Automatic Generation and Qualification of Assertions on Control Signals: A Time Window-Based Approach

Alessandro Danese, Francesca Filini, Tara Ghasempouri, Graziano Pravadelli

Automatic Generation and Qualification of Assertions on Control Signals: A Time Window-based Approach

Alessandro Danese, Francesca Filini, Tara Ghasempouri, and Graziano Pravadelli

Department of Computer Science, University of Verona, Italy
alessandro.danese@univr.it, francesca.filini@studenti.univr.it,
tara.ghasempouri@univr.it, graziano.pravadelli@univr.it

Abstract. Assertion-based verification (ABV) is a promising approach for proving that the design implementation is consistent with the designer's intents. However, ABV effectiveness depends on the quality of the assertions that are defined to capture the designer's intents. Assertions are generally defined by verification engineers that manually convert informal specifications in logic formulas according to their expertise. However, manual definition is a time-consuming and error-prone activity, which may fail to exhaustively cover either the intended specification or the implemented behaviours. For this reason, different mining approaches have been recently proposed for the automatic generation of assertions. Unfortunately, in most cases, existing mining tools generate a set of over-constrained assertions. As a consequence, each assertion in the set is a long formula that describes a very specific behaviour of the design under verification (DUV). Thus, in the effort of covering as much DUV behaviours as possible, these approaches generate a huge amount of assertions with a negative impact on the total time required for their verification. To overcome this drawback, this paper introduces a dynamic approach that incrementally analyses control signals on DUV execution traces for mining more expressive temporal assertions that better capture the I/O communication protocol. Then, to evaluate the effectiveness of the generated assertions in covering the intended behaviours, a technique is proposed to estimate assertion's interestingness by ranking them according to metrics typically adopted in the context of data mining.

Keywords: assertion mining, assertion qualification, assertion-based verification

1 Introduction

Despite the advancement in simulators and formal methods, the verification result is only as good as the specifications defined to capture the designer's intent. Without a good set of specifications, designers can easily lose the control of the system design, and they cannot understand what behaviour of the

system is implemented in the reality. Unfortunately, even if everybody agrees that the first and the most important step for a good development task is the collection of all the intents and specifications, often this does not receive an appropriate attention. In the last decade, assertion-based verification (ABV) has arises as one of the most popular solutions for electronic system level (ESL) and Register Transfer Level (RTL) verification [1]. ABV relies on the definition of assertions, i.e., logic formulas, generally written according to temporal logics, like LTL and CTL [2], and property specification languages, like PSL [3], that formalize the behaviours of the DUV by overcoming the ambiguity of natural languages and providing the engineers with precise and well-defined specifications. However, manual definition of assertions requires high expertise and it is an error-prone and time-consuming activity. Main problems are related to the risk of defining assertion sets that are incomplete (i.e., unable to cover all expected behaviours of the DUV), inconsistent (i.e., with contradicting assertions), redundant (i.e., with assertions that are logical consequence of others), and including vacuous assertions (i.e., assertions that are true independently from the DUV, and thus irrelevant). As a result, a false sense of security is induced by an ABV campaign conducted with a low-quality set of assertions. As a complementary approach to manual definition of assertions, several approaches and tools have been proposed for automatically extracting safety assertions in the form $always(antecedent \rightarrow consequent)$ from the implementation of the DUV [4–8]. These approaches either rely on static analysis of the DUV source code or they dynamically mine assertions from execution traces of the DUV. The first are accurate and provide assertions that are formally proved to be satisfied by the DUV, but they do not scale well for complex DUVs. The second provide only likely assertions, whose quality depends on the observed execution traces (i.e., likely assertions are guarantee to hold at least for the considered execution traces, however a counter example could be finally found), but they do not require the source code and guarantee a better scalability for complex DUVs. Independently from the adopted techniques, mined assertions can be compared with design intents to discover unexpected behaviours implemented in the DUV, to confirm that relevant behaviours are actually implemented, and for documentation purposes. Existing approaches generally extract a high number of long assertions, but each of them covers a few specific behaviours of the DUV. This depends on the fact that mined assertions are over-specified, i.e., their antecedents and/or consequents include several atomic propositions that predicate almost on all primary inputs and outputs of the DUV. Moreover, mined assertions mix data and control signals making difficult the characterization of the I/O communication protocol of the DUV. Large sets of over-constrained assertions make impractical the analysis of mined assertions by verification engineers and greatly increase verification time. Furthermore, while vacuity and inconsistency in the set of generated assertions are generally avoided by the mining approach itself, assertion incompleteness and redundancy may still affect the outcome of assertion mining. Thus, a qualification phase to identify the most interesting assertions is necessary to focus designer's attention on relevant assertions that capture ex-

pected (or unexpected) behaviours implemented in the DUV. Unfortunately, as the number of mined assertions can be very high, their manual qualification is almost impractical, while current approaches for automatic evaluation are still unsatisfactory from the point of view either of the effectiveness [9–12] or of the efficiency [13–16].

In the previous context, the goal of this paper consists of overcoming the drawbacks of existing works in assertion generation and assertion qualification by proposing:

- First, a dynamic *assertion generation* approach that infers assertions by incrementally analysing time windows of the DUV's execution traces. The algorithm searches for recurrent temporal patterns among atomic propositions predicating on I/O control signals. Mined assertions are in average shorter, more expressive from the point of view of the I/O communication protocol, and then simpler to be understood by humans, compared, for example, to the approach proposed in [5]. This reduces verification time, and increases the effectiveness of verification engineers in discovering design errors by analysing mined assertions.
- Secondly, an automatic *qualification* approach for evaluating the quality of extracted assertions and rank them accordingly. The estimation of the interestingness of assertions is achieved by ranking them according to probabilistic metrics typically adopted in the context of data mining (i.e., support and correlation coefficient) [17, 18], which we adapt here for the specific case of assertion mining. From the point of view of the general concept, data mining and assertion mining share the same idea (extracting rules from data), but they have several differences that make practically different how these metrics are computed and interpreted for evaluating the interestingness of assertions.

The rest of the paper is organized as follows. Section 2 summarizes the related works. Section 3 presents some preliminary definitions and concepts relevant for understanding the technical details of our approach. Section 4 provides an overview of the proposed methodology, which is then thoroughly described in Section 5 and Section 6, respectively, for assertion mining and assertion qualification. Finally, experimental results and concluding remarks are reported, respectively, in Section 7 and Section 8.

2 Related Works

Different strategies have been proposed for assertion mining. Among the first works in the software domain, scenario-based specification mining approaches proposed in [19, 20] require instrumenting the source code to store the sequence of method calls among the components of the design in an execution trace. Data mining methods are then applied to program execution traces in order to mine strongly observed inter-component universal sequence diagrams in the form of live linear sequence charts (LSC). LSCs represent how the components cooperate

to implement certain system features. However, these approaches are not aimed at discovering the complete behaviour of the components but only the collaboration among them. Other works mine the specifications of the design in form of algebraic equation [21] or Hoare-style equations of pre and post-conditions [22]. In particular, [22] automatically deducts arithmetic relationships that predicate exclusively on values and data-structures, but the temporal behaviours are not considered. Temporal assertion mining is described in [23], where a mining tool, GoldMine, is proposed for extracting Boolean-level assertions for HW components. On the contrary, [24, 9] and [11] propose methodologies for specification mining which are able to manage indistinctly HW and embedded SW descriptions according to a set of pre-defined temporal patterns. They rely on Daikon [25] to mine relevant arithmetic/logic relations among the variables of the DUV from execution traces. In particular, [24, 9] try to mine a set of temporal assertions reporting how arithmetic/logic relations change during the execution of the design. Even if these works introduce a novel approach in which the extraction of temporal relations between arithmetic/logic expressions represents a good strategy to describe more closely the behaviours of the design, the generated assertions are hard to be understood because they involve all primary inputs and outputs of the DUV. As results, mined assertions cannot be easily analysed from designers to detect unexpected behaviours. On the other hand, [11] is focused only on catching simple arithmetic/logic relations on the assumption that a behaviour of the DUV can be justified essentially through a comparison between the set of verified arithmetic/logic relations in an execution instant and the set of falsified arithmetic/logic relations in the closer next execution instants. Finally, commercial tools are also available for automatic assertion generation, e.g., Atrenta BugScope [26] and Jasper ActiveProp [7] that works on RTL models at Boolean level, but no arithmetic/logic expressions are considered among more abstracted data types.

