure uses T to represent the true value, and F to show the false value. The simulation trace consists of 5 records, divided into two categories: input variables and output variables. Each variable is assigned its corresponding value at each time instant. The last two records in Fig. 2.2 are marked as not available (NA) due to the absence of data after time instant t4 to be moved in front of these two records.

Notably, ARTmine primarily focuses on mining the essential temporal patterns such as *next[N]*, which hold significant importance in ABV. Nevertheless, the method is easily extensible to other temporal patterns like *before[N]*, due to the symmetry shared between these two patterns [25].

Preprocessing of Simulation Trace – until:

Time	Inputs			Outputs	
	v1	v2	v3	v4	v5
t0	001	01	F	11	T
t1	001	11	F	10	T
t2	101	00	T	01	T
t3	110	10	F	10	F
t4	000	01	T	11	F

(1)

moving 2 time instants

Preprocessed

Time	Inputs			Outputs	
	v1	v2	v3	v4	v5
t0	001	01	F	01	T
t1	001	11	F	10	F
t2	101	00	T	11	F
t3	110	10	F	NA	NA
t4	000	01	T	NA	NA

(2)

Figure 2: (1) Simulation trace (2) Preprocessed simulation trace

Algorithm 1 ARTmine – Association Rule Mining

1: Inputs: ▷ initialization
2: N = time instant;
3: simulation trace $\mathcal{ST}$;
4: minimum support threshold min_supp;
5: minimum confidence threshold min_conf;
6: Set of frequent itemsets $\mathcal{FI}$;
7: α = value of variable a in time t1;
8: β = value of variable a in time t2;
9: Output:
10: Set of association rules $\mathcal{R}$;
11: **case** (next[N]): ▷ preprocessing of next[N]
12: **for** all records of $\mathcal{ST}$:
13: **move** each output of $\mathcal{ST}$, N records to up;
14: **store**(moved $\mathcal{ST}$);
15: **go to** line 32;
16: **case** (until): ▷ preprocessing of until
17: $\mathcal{F} = (\beta - \alpha)/\alpha$;
18: **if** $\mathcal{F} = 0$ or $\mathcal{F} = undefined$: ▷ $\mathcal{F} = 0 \div 0 = undefined$
19: $\beta = 0$;
20: **if** $\mathcal{F} = \infty$:
21: $\beta = 0.5$;
22: **if** $\mathcal{F} = -1$:
23: $\beta = -1$;
24: **store**(modified $\mathcal{ST}$);
25: **go to** line 32;
26: **case** (eventually): ▷ preprocessing of eventually
27: **for** every input of the $\mathcal{ST}$ in time t:
28: **for** every output from time(t+1) to time(t+N):
29: **move** the output to the front of the input in time t;
30: **store**(moved $\mathcal{ST}$);
31: **go to** line 32;
32: **initialize** $\mathcal{FI}$ to be an empty set; ▷ association rule mining
33: **generate** frequent itemsets of size 1 and add them to $\mathcal{FI}$;
34: **while** $\mathcal{FI}$ is not empty do:
35: **generate** candidate itemsets Ck+1 of size k+1 by joining frequent itemsets of size k;
36: **for** each record in $\mathcal{ST}$:
37: **count** the support for each candidate itemset in Ck+1;
38: **prune** the candidate itemsets in Ck+1 that is not equal to the minimum support threshold min_supp;
39: **add** the remaining candidate itemsets to $\mathcal{FI}$;
40: **Increment** k;
41: **generate** association rules from the frequent itemsets in $\mathcal{FI}$:
42: **for** each frequent itemset X in $\mathcal{FI}$:
43: **generate** all non-empty subsets Y of X;
44: **for** each subset Y:
45: **generate** the rule Y => X-Y;
46: **calculate** the support and confidence of each rule;
47: **prune** the rules that do not meet the minimum confidence threshold min_conf;
48: **add** the remaining rules to $\mathcal{R}$;
49: **return** $\mathcal{R}$;

Lines 16 to 24 in Algorithm 1 detail the preprocessing procedure for the *until* pattern (Definition 6). To clarify the algorithm's hypothesis, consider a temporal pattern $antecedent \rightarrow until\ consequent$, where $antecedent$ comprises input variables from the simulation trace, and $consequent$ represents an output variable from the simulation trace. The preprocessing method explores the points in the simulation trace where the value of an input variable undergoes a change and subsequently identifies the corresponding output. This information is obtained through the implementation of lines 17 to 23 in Algorithm 1. In this

algorithm, α and β represent the values of a variable at two consecutive time instants (*e.g.*, t1 and t2).