Concerning assertion qualification, current approaches are still unsatisfactory to measure the quality and interestingness of assertions. In [9], a stressing phase is proposed only to verify the likelihood that mined assertions are globally satisfied (and not only for the execution traces analysed by the miner), but no strategy is proposed to measure their interestingness in covering DUV behaviours. In [10], interestingness estimation is based on the number of propositions included in the antecedent of the assertion, according to the fact that an assertion with a lower number of propositions in its antecedent has an higher input space coverage than one with many propositions in its antecedent. However, the correlation between the antecedent and the consequent of an assertion is not considered. To solve this drawback, in [11] a ranking function is proposed that evaluates the quality of the mined assertions in terms of cause-effect relationship between antecedent and consequent of an assertion. Finally, in [12], mined assertions are said to be generally ranked according to their frequency of occurrences and time of first occurrence but no specific approach is presented.

As an opposite class of approaches, coverage metrics have been widely studied for qualification of assertions [13–16]. Most of these works relies on mutation

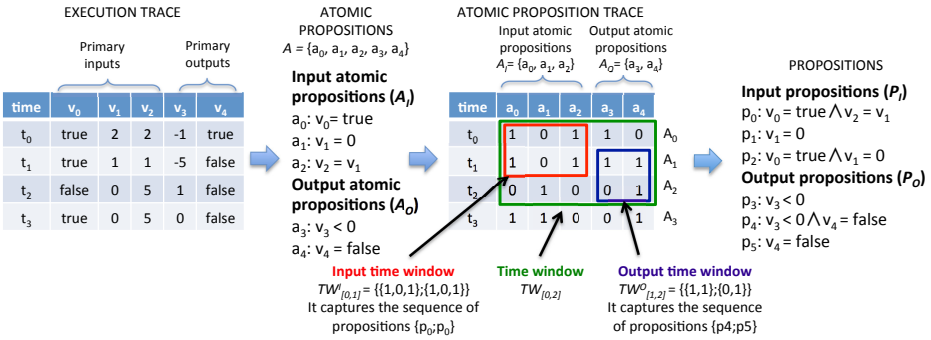


Fig. 1: Exemplification of execution trace, atomic proposition trace and time window.

analysis, which requires perturbing the DUV implementation by injecting mutations (faults) to check, either statically [14, 15] or dynamically [16], whether they change the truth values of the assertions; mutations that do not cause a change are said to be not detected. Assertions that detect a few mutations are less interesting than assertions detecting an higher number of mutations. Not detected mutants generally highlight area/behaviours of the DUV that are not covered by any of the defined assertions showing a hole on the coverage. Dynamic approaches like [16] scale better with respect to static techniques, however, they still require long simulation runs for checking each assertion for each mutation with a significant set of testbenches. When the number of assertions is very high, as in the case of assertions extracted automatically, evaluating their interestingness through mutation analysis becomes a very time-consuming activity.

3 Background and Preliminaries

Before describing the proposed methodology, some definitions and concepts concerning assertion generation and assertion qualification are reported to create the necessary background.

3.1 Definitions

Definition 1 ***(Execution trace)*** *Given a finite sequence of simulation instants* $\langle t_1,...,t_n \rangle$ *and a model* M *working on a set of variables* V*, an execution trace of* M *is a finite sequence of pairs* $T = \langle (V_1,t_1),...,(V_n,t_n) \rangle$ *where* $V_i = eval(V,t_i)$ *is the evaluation of variables in* V *at simulation instant* t_i*.*

More informally, an execution trace describes for each simulation instant t_i the values assumed by each variable included in V during the evolution of the design under verification M. In this paper, variables in V are primary inputs and primary outputs representing control signals of the DUV. An example of an execution trace is reported in Figure 1 (left). By analysing an execution trace, we

can extract a set of atomic propositions that predicate on variables included in V.

Definition 2 ***(Atomic proposition)*** *An atomic proposition is a formula that does not contain any logical connective.*

In our methodology, a set of atomic propositions is organized in an array $A = \{a_1, \ldots, a_m\}$ which is further divided in two sub-arrays: the array of *input atomic propositions* $A_I = \{a_0, \ldots, a_i\}$, whose elements are atomic propositions that predicate only on primary inputs of the DUV, and the array of *output atomic propositions* $A_O = \{a_{i+1}, \ldots, a_m\}$, whose elements predicate only on primary outputs of the DUV. We do not consider any atomic proposition that mixes primary inputs and primary outputs. Examples of atomic propositions are reported in Figure 1 (centre). From an execution trace and the corresponding set of atomic propositions, we can generate an atomic proposition trace.

Definition 3 ***(Atomic proposition trace)*** *Given an execution trace T and an array of atomic propositions A, an atomic proposition trace is a finite sequence of pairs $\omega = \langle (A_0, t_0), \ldots, (A_n, t_n) \rangle$ where $A_i = eval(A, t_i)$ is an array that represents the evaluation of atomic propositions in A at simulation instant t_i, i.e., $A_i[j] = 0$ if $A[j] = false$ at time t_i, $A_i[j] = 1$ otherwise.*

The atomic proposition trace is the data structure we use to mine propositions.

Definition 4 ***(Proposition)*** *A proposition is a composition of atomic propositions through logic connectives. An atomic proposition itself is a proposition.*

It this paper, we consider propositions involving only the logic *and* ($\wedge$) as connective, and we classify them in two different sets:

- The set of *input propositions* (P_I): a proposition p belongs to P_I if it is composed only of input atomic propositions and $\wedge$ connectives;
- The set of *output propositions* (P_O): a proposition p belongs to P_O if it is composed only of output atomic propositions and $\wedge$ connectives.

An example of an atomic proposition trace and a set of input/output propositions that can be extracted from it are shown in Figure 1 (right). To represent input and output propositions in a compact and efficient way, we use an array-based notation. Given an array of input (respectively, output) atomic propositions A_I (A_O), an input (output) proposition is represented by an array of Boolean values p such that $p[i] = 0$ if the input (output) atomic proposition $A_I[i]$ ($A_O[i]$) is not used in the proposition, $p[i] = 1$ otherwise. For example, the input proposition p_0 in Figure 1 can be represented by the array $\{1, 0, 1\}$.

Definition 5 ***(Time window)*** *Given an atomic proposition trace $\omega = \langle (A_0,t_0), \ldots, (A_n,t_n) \rangle$, and two simulation instants t_i and t_j such that $1 \leq t_i \leq t_j \leq n$, a time window $TW_{[i,j]} = \langle (A_i,t_i), \ldots, (A_j,t_j) \rangle$ is the subsequence of contiguous elements of ω included between instant t_i and instant t_j.*

Given a time window $TW_{[i,j]}$, and a simulation instant t_k such that $t_i \leq t_k \leq t_j$, we can separate $TW_{[i,j]}$ in two parts:

– The *input time window* $TW^I_{[i,k]}$, which is composed of elements of $TW_{[i,j]}$ included in the simulation instants between t_i and t_k, restricted to the input atomic propositions.
– The *ouptut time window* $TW^O_{[k,j]}$, which is composed of elements of $TW_{[i,j]}$ included in the simulation instants between t_k and t_j, restricted to the output atomic propositions.

The input time window $TW^I_{[i,k]}$ and the corresponding output time window $TW^O_{[k,j]}$ overlap for exactly one simulation instant (t_k). Given a time window $TW_{[i,j]}$, we can generate $j - i + 1$ different couples of input/output time windows, one for each simulation instant $t_k \in [t_i, t_j]$. For example, in the atomic proposition trace of Figure 1, the green box highlights a time window composed of 3 simulation instants in the interval $[t_0, t_2]$, the red box corresponds to an input time window in the interval $[t_0, t_1]$, and finally the blue box shows the corresponding output time window in $[t_1, t_2]$. In the rest of the paper we will represent an input (output) time window by the sequence of arrays corresponding to the input (output) propositions it captures. For example, in Figure 1, the input time window corresponding to the interval $[t_0, t_1]$ is represented by the sequence of two arrays $\{\{1, 0, 1\}; \{1, 0, 1\}\}$.

By analysing input and output time windows in the atomic proposition trace we can mine temporal assertions.

Definition 6 ***(Temporal assertion)*** *A temporal assertion is a composition of propositions through temporal operators and logic connectives.*

In this paper, we consider Linear Temporal Logic (LTL) assertions in the form $G(antecedent \rightarrow consequent)$, where G is the LTL *always* operator[1], and *antecedent* and *consequent* may involve only X, i.e., the LTL *next* operator[2] and the $\wedge$ connective. Moreover, *antecedent* is composed only of an arbitrary number of propositions belonging to the set P_I extracted by analysing an input time window $TW^I_{[i,k]}$, while *consequent* includes a single proposition belonging to the set P_O extracted by analysing the corresponding output time window $TW^O_{[k,j]}$. We selected this specific form of assertion, since it is suited to describe the behaviour of the I/O communication protocol of a DUV, which, as reported in the introduction, represents the target for the current work. For example, from the atomic proposition trace of Figure 1 the following temporal assertion can be mined: $G((p_0 \wedge X^2(p_1)) \rightarrow X^3(p_5))$.

In according to the above definitions and notations, we define hereafter a set of operators working on propositions that will be used to illustrate the mining methodology proposed in Section 5.

RemoveAtomicProp: Given an array of Boolean elements p corresponding to a proposition, and an index i corresponding to the atomic proposition a_i, which

[1] Given a formula α, $G(\alpha)$ means that α is always true.
[2] Given a formula α, $X(\alpha)$ means that α is true at the next instant. As a short cut, we will write $X^n(\alpha)$ to represent the application of n consecutive next operators to the formula α.

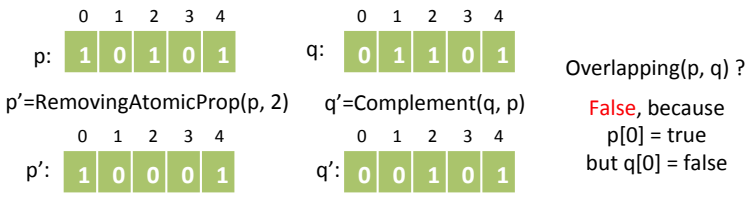


Fig. 2: Examples of application of *RemoveAtomicProp*, *Complement* and *Overlapping*.

represents the element at position i of p, $RemoveAtomicProp(p, i)$ returns a new array p' where $p'[j] = p[j]$ for all indexes $j \neq i$ and $p'[i] = 0$, i.e., the proposition corresponding to p' does not include a_i.

AddAtomicProp: Given an array of Boolean elements p corresponding to a proposition, and an index i corresponding to the atomic proposition a_i, which represents the element at position i of p, $AddAtomicProp(p, i)$ returns a new array p' where $p'[j] = p[j]$ for all indexes $j \neq i$ and $p'[i] = 1$, i.e., the proposition corresponding to p' includes a_i.

Overlapping: Given two arrays of Boolean elements p and q corresponding to two propositions predicating over the same array of atomic propositions, $Overlapping(p, q)$ returns *true* if $q[i] = 1$ for at least all indexes i such that $p[i] = 1$, it returns $false$ otherwise.

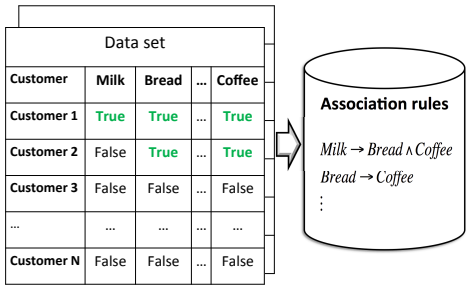
Complement: Given two arrays of Boolean elements p and q corresponding to two propositions predicating over the same array of atomic propositions, $Complement(p, q)$ returns a new array p' where $p'[i] = false$ for all indexes i such that $q[i] = false$, and $p'[j] = p[j]$ for all indexes j such that $q[i] = true$.

For example, Figure 2 shows the application of operators *RemoveAtomicProp*, *Complement* and *Overlapping* to propositions. In a more general way, we apply *Complement* and *Overlapping* also to sequences of propositions of the same length. In this case, given two sequences of propositions s_1 and s_2, *Complement* and *Overlapping* operate iteratively on each couple of corresponding elements (p_i, q_i) such that $p_i \in s_1$ and $q_i \in s_2$.

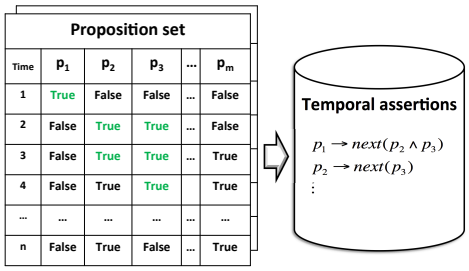
In addition to the previous definitions, which are necessary to understand the assertion mining approach detailed in Section 5, the rest of this section summarizes some concepts concerning data mining, such as itemsets, transactions and association rules, which are necessary to present the assertion qualification methodology proposed in Section 6.

Definition 7 *Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items. Let $D = \{d_1, d_2, \ldots, d_m\}$ be a data set, i.e., a set of observations, called transactions, with respect the set of items I. Each element in D contains a subset of the items in I. An association rule is defined as an implication of the form $X \rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. X and Y are called itemsets.*

Figure 3(a) shows an example of a data set which describes the behaviours of customers in a supermarket with respect to a set of items (i.e., milk, bread,



(a) Data mining concept.



(b) Assertion mining concept.

Fig. 3: Similarities between data mining (a) and assertion mining (b).

..., coffee). Data mining approaches are generally intended to extract association rules from data sets, which are then used to predict non trivial, implicit, previously unknown and potential useful information, like, for example,"when milk is bought bread and coffee are generally bought too", which is expressed by the association rule $Milk \rightarrow Bread \wedge Coffee$. Assertion mining deals instead with executing traces and temporal assertion which are formalized in Def. 1 and Def. 6 respectively. An example of a temporal assertion in Linear Time Logic is $always(p_1 \rightarrow next(p_2 \wedge p_3))$ which states it always happens that p_2 and p_3 are satisfied one simulation instant later than p_1 becomes true (Figure 3(b)).

In this paper, without lack of generality, and to preserve the independence of the proposed qualification methodology from specific temporal patterns instantiated in the analysed assertions, we generically consider formulas in the form $A \rightarrow C$, where the antecedent A and the consequent C are composed by propositions, logic connectives, and temporal operators according to the selected temporal logic. The initial hypothesis is that the analysed assertions are true on the DUV.