The result of the equation on line 17 of the algorithm is stored in $\mathcal{F}$. Based on this value, we perform a mapping on the simulation trace as follows to explore the changes in variable values in the simulation trace according to the definition of *until* pattern:

- the first row of the simulation trace remains unchanged.
- if $\mathcal{F} = 0$ or $\mathcal{F} = undefined$, β is mapped to 0, indicating no change in the values of variables (α and β).
- if $\mathcal{F} = \infty$, β is mapped to 0.5, indicating a change from 0 (α) to 1 (β).
- if $\mathcal{F} = -1$, β is mapped to -1, indicating a change from 1 (α) to 0 (β).

The mapped values (*i.e.*, 0, 0.5, and -1) only serve as indicators for the tool to enhance the detection of changes in the simulation trace and facilitate the categorization of these distinct changes.

Although ARTmine focuses on mining the crucial temporal patterns in ABV, it can be readily extended to other patterns like *release* [25] with minor changes in its algorithm as *release* and *until* patterns share certain similarities. Specifically, the *until* pattern specifies that the consequent must hold until the antecedent becomes true, whereas the *release* pattern mandates that the consequent must persist continuously until the antecedent becomes true [25].

Preprocessing of Simulation Trace – eventually:

Algorithm 1 preprocesses the simulation trace for the *eventually* pattern in lines 26 to 31. To clarify the algorithm's hypothesis, consider a rule of the form $antecedent \rightarrow eventually!\ consequent$, where $antecedent$ comprises input variables from the simulation trace, and $consequent$ represents an output variable from the simulation trace. To prepare the concept of time for the *eventually* pattern according to Definition 7, for each row of inputs at time t of the simulation trace, all the outputs from time t+1 to t+n are moved to the front of the input at time t.

Association Rule Mining – next[N], until, eventually:

After preprocessing the simulation trace in the preceding steps to handle temporal patterns *next[N], until*, and *eventually*, the resulting preprocessed simulation trace is subsequently fed into lines 32 to 49 of Algorithm 1 to mine association rules for these three patterns. This part of the algorithm is executed similarly across all pattern types.

In lines 32 to 40 of Algorithm 1, frequent itemsets ($\mathcal{FI}$) (Definition 8) of various sizes (1-itemsets, 2-itemsets, etc.) are generated iteratively until the $\mathcal{FI}$ list is empty. Specifically, the algorithm mines frequent itemsets whose support values (Definition 10) exceed the min_supp value (Definition 11), while pruning the others. In this algorithm, 1-itemsets consist of individual variables of simulation trace, 2-itemsets are pairs of variables, etc.

After mining the frequent itemsets and adding them to the $\mathcal{FI}$ list, in lines 41 to 49 of the algorithm, the association rules are extracted from the $\mathcal{FI}$ list. To clarify these lines of the algorithm, let's consider an example where $\mathcal{FI}$ is equal to 4-itemsets of {A, B, C, D}. To generate association rules from this frequent itemset, we consider all non-empty subsets of it. These subsets are 1-itemsets of {A}, {B}, {C}, {D}, 2-itemsets of {A, B}, {A, C}, {A, D}, {B, C}, {B, D}, {C, D}, as well as the 3-itemsets of {A, B, C}, {A, B, D}, {A, C, D}, {B, C, D}, and 4-itemset of {A, B, C, D}. Afterward, for each non-empty subset Y, we generate an association rule of the form Y => X-Y. By doing so, the algorithm considers all possible combinations of items in the frequent itemset to identify significant associations between different sets of items. For example, if X = {A, B, C, D} and Y = {A, B, C}, then we generate the rule {A, B, C} => {D}. The antecedent of the rule (Y) is the subset {A, B, C}, and the consequent of the rule (X-Y) is the set difference between the frequent itemset {A, B, C, D} and the antecedent {A, B, C}, which is {D}. This process is repeated for each non-empty subset of $\mathcal{FI}$, yielding a set of association rules. Finally, we evaluate each association rule for its support and confidence, pruning those that are below the min_conf threshold (Definition 13). In this part of the algorithm, ARTmine optimizes assertion generation by thoroughly inspecting non-empty subsets of itemsets, exploring diverse combinations, and selectively pruning those that are a subset of each other and also fall below the min_supp and min_conf thresholds. This process minimizes redundancy while ensuring the creation of valid and accurate assertion sets with minimal overhead.