3.2 Comparing Data Mining and Assertion Mining

The overall goal of data mining is to extract information from a data set and transform it into an understandable and useful structure. This structure allows user analysing data from many different dimensions, categorizing them and summarizing correlations between items in a database. For example, analysing data from behaviours of different customers as reported in Figure 3(a) leads to obtain useful information and helps analysers to decide which trend is more interesting for marketing. Association rules can also be extracted when data are referred to time sequences. In this case, temporal data mining strategies are adopted, whose goal is to discover hidden relations between sequences and sub sequences of events [27]. In any case, the mined (temporal) association rules are a prediction for future behaviours, which may be true or not. Metrics are thus used to estimate the probability that rules extracted from past observations can be valid also in the future. On the contrary, the main goal of assertion mining consists of extracting formulas that exactly describe the functionality implemented in the DUV, which is not ambiguous and does not vary in the future, except in the case the implementation is changed. Assertion mining is thus not intended to predict the future, but to formalize the actual set of DUV behaviours. Summarizing, main similarities among data mining and assertion mining are the presence of a set of data that represents observations with respect to past behaviours exposed by the observed target (customers, DUV, ...), and the need of extracting association rules that formalize such observations. As shown in Figure 3(a) and Figure 3(b), items, data sets, and association rules in data mining correspond, respectively, to propositions, execution traces, and temporal assertions in assertion mining. Meanwhile, the main difference between data mining and assertion mining is represented by the concept of transaction (i.e., a row in a data set), which does not have a direct correspondence with a row of an execution trace, because a temporal assertion is composed by one antecedent and one consequent that are true in different instants inside the execution trace. This difference impacts on the way metrics typically adopted for evaluating association rules in data mining can be reused for measuring the interestingness of assertions. Finally, another difference is related to the final goal of the mining: in one case the prediction of future behaviours, in the other the formalisation of actual (unmodifiable, except in the case the DUV functionality is changed) behaviours.

4 Methodology

The proposed methodology (Figure 4) starts from a set of execution traces obtained by simulating the DUV, which represent the input for the assertion miner. The miner first extracts atomic propositions representing the building blocks and then compose them to generate temporal assertions. Finally, the generate assertions are evaluated according to their degree of interestingness and a final ranked set of assertions is provided.

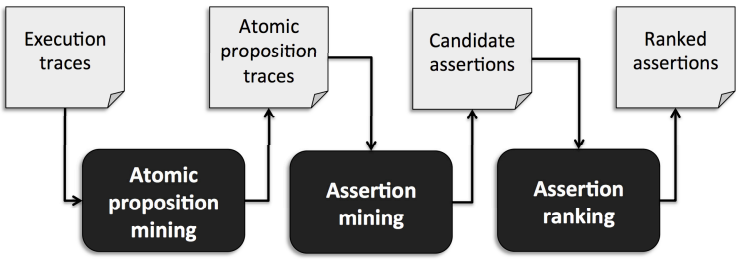


Fig. 4: Methodology overview.

5 Assertion Mining

Given a DUV and a related set of execution traces, the proposed methodology consists of two main phases: (i) mining of atomic propositions and generation of the corresponding atomic proposition traces, and (ii) extraction of input and output time windows and mining of temporal assertions. Since, the first phase has been already faced in [9], this paper focuses only on the second phase.

The main procedure of the proposed approach is showed in Algorithm 1. The function MAIN takes as parameters an atomic proposition trace ω and two thresholds: tw_len, and max_len, with $max\_len \leq tw\_len$. The first threshold (tw_len) advises how many clock cycles are required, at maximum, by the DUV to compute its functionality. This parameter, which can generally be obtained from the documentation or from simulation of the DUV, represents the length of the time windows that will be analysed by the miner inside ω. The second threshold (max_len) is used to bound the maximum number of instants that will be considered to mine the antecedent of a temporal assertion, i.e., it represents the maximum length for an input time window. Several DUVs need a high number of clock cycles for completely computing their functionality, but only few of them could be necessary to set control signals such that the computation phase starts. For instance, to mine interesting antecedents for an encryption/decryption component, it is relevant catching what happens to control signals during the initialization phase, before the computation proceeds without affecting control signals till outputs are ready. Similarly, in the case of a network component, it is relevant to capture in the antecedent of an assertion how control signals change during the phase that makes data transmission starting. The max_len threshold is then intended to guide the mining procedure such that, for mining antecedents, it focusses on the initial part of the considered time windows when the DUV is characterized, after an initialization phase, by a long elaboration phase that does not involve control signals any more till results are ready. In this case, generally max_len is small with respect to tw_len. On the contrary, max_len is set to be equal to tw_len for DUVs where control signals can change the functionality of the DUV at each simulation instant.

```
1: function MAIN(ω, tw_len, max_len) return assertions_set
2:     len = 1
3:     assertions_set = ∅
4:     while len ≤ max_len do
5:         behaviours = getBehaviours(ω, tw_len, len)
6:         pruningBehaviours(assertions_set, behaviours)
7:         candidates = getAssertions(ω, behaviours)
8:         assertions_set = getAssertions(ω, behaviours)
9:         len = len + 1
10:    end while
11: end function
```

Algorithm 1: Main procedure.

After initializing the *assertions_set* to the empty set, the main loop of Algorithm 1 cyclically calls the following three functions, by varying the length (len) of the considered input time window at each iteration, while $len \leq max\_len$.

1. $getBehaviours(\omega, tw\_len, len)$: it detects the behaviours exposed by the DUV in the atomic proposition trace. Each behaviour is represented by a sequence (with length between 1 and len) of input propositions associated to a unique output proposition. For example, in the atomic proposition trace of Figure 1, each time the input proposition $p_0 = \{1, 0, 1\}$ (corresponding to $a_0 \wedge a_2$) is true at a simulation instant t_i, the output proposition $p_5 = \{0, 1\}$ (corresponding to a_4) holds at simulation instant t_{i+1}. Thus, a behaviour can be identified that associates the input proposition p_0 to the output proposition p_5 with an *offset* of one simulation instant.
   The second behaviour associates p_0 to the output proposition $p_5 = \{0, 1\}$, because each time p_0 holds, the output proposition a_4 is satisfied in the next simulation instant.
2. $pruningBehaviours(assertion\_set, behaviours)$: it removes all the behaviours returned by $getBehaviours$ that are already captured by assertions mined in the previous iterations of the main procedure. For example, if the assertion $\alpha = G(a_2 \rightarrow X(a_4))$ is already included in the $assertion\_set$, a behaviour, which associates, with an offset of one simulation instant, the input proposition p_0 to the output proposition p_5, can be discarded, since it will create the assertion $G(a_0 \wedge a_2 \rightarrow X(a_4))$ that is a logical consequence of α.
3. $getAssertions(\omega, behaviours)$: it is in charge of creating a set of candidate temporal assertions, compliant with the form described in Subsection 3, starting from the behaviours survived after $pruningBehaviour$ is called.

The next sections describe in details how the previous functions work to mine temporal assertions.

5.1 Mining of Interesting Behaviors

The function $getBehaviours$ is implemented as shown in Algorithm 2. It takes as arguments an atomic proposition trace ω and the two thresholds tw_len

```
1: function getBehaviours(ω, tw_len, len) return behaviours
2:     behaviours = ∅
3:     dictionary = ∅
4:     t_i = 0
5:     while t_i ≤ (length(ω) − tw_len) do
6:         in = propSeq(TW^I_[t_i, t_i+len−1])
7:         out = propSeq(TW^O_[t_i+len−1, t_i+tw_len−len+1])
8:         ⟨in, old_out⟩ = findInDictionary(dictionary, in)
9:         if ⟨in, old_out⟩ = NIL then
10:            dictionary = dictionary ∪ ⟨in, out⟩
11:        else
12:            new_out = Complement(old_out, out)
13:            dictionary = dictionary \ ⟨in, old_out⟩
14:            dictionary = dictionary ∪ ⟨in, new_out⟩
15:        end if
16:        t_i = t_i + 1
17:    end while
18:    for all ⟨in, out⟩ ∈ dictionary do
19:        offset = 0
20:        for all p ∈ out do
21:            if p ≠ NIL then
22:                behaviours = behaviours ∪ ⟨in, offset, p⟩
23:            end if
24:            offset = offset + 1
25:        end for
26:    end for
27: end function
```

Algorithm 2: Extraction of DUV behaviours.

and *len*. Its goal is to detect behaviours exposed in the execution traces of the DUV, in the form of associations between a sequence of input propositions and a corresponding sequence of output propositions. In order to detect such associations, a time window $TW_{[t_i,t_i+tw\_len-1]}$ of length tw_len is analysed for each simulation instants t_i (lines 5-17). Such a time window is decomposed in an input time window of length len, and a corresponding output time window of length $tw\_len - len + 1$. Then, from the input time window, the function *propSeq* extracts the corresponding sequence of input propositions (in) that hold in the interval $[t_i, t_i + len - 1]$ (line 6). Similarly, *propSeq* extracts from the output time window the sequence of output propositions (out) that hold in the interval $[t_i + len - 1, t_i + tw\_len - len + 1]$ (line 7). For example, let us consider the atomic proposition trace of Figure 1 and let we fix $tw\_len = 2$ and $len = 1$. At time t_0, we have $in = \{1, 0, 1\}$ while out is represented by the sequence $\{\{1, 0\}; \{1, 1\}\}$. The function then searches if in is already present in the dictionary of the collected input/output propositions pairs (line 8-9). If this is not the case (line 10), a new association $\langle in, out\rangle$ is added to the dictionary. On the contrary, when in is already present, the operator *Complement* is applied to return a new sequence of output propositions new_out, where atomic propositions excluded from elements of out are excluded also from the corresponding elements of new_out. Then, the couple $\langle in, old\_out\rangle$ is replaced in the dictionary by $\langle in, new\_out\rangle$ (lines 12-14). This replacement is necessary to refine the already collected behaviours by removing from their output propositions the atomic propositions that become false

in the current output time window, such that only behaviours that are never violated throughout the trace are finally collected. The replacement happens, for example, at simulation instant t_1 of the atomic proposition trace of Figure 1. In fact, at t_1, $in = \{1, 0, 1\}$ is associated to $out = \{\{1, 1\}; \{0, 1\}\}$. However, in was already associated to $old\_out = \{\{1, 0\}; \{1, 1\}\}$ at t_0. Thus, in the dictionary $\langle \{1, 0, 1\}, \{\{1, 0\}; \{1, 1\}\}\rangle$ is replaced by $\langle \{1, 0, 1\}, \{\{1, 0\}; \{0, 1\}\}\rangle$ after the application of $Complement(old\_out, out)$. After the creation of the dictionary that collects associations between sequences of input and output propositions, the final loop (lines 18-26) creates a set of behaviours for each pair $\langle in, out\rangle$. In particular, a behaviour, represented by a triplet $\langle in, offset, p\rangle$, is extracted for each proposition p captured in the sequence of output propositions out, where $offset$ represents the distance, computed in simulation instants, between the last element of in and p. For example, for the pair $\langle \{1, 0, 1\}, \{\{1, 0\}; \{0, 1\}\}\rangle$ the following two behaviours are extracted: $\langle \{1, 0, 1\}, 0, \{1, 0\}\rangle$ and $\langle \{1, 0, 1\}, 1, \{0, 1\}\rangle$ to represent respectively that the output proposition $\{1, 0\}$ holds exactly at the same time of input proposition $\langle \{1, 0, 1\}$ (*offset* is 0), while the output proposition $\{0, 1\}$ holds one simulation instant later (*offset* is 1).

5.2 Pruning of Behaviours

The function *pruningBehaviours* takes as arguments a set of assertions and a set of triplets representing behaviours collected by *getBehaviours* in the form $\langle in, offset, p\rangle$. The goal of *pruningBehaviours* is to preserve only the triplets that are not already covered by assertions collected in previous iteration of Algorithm 1. A triplet is covered by an assertion α when the following conditions are true concurrently:

1. All input atomic propositions included in the antecedent of α are also present in in (i.e., values assigned to primary inputs of the DUV that satisfy the atomic propositions included in in satisfy also the antecedent of α).
2. All output atomic propositions included in the consequent of α are also present in p (i.e. the consequent of α is at least as detailed as p).
3. The distance, computed in simulation instants, between the last input atomic proposition of the antecedent of α and the consequent of α equals $offset$.

Triplets that falsify at least one of the previous conditions are preserve, the others are discarded.

5.3 Mining of Assertions

Given an atomic proposition trace ω and a set of triplets of the form $\langle in, offset, p\rangle$ representing behaviours preserved by the *pruningBehaviours* function, the *getAssertions* function works as described in Algorithm 3. Its goal is to extract an assertion of the form $G(antecedent \rightarrow consequent)$ from every triplet, such that the input propositions $\{p_0, \ldots, p_i\}$ captured inside the sequence in act as an antecedent of the form $(p_0 \wedge \cdots \wedge X^i(p_i))$, while $X^{i+offset}(p)$

```
1: function getAssertions(ω, behaviours, len) return ass_set
2:     ass_set = ∅
3:     for all ⟨in, offset, p⟩ ∈ behaviours do
4:         s_in = simplify(⟨in, offset, p⟩)
5:         ass_set = ass_set ∪ makeAss(s_in, offset, p)
6:         pruningBehaviours(ass_set, behaviours)
7:     end for
8: end function
9:
10: function simplify(ω, ⟨in, offset, p⟩)
11:     len = length(in)
12:     p_off = len − 1 + offset
13:     for all  q ∈ in do
14:         s_q = q
15:         i = 0
16:         while i < length(q) do
17:             s_q = RemoveAtomicProp(s_q, i)
18:             new_in = (in \ q) ∪ s_q
19:             t_i = 0
20:             while t_i ≤ (length(ω) − p_offset) do
21:                 temp = propSeq(TW^I_[t_i, t_i+len−1])
22:                 if Overlapping(new_in, temp) then
23:                     temp = propSeq(TW^O_[p_off, p_off])
24:                     if !Overlapping(p, temp) then
25:                         s_q = AddAtomicProp(s_q, i)
26:                         break
27:                     end if
28:                 end if
29:                 t_i = t_i + 1
30:             end while
31:         end while
32:         in = (in \ q) ∪ s_q
33:     end for
34:     return in
35: end function
```

Algorithm 3: Generation of temporal assertions.

represents the consequent. This is performed by the *makeAss* function at line 5. After an assertion is added to the set of candidates ass_set, the *pruningBehaviours* function is called to remove behaviours implicitly covered by the new assertion (line 6).

In order to increase the DUV behaviours covered by the mined assertions, before calling *makeAss*, the input proposition included in the sequence in is first simplified by removing atomic propositions from the antecedent such that the consequent can be verified by a higher number of simulation instants, thus enforcing the final assertion (line 4). For instant, let us consider the triplet $\langle in, offset, p\rangle$ represented by $\langle\{\{1,1,0\}\}, 0, \{\{0,1\}\}\rangle$. Looking at Figure 1, we can see that the output proposition $\{0,1\}$ is true at both simulation instants t_2 and t_3, but the input proposition $\{1,1,0\}$ is verified only at simulation instant t_3. However, if we set to $false$ the first input atomic proposition of $\{1,1,0\}$, we obtain the proposition $\{0,1,0\}$, which is true at both simulation instant t_2 and t_3. Thus, $in = \{\{1,1,0\}\}$ can be replaced by $s\_in = \{\{0,1,0\}\}$ in the triplet to cover a wider time window in the atomic proposition trace. In this way, the assertion $G(a_1 \rightarrow a_4)$ can be extracted instead of $G(a_0 \wedge a_1 \rightarrow a_4)$. The first is preferred because it implies the second. The simplification of the sequence

of input propositions *in* is performed by the function *SIMPLIFY* (lines 10-33). Given an input propositions q belonging to the sequence *in*, it makes a copy s_q of q (line 14), and then it performs the following steps for each atomic proposition a_i included in s_q (lines 16-31):

- remove a_i from s_q (line 17);
- create a new sequence of propositions new_in from *in* by replacing the proposition q with s_q (line-18);
- check, for every simulation instant t_i, if the new sequence new_in is true on the atomic proposition trace (line-22), but the output proposition p is false (line 24). In this case, a counter example is found that shows we cannot remove the atomic proposition a_i from s_q, otherwise the association between s_q and p is not valid any more. Thus a_i is restored inside s_q (line 25). If a counter example is not found, a_i can be definitely removed.