Increasing the min_supp value results in fewer assertions that describe more general design behavior, while decreasing the min_supp value leads to assertions covering rare design behavior (corner cases). Similarly, raising the min_conf value produces fewer but more valid assertions. Valid assertions refer to assertions that will not be violated during the simulation with different scenarios. The utilization of these values in ARTmine facilitates an effective verification process. In this paper, min_supp and min_conf are set to 0.01 and 1, respectively, as we aim to discover corner cases while achieving high design behavior coverage (details are presented in Section IV).

At this point, with the completion of the association rule mining for all three patterns, these rules serve as the fundamental components of the *Assertion mining* phase.

B. Assertion Mining

This phase consists of two steps: *Time notation* and *Assertion conversion*, which are explained in the following:

1) Time Notation

In this step, the method integrates the concept of time into the *association rules* generated in the first phase, leading to a set of *temporal association rules*. Algorithm 2 covers the time notation process for all three temporal patterns, detailed in the following sections. The initial 7 lines of the algorithm handle its initialization. It involves $\mathcal{U}$ as a set of mined association rules, and $\mathcal{R}$, $\mathcal{R}'$, and $\mathcal{R}''$ as three distinct association rules.

Time Notation – next[N]:

After mining association rules in the first phase (section III-A), the method provides us a set of rules in the general form of $antecedent \rightarrow consequent$. In this step, ARTmine determines to which temporal pattern each extracted rule belongs. Subsequently, it assigns the corresponding time label to the rule using Algorithm 2.

Line 9 of the algorithm detects which rules are associated with the *next[N]* pattern. Subsequently, line 10 assigns the corresponding N to *next[N]* in the rule $antecedent \rightarrow next[N]\ consequent$. If the antecedent value matches an

Algorithm 2 ARTmine – Time Notation

1: $\mathcal{U}$ = set of mined association rules; ▷ initialization
2: $\mathcal{R}$ = $antecedent \rightarrow consequent$ in time instant t_i;
3: $\mathcal{R}'$ = $antecedent \rightarrow consequent$ in time instant t_j;
4: $\mathcal{R}''$ = $consequent \rightarrow antecedent$;
5: $\{\mathcal{R}, \mathcal{R}'\} \in \mathcal{U}$;
6: TID = time instant difference;
7: move_count = number of moved records in the preprocessing phase, it corresponds with N in the next[N] pattern;
8: **for** a rule $\mathcal{R}$ in $\mathcal{U}$: ▷ next[N] time notation
9: **if** ($antecedent$ == an input of preprocessed sim. trace) and (($consequent$ == an output of preprocessed sim. trace) and (move_flag == true)):
10: **label** $\mathcal{R}$ as **next**[move_count] temporal association rule;
11: **else**:
12: **discard** $\mathcal{R}$;
13: **if** (($\mathcal{F}$ = 0.5) or ($\mathcal{F}$ = -1)) and ($\mathcal{R}'' \in \mathcal{U}$): ▷ until time notation
14: **label** $\mathcal{R}$ as **until** temporal association rule;
15: **discard** $\mathcal{R}''$;
16: **for** rules in form of $\mathcal{R}$ and $\mathcal{R}'$ in $\mathcal{U}$: ▷ eventually time notation
17: **if** ($count(antecedent)$ == $count(consequent)$) and (move_flag == true):
18: **for** each corresponding $antecedent$ and $consequent$ in $\mathcal{R}$:
19: TID[$\mathcal{R}$] = time_instant[$antecedent$] - time_instant[$consequent$];
20: **for** each corresponding $antecedent$ and $consequent$ in $\mathcal{R}'$:
21: TID[$\mathcal{R}'$] = time_instant[$antecedent$] - time_instant[$consequent$];
22: **if** (TID[$\mathcal{R}$] < 0) and (TID[$\mathcal{R}'$] < 0) and (TID[$\mathcal{R}$] != TID[$\mathcal{R}'$]):
23: **label** $\mathcal{R}$ as **eventually** temporal association rule;

input in the simulation trace, and the consequent value has already been moved to another record of it ('move_flag == true'), the rule is labeled as a *next* temporal association rule. Otherwise, other mined rules are discarded.