6 Assertion Qualification

The degree of interestingness of assertions extracted by applying the methodology described in the previous section is evaluated according to a re-adaptation of metrics that are traditionally adopted in the context of data mining. Assertions are then ranked according to such a metrics.

6.1 Metrics

Several metrics have been proposed in data mining for evaluating the interestingness of association rules. The use of metrics allows analysers evaluating the rules from different points of view [17, 28]. For instance, *odds ratio* and *entropy* are appropriate for estimating the probability of distribution of items, *support* and *confidence* are able to calculate the interestingness of an association rule based on the number of item's occurrences; while the *correlation coefficient* is suited to determine the dependency between set of items.

In the context of assertion qualification, metrics that provide information about the degree of accuracy of a rules with respect to the probability it will hold in the future (like for example, confidence, which estimates the joint probability between occurrences of the antecedent and the consequent in the data set) are not relevant, because we know that assertions under analysis are always true on the DUV. We are instead interested in metrics that measure the interestingness of an assertion with respect to covered behaviours, number of activations, and correlation between antecedents and consequents. For this reason, we identified *support* and *correlation coefficient* as the most interesting metrics for assertion evaluation. Their definition in the context of data mining are hereafter reported together with considerations related to how they can be adapted to be suited for assertion evaluation.

Definition 8 *Given a set of items I, and the corresponding set of transactions D, a rule $X \rightarrow Y$ has support S if X and Y occur concurrently in S percent of transactions in D.*

In practice, to compute the support of an association rule, it is necessary to count how many rows in the transaction set table contain both X and Y. In case of temporal assertions, the support corresponds instead to the number of times a temporal assertion occurs (i.e., its antecedent is fired and then its consequent is satisfied) in the execution traces with respect to the total number of occurrences corresponding to the other temporal assertions under analysis. This requires a different computation approach with respect to data mining. For example, let us consider a temporal assertion $A \rightarrow C$ that occurs 10 times in a set of execution traces. If it belongs to a set of temporal assertions that globally occur 1000 times in the same execution traces, the support of $A \rightarrow C$ is $10/1000 = 0.01$.

Definition 9 *Given a set of items I, and the corresponding set of transactions D, the correlation coefficient of the rule $X \rightarrow Y$ is the covariance of X and Y divided by the product of their individual standard deviations.*

More informally, the correlation coefficient can determine if antecedent and consequent are related or not by observing whether occurrences of the antecedent depend on occurrences of the consequent and vice versa. For example, Figure 4 graphically shows the meaning of the correlation coefficient with respect to the association rule $X \rightarrow Y$. On the left, X and Y has a positive correlation, i.e., an increment in occurrences of X corresponds to an increment in occurrences of Y. In the middle, a negative correlation is shown. Finally, on the right, no dependence between X and Y exists. Higher is the correlation coefficient higher is the interestingness of the analysed rule.

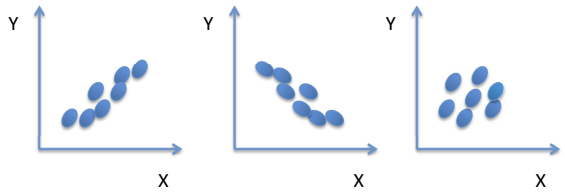


Fig 4: The correlation coefficient: positive correlation (on the left), negative correlation (in the middle), no correlation (on the right).

6.2 Assertion Ranking

For estimating the interestingness of assertions, we implemented an assertion ranker based on support and correlation coefficient. The work flow of the proposed methodology is then divided in 3 main steps (Figure 5):

1. *Counting of occurrences*: In this phase, the number of times an assertion is verified in the execution traces is computed. Then, each assertion is decomposed in antecedent and consequent and their respective frequencies in the execution traces are computed too.

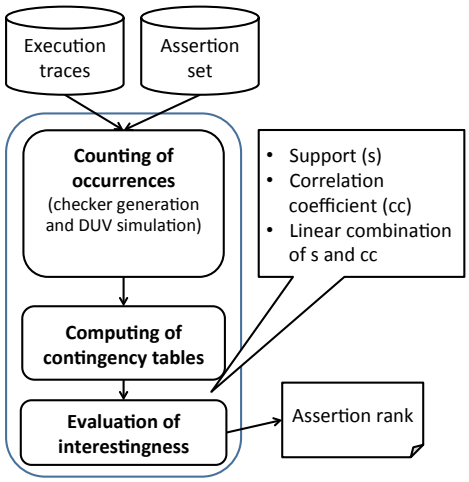


Fig 5: Overview of qualification methodology.

2. *Computation of contingency tables*: the information collected in step 1 is then organized in contingency tables (one per each assertion) that represent the ingredients for the computation of the evaluation metrics in the final step. Contingency tables make simpler the extraction of information like how many times an antecedent and the corresponding consequent occur in the execution trace, how many times an antecedent occurs but the corresponding consequent does not, and how many times a consequent occurs but the corresponding antecedent does not.
3. *Evaluation of interestingness*: The final step, starting from the contingency tables, computes support, correlation coefficient, and their linear combination to obtain a final metrics that considers both of them. Their combination is necessary because support and correlation coefficient separately may provide very different estimations, which only partially characterise the quality of each assertion, as clarified later in this section.

In the following of this section the three steps of the proposed methodology are described.

Counting of Occurrences To count occurrences of assertions, antecedents and consequents, we generate a checker for each assertion. A checker can be considered an automaton that monitors the evolution of the DUV during simulation and raises a failure when the corresponding assertion is violated [29]. To perform such a verification, the checker exactly knows when the antecedent is fired and when the consequent is then satisfied. Thus, it can be used for counting of occurrences as required for our estimation.

For example, the automaton generated for counting occurrences for an assertion like $always(p_a \rightarrow next(p_c))$ is reported in Figure 6. The automata starts in

the initial state *ant*. It remains in this state (corresponding to a vacuous satisfaction of the assertion) till the antecedent p_a is finally fired (transition $T3$). Then, it moves to the state *cons*, where it stays by continuously traversing $T4$ at each simulation instant while p_a remains true and p_c is also satisfied. This represents the case in which the assertion is activated and satisfied (non vacuously) for several consecutive simulation instants. The assertion is non vacuously satisfied also when the automaton exits *cons* by traversing $T7$, which corresponds to the case p_c still holds but p_a stops to be fired. Alternatively, the automaton exits *cons* to reach the *error* state through $T5$ in case p_c stops holding. In this case the assertion is falsified, but according to our assumption (assertions are true in the DUV) this never happens in our methodology. The number of occurrences of the assertion corresponds to the number of traversals of transitions $T4$ and $T7$. The number of times the antecedent is fired corresponds to the number of traversals of $T3$ and $T4$. Finally, the number of times the consequent is fired corresponds to the number of traversals of $T1$ and $T4$.

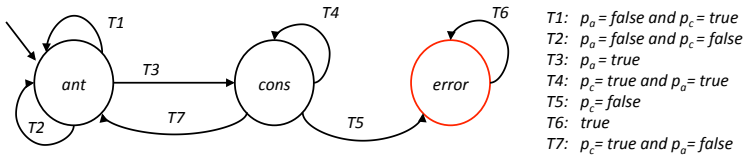


Fig 6: Example of the checker for assertion $always(p_a \rightarrow next(p_c))$.