Time Notation – until:

Lines 13 to 15 in Algorithm 2 describe the process of identifying the *until* pattern. If the value of $\mathcal{F}$ (computed in phase 1) is 0.5 or -1, and for this value, both $antecedent \rightarrow consequent$ and $consequent \rightarrow antecedent$ rules were obtained in the first phase, the rule is labeled as an *until* temporal association rule. The rule $consequent \rightarrow antecedent$ is then removed from the set of mined rules. Indeed, the essential criterion for identifying a rule as an *until* pattern is the presence of both the $antecedent \rightarrow consequent$ and $consequent \rightarrow antecedent$ rules in the set of mined rules of phase 1.

Time Notation – eventually:

Lines 16 to 23 in Algorithm 2 are for detecting the *eventually* pattern. To achieve this, ARTmine first calculates the occurrences of $antecedent$ and $consequent$ in the preprocessed simulation trace, denoted as $count(antecedent)$ and $count(consequent)$, respectively. If these two values are equal, the time instants of their occurrences are captured. Subsequently, the difference in time instants (TID in Algorithm 2) between $antecedent$ and $consequent$ is calculated. If the differences are negative and non-equivalent (line 22), the extracted association rule represents an *eventually* pattern and is labeled as an *eventually* temporal association rule (lines 23). In fact, the essential condition for identifying a rule as an *eventually* pattern is that the TIDs must be negative and non-equivalent.

2) Assertion Conversion

In this step, the mined temporal association rules are transformed into temporal assertions using the labels assigned in the *Time notation* step (section III-B1). ARTmine provides assertions in SVA syntax. However, in this section, we will focus on explaining the general format of temporal mined rules in PSL [22] as it is more understandable.

The output of the *Time notation* step for temporal association rules labeled as *next[N]* is converted to the PSL format of "*always(antecedent → next[N] consequent)*". Temporal association rules labeled as *until* are transformed into "*always(antecedent until consequent)*", and rules with the *eventually* label are changed to "*always(antecedent → eventually! consequent)*" in PSL. The generated assertion sets in this phase are now ready for the verification process.

IV. Experimental Results

ARTmine[1] has been implemented in Python and evaluated using several benchmarks developed in Verilog and SystemVerilog. The benchmarks include some of the ISCAS'89 designs from [26] and Arb2, Decoder, Multdiv, Controller, and Id_stage from the GoldMine repository in [27]. One benchmark is the Bridge that connects memory and IO. The Arbiter and LBDR benchmarks are vital components of an open-source project named NoC router Bonfire [28].

Table I: Description of injected mutants

Mutation operator types	List of operators
arithmetic operators	+, −, ×, /, %
relational operators	==, !=, >, <, >=, <=
logical operators	&&, ||
assignment operators	+=, − =, ×=, /=, %=, =
unary operators	+, −, ∼, !
bitwise operators	≪, ≫, &, |, ∧
bitwise assignment operators	≪=, ≫=, & =, | =, ∧ =

Table II: Experimental results of ARTmine

Benchmarks	T_Len	Lines	I/O	#A-*N*	#A-*U*	#A-*E*	ET*N*	ET*U*	ET*E*
Arb2	100	28	6	8	0	0	0.001s	0.11s	0.05s
Id_stage	1K	813	82	1793	107	2	13s	6m11s	50s
Decoder	1K	426	4	369	20	1	0.24s	2m	38s
Controller	10K	788	57	748	72	1	12s	5m14s	37s
Multdiv	1K	559	15	348	79	4	49s	6m9s	3m12s
Arbiter	30K	245	22	105	55	3	10s	6m	3m
LBDR	80K	95	13	195	27	3	57s	7m3s	4m
Bridge	100K	196	16	79	36	1	7s	13m	7m
S27	1K	36	5	1	0	0	0.05s	20s	15s
S15850	1K	11247	227	9542	227	3	1m55s	16m42s	5m13s
S35932	1K	39481	355	1084	48	1	54s	9m11s	4m5s
S38417	1K	26190	134	1509	308	12	18s	15m	3m
S38584	1K	22734	342	5093	1460	8	1m10s	22m21s	8m14s

To evaluate the assertion sets, we have implemented an automatic mutant generator and injector following the details in [29]. We used a complete set of mutants that converted all operators and bits, injecting them into the RTL designs. Table I provides details on the injected mutants, including the types of mutated operators in the column 'Mutation operator types' and the changes made to the operators listed in each row of the 'List of operators' column. Additionally, all 0 and 1 bits have been changed to each other. Furthermore, in all the experiments for ARTmine the values for the minimum support (Definition 11) and minimum confidence (Definition 13) have been set to 0.01 and 1, respectively. These values allow the verification engineer to guide ARTmine in mining fewer yet more valid assertions, effectively considering the corner cases of designs. Moreover, N has been set to 2 for the *next[N]* pattern, but it can be adjusted to other values.

Table II shows experimental results of ARTmine, including the number of mined assertions for *next[N], until*, and *eventually* patterns (columns '#A-*N*', '#A-*U*', and '#A-*E*'),

[1] https://github.com/MohammadRezaHeidariIman/ARTmine

Table III: Comparison of ARTmine with other assertion miners

Benchmarks	#Mutants	#Assertions			Mutant Detection (%)			Execution time		
		ARTmine	HARM	GoldMine	ARTmine	HARM	GoldMine	ARTmine	HARM	GoldMine
Arb2	10	8	12	9	100	100	100	0.161s	0.49s	2.4s
Id_stage	660	1902	105716	2790	90	80	45	7m14s	13m41s	18m11s
Decoder	400	390	780	418	69	53	40	2m38.24s	4m33s	4m11s
Controller	644	821	13672	1068	91	76	37	6m3s	9m17s	15m51s
Multdiv	1801	431	21620	700	96	88	54	10m10s	17m30s	13m23s
Arbiter	860	163	2017	113	100	66	68	9m10s	20m14s	26m12s
LBDR	632	225	480	100	100	39	18	12m	15m38s	16m10s
Bridge	560	116	1596	806	78	50	31	20m7s	1h15m	36m4s
S27	12	1	1	NS *	100	100	NS *	35.05s	0.56s	NS *
S15850	2642	9772	68174	NS *	80	60	NS *	23m50s	33m09s	NS *
S35932	3700	1133	159315	NS *	52	34	NS *	14m10s	38m18s	NS *
S38417	3022	1829	15889	NS *	68	40	NS *	18m18s	25m12s	NS *
S38584	2905	6561	145925	NS *	84	62	NS *	31m45s	41m37s	NS *

* GoldMine could not provide any solution (assertion) for this benchmark.

along with their corresponding execution times ('ET*N*', 'ET*U*', and 'ET*E*'). In this table, column 'T_Len' indicates the length of the simulation traces, and column 'Lines' shows the lines of code for each benchmark. Furthermore, 'I/O' is the number of I/O in each benchmark. Experimental results show that ARTmine is capable of generating a reasonable number of assertions in a suitable amount of time, even for large-scale designs like the ISCAS'89 benchmarks.

Table III presents the efficiency of ARTmine in contrast to the most popular assertion miners in the literature, *i.e.*, HARM [10] and GoldMine [8]. The column '#Mutants' represents the number of injected mutants for each benchmark. The column '#Assertions' compares the number of assertions generated by ARTmine, HARM, and GoldMine, and the column 'Mutant Detection (%)' presents the percentage of the detected mutants. To mine the assertions for HARM, and GoldMine, we ran the tools available on their repositories in [10] and [27]. In our experiments, all assertion miners used the same simulation traces and designs, and mutant injection was performed similarly for all of them.

As can be seen, ARTmine has generated significantly fewer assertions than the other tools (in almost all cases), while being considerably more effective in mutant detection. The results indicate that HARM has generated an excessive number of assertions for most benchmarks, potentially prolonging the verification process. Conversely, GoldMine cannot handle ISCAS'89 benchmarks, as indicated by 'NS' (No Solution) in Table III. Unlike other tools, GoldMine only works with Verilog designs and cannot handle simulation traces or vcd files [8]. Moreover, it is limited to mining assertions from designs in the RTL format [8], while the ISCAS'89 benchmarks are implemented in the gate-level format, making GoldMine unable to mine any assertion for them. However, we employed ISCAS'89 benchmarks to evaluate our assertion miner with large-scale designs and compare it to the latest miner in the literature, HARM [10]. Moreover, ARTmine exhibits shorter execution times compared to HARM and GoldMine.