Computation of Contingency Tables Support and correlation coefficient can be effectively computed by relying on a 2×2 frequency count matrix called contingency table [30], whose computation derived from the counting of occurrences performed in the previous step. Given an assertion $A \rightarrow C$, its contingency table represents the relation between A and C. The cells of the table contain the following information (Table 1):

- Cell f_{11} is the number of times where A is true and consequently C is true in the execution traces;
- Cell f_{10} is the number of times where A is true but consequently C is false and other consequents than C are true in the execution traces, i.e., it is the sum of occurrences of assertions $A \rightarrow C'$ included in the considered assertion set with $C \neq C'$. It is worth noting that $A \rightarrow C$ and $A \rightarrow C'$ are not inconsistent, because C and C' refer to different temporal instants. For example, $always(p_1 \rightarrow next(p_2))$ and $always(p_1 \rightarrow next(next(p_3)))$ can be both true for the same DUV.
- Cell f_{01} is the dual of f_{10}, i.e., it is the number of times where A is false but A' different from A is true and consequently C is true in the execution traces, i.e., it is the sum of occurrences of assertions $A' \rightarrow C$ included in the considered assertion set with $A \neq A'$. In this case, A and A' can also be conflicting

	C	$\bar{C}$	
A	f_{11}	f_{10}	f_{1X}
$\bar{A}$	f_{01}	f_{00}	f_{0X}
	f_{X1}	f_{X0}	f_{XX}

Table 1: Contingency table for $A \rightarrow C$.

$AssertionID$	$Assertion$	$Occurrence$
A1	$always(A \rightarrow A\ until\ F)$	468
A2	$always(B \rightarrow B\ until\ G)$	436
A3	$always(C \rightarrow C\ until\ H)$	481
A4	$always(D \rightarrow D\ until\ I)$	361
A5	$always(E \rightarrow next(J))$	524
A6	$always(E \rightarrow next[2](J))$	516
A7	$always(E \rightarrow next[3](J))$	509

Table 2: An assertion set with the corresponding number of occurrences in the execution traces.

because this doest not represent an inconsistency for the assertion set. For example, $always(p_1 \rightarrow next(p_2))$ and $always(p_3\ until\ p_4 \rightarrow next(p_2))$ can be both true for the same DUV.

- Cell f_{00} is the number of times an assertion is true, whose antecedent and consequent are both different, respectively, from A and C, i.e., it is the sum of occurrences of the other assertions included in the analysed set.
- Cell f_{1X} is the sum of cells f_{11} and f_{10}.
- Cell f_{0X} is the sum of cells f_{01} and f_{00}.
- Cell f_{X1} is the sum of cells f_{11} and f_{01}.
- Cell f_{X0} is the sum of cells f_{10} and f_{00}.
- Cell f_{XX} is the grand total.

As an illustrative example, let us consider assertions reported in Table 2. For sake of clearness, and without loss of generality, the table does not show the atomic propositions composing antecedents and consequents of assertions, but only the temporal relations between them in PSL syntax [3]. The corresponding contingency tables are reported in Table 3. For example, for assertion $A1$, f_{11} correspond to the total number of occurrences of $A1$ in the analysed execution traces; f_{10} is equal to 0, since antecedent A does not appear in none of the other assertions; f_{01} is 0 since consequent $A\ until\ F$ does not appear in none of the other assertions; and finally, f_{00} is obtained by summing the occurrences of all the other assertions except $A1$. Cells f_{10} for assertions $A5$, $A6$ and $A7$ are not zero since they share the same antecedent E. Thus, f_{10} for $A5$, $A6$ and $A7$ are, respectively, the sum of occurrences of $A6$ and $A7$, $A5$ and $A7$, and $A5$ and $A6$. Similar considerations allow computing values for all the other cells of Table 3.

$AssertionID$	f_{11}	f_{10}	f_{01}	f_{00}
$A1$	468	0	0	2827
$A2$	436	0	0	2859
$A3$	481	0	0	2814
$A4$	361	0	0	2934
$A5$	524	1025	0	1746
$A6$	516	1033	0	1746
$A7$	509	1040	0	1746

Table 3: Contingency tables of assertions reported in Table 2.

Evaluation of Interestingness Contingency tables provide basic ingredients for the computation of support and correlation coefficient of a temporal assertion. Concerning support, according to Definition 8, it is simply computed with the following formula:

$$s = \frac{f_{11}}{f_{XX}}. \tag{1}$$

The computation of the correlation coefficient for an assertion $A \rightarrow C$, according with Definition 9, is obtained instead by means of the following formula:

$$\rho = \frac{cov(A, C)}{\sigma A \cdot \sigma C} \tag{2}$$

where $cov(A, C)$ is the covariance of A and C, while σA and σC are the standard deviation, respectively, of A and C. Disregarding mathematical steps, the correlation coefficient can be computed in terms of the cells of a contingency table as follows:

$$\rho = \frac{f_{11} \cdot f_{00} - f_{10} \cdot f_{01}}{\sqrt{f_{1X} \cdot f_{0X} \cdot f_{X1} \cdot f_{X0}}} \tag{3}$$

According to equation (1) the support ranks in the highest positions assertions that occur frequently in the execution traces. However, we can have very interesting assertions that occur a few times because they refer to corner cases. On the other hand, the correlation coefficient privileges assertions where the number of occurrences of the antecedent better matches the number of occurrences of the consequent, but assertions where these numbers are low could be extracted by chance without representing a real behaviour of the DUV. For this reason a combination of support and correlation coefficient provides a more accurate estimation of assertion interestingness. Thus, we propose the measure the interestingness of an assertion A through the following formula:

$$I(A) = \alpha * s_n(A) + (1 - \alpha) * \rho_n(A) \tag{4}$$

where, $\alpha \in [0, 1]$, and $s_n(A)$ and $\rho_n(A)$ are the value obtained by normalizing , respectively, the support s and the correlation coefficient ρ of A with respect to the whole set of analysed assertions. At varying of α the role of support becomes more or less important with respect to the role of the correlation coefficient in

determining the final estimation of assertion interestingness. In our experiments best results have been obtained with $\alpha = 0.4$.

7 Experimental Results

Experimental results have been carried out on an Intel Xeon E5649 @2.53Ghz equipped with 8 GB of RAM and running Linux OS. The benchmarks considered for evaluating the proposed mining strategy belong to the Open-Source-Test-Case (OSTC) platform developed as reference case study for the European project SMAC [31]. In particular, we considered the RTL implementation of the *UART* [32] and *BUS-APB* [33] components. These two benchmarks have been selected because they present different characteristics from the input/output latency point of view, i.e. the number of clock cycles required, at maximum, to compute the component's functionality. The I/O latency is an important parameter for mining approaches because longer is the I/O latency, higher is the time spent by the miner to create an assertion that puts in relation values provided to primary inputs with values obtained on primary outputs. *UART*, which is practically a parallel-to-serial/serial-to-parallel converter, requires 665 clock cycles before the output bit stream is produced, once data are provided in input for the conversion. On the contrary, the input/output latency of *BUS-APB* is 2 clock cycles.

Table 4 reports, for each component, the lines of code (*Lines*), the number of bits corresponding to control signals belonging to the primary inputs (*PIs*) and to the primary outputs (*POs*), and the input/output latency (*I/O latency*). Execution traces composed of 10,000 clock cycles have been generated for the two benchmarks by simulation.

The mining methodology proposed in this paper has been compared with a state-of-the-art approach presented in [5], which mines assertions from execution traces through an induction algorithm based on a decision tree [34]. The comparison between the two approaches is reported in Table 5 and Table 6 concerning, respectively the characteristics of the mined assertions and mining execution times, and the quality of the mined assertions measured in terms of mutation coverage [16].