V. Conclusion

In this paper, we proposed an association rule mining algorithm that forms the foundation of ARTmine, an automatic assertion miner that efficiently generates accurate assertion sets encompassing *next[N]*, *until*, and *eventually* temporal patterns. ARTmine outperforms other methods by detecting more injected mutants with fewer assertions.

References

[1] M. Boulé *et al.*, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring.* Springer, 2008.

[2] X. T. Ngo *et al.*, "Hardware property checker for run-time hardware trojan detection," in *2015 ECCTD*, 2015, pp. 1–4.

[3] S. Katz *et al.*, "Have I written enough properties? A method of comparison between specification and implementation," in *ACM CHARME*, 1999, pp. 280–297.

[4] J. Malburg *et al.*, "Property mining using dynamic dependency graphs," in *ASP-DAC*, Jan 2017, pp. 244–250.

[5] G. Martino *et al.*, "Syntax-guided enumeration of temporal properties," in *2019 FDL*, 2019, pp. 1–8.

[6] A. Danese *et al.*, "A-team: Automatic template-based assertion miner," in *54th DAC conf.*, 2017, pp. 1–6.

[7] R. Hariharan *et al.*, "From rtl liveness assertions to cost-effective hardware checkers," in *2018 Conference on DCIS*, 2018, pp. 1–6.

[8] S. Vasudevan *et al.*, "Goldmine: automatic assertion generation using data mining and static analysis," in *DATE*, 2010, pp. 626–629.

[9] S. Hertz *et al.*, "Mining hardware assertions with guidance from static analysis," *IEEE TCAD*, vol. 32, no. 6, pp. 952–965, 2013.

[10] S. Germiniani *et al.*, "Harm: A hint-based assertion miner," *IEEE TCAD*, vol. 41, no. 11, pp. 4277–4288, 2022.

[11] ——, "Exploiting clustering and decision-tree algorithms to mine ltl assertions containing non-boolean expressions," in *VLSI-SoC*, 2022.

[12] M. R. Heidari Iman *et al.*, "A methodology for automated mining of compact and accurate assertion sets," in *NorCAS*, 2021, pp. 1–7.

[13] ——, "Immizer: An innovative cost-effective method for minimizing assertion sets," in *DSD*, 2022, pp. 1–8.

[14] M. R. Heidari Iman *et al.*, "An automated method for mining high-quality assertion sets," *MICPRO*, vol. 97, p. 104773, 2023.

[15] T. Ghasempouri *et al.*, "On the estimation of assertion interestingness," in *VLSI-SoC*, Oct 2015, pp. 325–330.

[16] A. Danese *et al.*, "Automatic extraction of assertions from execution traces of behavioural models," in *DATE*, 2015, pp. 67–72.

[17] S. M. Ghafari *et al.*, "A survey on association rules mining using heuristics," *WIREs DM & Knowl. Disc.*, vol. 9, no. 4, p. e1307, 2019.

[18] J. Han *et al.*, "6 - mining frequent patterns, associations, and correlations: Basic concepts and methods," 2012, pp. 243–278.

[19] A. Segura-Delgado *et al.*, "Temporal association rule mining: An overview considering the time variable as an integral or implied component," *WIREs DM & Knowl. Disc.*, vol. 10, no. 4, p. e1367, 2020.

[20] C. Antunes *et al.*, "Temporal data mining: an overview," 2001.

[21] S. Bilqisth *et al.*, "Determination of temporal association rules pattern using apriori algorithm," *IJCCS Journal*, vol. 14, p. 159, 04 2020.

[22] "Standard for property specification language (psl), ieee standard," p. 1–184, 2012.

[23] M. Shahin *et al.*, "Exploring factors in a crossroad dataset using cluster-based association rule mining," 2022, the 13th conf. on ANT.

[24] M. R. H. Iman *et al.*, "Anomalous file system activity detection through temporal association rule mining," in *9th ICISSP*, 2023, pp. 733–740.

[25] C. Czepa *et al.*, "Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability," *Software & Systems Modeling*, vol. 18, 12 2019.

[26] "ISCAS'89," https://sportlab.usc.edu/~msabrishami/benchmarks.html.

[27] "Goldminer," https://bitbucket.org/debjitp/goldminer/src/master/example/.

[28] "Project Bonfire Network-on-Chip," https://github.com/Project-Bonfire/Bonfire_handshake, 2017.

[29] U. Repinski *et al.*, "Combining dynamic slicing and mutation operators for esl correction," in *ETS*, 2012, pp. 1–6.