Columns 2 and 3 of Table 5 report the configuration parameters, i.e., the length of considered time windows (tw_len) (which corresponds to the I/O latency of the DUV), the maximum number of propositions allowed in the antecedent of the mined assertions (max_len) for the time window approach (i.e., the maximum number of clock cycles that are observed in the antecedent), and the maximum depth of the analysed decision tree (max_depth) for the approach proposed in [5]. The parameters max_len has been selected according to the characteristics of the DUVs. For example, $max\_len = 1$ for *UART* because the values assigned to the input control signals to start the data elaboration are provided in a single clock cycle, while $max\_len = 2$ for *BUS-APB* since input control signals influence the bus functionality during the whole elaboration phase that always embraces 2 clock cycles. On the contrary, for the decision-tree

DUV	*Lines*	*PIs*	*POs*	*I/O latency*
BUS-APB	390	6	12	2
UART	6819	10	9	665

Table 4: Characteristics of benchmarks.

Time window-based approach

	Configuration parameters		*Results*			
DUV	*tw_len*	*max_len*	*# ass.*	*# ant.*	*# cons.*	*Time*
BUS-APB	2	2	24	3.3	11.1	1 s.
UART	655	1	21	2.94	6.47	720 s.

Decision tree-based approach [5]

	Configuration parameters		*Results*			
DUV	*tw_len*	*max_depth*	*# ass.*	*# ant.*	*# cons.*	*Time*
BUS-APB	2	12	86	2.82	1	1 s.
UART	665	10	39	5.6	1	5820 s.

Table 5: Number of assertion extraced by the time-window approach and the decision-tree based approach.

based approach the maximum depth of analysed decision tree must be specified; we tested different values and we saw that for values higher than 10 and 12, respectively for *UART* and *BUS-APB*, the execution time of the algorithm increases without improving the quality (measured in terms of mutation coverage) of the mined assertions. Then, Columns 4-7 report the mining results, i.e, the number of extracted assertions (*# ass.*), the average number of input atomic propositions included in the antecedent of the extracted assertions (*# ant.*), the average number of output atomic propositions included in the consequent of the extracted assertions (*# cons.*), and the total time required for the mining procedure (*time*). Looking at the results, we see that the number of assertions generated by our approach is smaller than the number of assertions generated by [5]. However our assertions are composed of consequents with a higher number of atomic propositions, which reflects in a better description of the behaviours of the primary outputs of the DUV when an antecedent is fired. On the contrary, antecedents are generally compact (i.e., the number of involved atomic propositions is small), thus assertions cover a large number of behaviours from the perspective of the DUV's primary inputs. Finally, concerning execution time, our approach outperforms the decision tree-based algorithm when applied to mine assertions on DUVs, whose I/O latency (which impacts on the offset between antecedent and consequent) is very high, like in the case of *UART*.

Time window-based approach

DUV	*# observ.*	*# covered*	*Avg*	*Time*
BUS-APB	22	22	10.27	70 s.
UART	149	99	26.85	4208 s.

Decision tree-based approach [5]

DUV	*# observ.*	*# covered*	*Avg*	*Time*
BUS-APB	22	22	0.8	83 s.
UART	149	58	9.08	46853 s.

Table 6: Comparison between the proposed approach and [5] based on mutant coverage.

7.1 Assertion Qualification

Mined assertions by adopting our approach and the decision tree-based algorithm have been then compared to evaluate their effectiveness in covering the DUV behaviours. The comparison has been done in terms of *mutant coverage*, which represent small alterations of the DUV's source code that perturb its functionality. A mutant is observable if, in comparison with a mutant-free DUV, its effect is visible as an alteration in the DUV's primary outputs. A mutant is covered by an assertion if the assertion fails when the mutant is observed at primary outputs. The mutant coverage C is then the ratio between covered mutants and observable mutants. Uncovered mutants highlight the incompleteness of the assertions set [16]. The well-known bit coverage fault model have been selected to inject mutants in the control signals of the DUVs [35]. Bit coverage alters, in single fault mode, each bit of the affected signal by fixing its value to 0 (stuck-at 0) or to 1 (stuck-at 1). Table 6 reports the results of the mutation analysis by showing the number of observable mutants (*# observ.*), the number of covered mutants (*# covered*), the average number of mutants covered by each assertion (*avg*), and finally the time required to simulate the OSTC platform connected to the set of checkers[3] corresponding to the assertions mined for *UART* and *BUS-APB* in presence of one mutant (*Time*).

The mutant coverage achieved for *BUS-APB* is 100% for both approaches, while the time window-based approach outperforms the decision-tree algorithm concerning mutant coverage of *UART*. Moreover, the number of mutants covered in average by each assertion mined with our approach is higher. Finally, concerning the simulation time, we observe that checkers corresponding to assertions mined by the decision tree algorithm require a longer simulation time, which greatly increases for assertions that predicate on DUV with a long I/O latency, as in the case of *UART*. We observed in particular, that antecedents of assertions generated according to [5] are composed of atomic propositions that

[3] A checker is an automaton that monitors the evolution of the DUV during simulation and raises a failure when the corresponding assertion is violated. We generated checkers for mined assertion by using IBM FoCs [36].

			Preserved mutants				Loss mutants			
DUV	$\#ass.$	$\#mut.$	$Q4_I$	$Q4_C$	$Q4_I \cup Q3_I$	$Q4_C \cup Q3_C$	$Q2_I \cup Q1_I$	$Q2_C \cup Q1_C$	$Q1_I$	$Q1_C$
UART	21	99	76	73	97	97	2	2	1	2
BUS-APB	24	22	18	NA	22	21	0	1	0	0

Table 7: Comparison between assertion ranking based on metrics I and mutant coverage C.

could be removed, since they do not affect the truth value of the assertions. This drawback is implicit in the use of a decision tree-based data structure, and it depends on the fact that an assertion generated at a leaf node necessarily includes atomic propositions predicating on variables involved in all previous levels of the tree. This leads to create assertions with longer antecedents, whose checkers require longer simulation times. Moreover, simulation times are affected by the total number of assertions which is higher in the case of the decision tree-based algorithm.

Finally, the accuracy of the interestingness estimation measured according to mutant coverage C has been compared with the metrics I we defined in Section 6. The hypothesis is that assertions with the highest mutant coverage are ranked in the highest positions also according to the proposed metrics I.

To experimentally prove the previous hypothesis, after the computation of the metrics I (with $\alpha = 0.4$) and the mutant coverage C, we divided assertions in 4 groups, respectively, $Q1_I, \ldots, Q4_I$ for I, and $Q1_C, \ldots, Q4_C$ for C. The division in groups has been done according with *quartiles* computed on I and C. In this way, the top 25%-ranked assertions with respect to I and C are included, respectively, in $Q4_I$ and $Q4_C$, while the worst 25%-ranked assertions are included in $Q1_I$ and $Q1_C$. Similarly, $Q3_I$ and $Q3_C$ include assertions between the first and the second quartile, while $Q2_I$ and $Q2_C$ include assertions between the second and the third quartile. Then, we analysed the impact of assertions belonging to the different groups in covering mutants. Results are reported, for assertions extracted according to our mining approach, in Table 7. After the DUV name, the second and the third Columns report, respectively, the number of analysed assertions ($\#ass.$) and the number of mutants totally covered by assertions ($\#mut.$). Then, Columns under *Preserved mutants* show how many mutants are still covered by preserving assertions belonging to only $Q4_I$ and only $Q4_C$, and to only $Q4_I \cup Q3_I$ and only $Q4_C \cup Q3_C$. Finally, Columns under *Loss mutants* show how many mutants remain uncovered by removing assertions belonging to $Q2_I \cup Q1_I$ and $Q2_C \cup Q1_C$, and to only $Q1_I$ and only $Q1_C$. It is evident from the results reported in Table 7 that measuring the interestingness of assertions according to the metrics I proposed in this paper ranks in the highest positions assertions that cover the most of mutants, while in the lowest positions remain assertions that very rarely cover mutants not yet covered by better ranked assertions. In this context, the ranking provided by I is even better than the ranking provided by C, since, for example, in the case of UART, 76 mutants are covered by assertions included in $Q4_I$, while only 73 mutants are covered by assertions included in $Q4_C$; on the opposite, only one mutant remains

DUV	*I time*	*I + sim time*	*C Time*
UART	2 s.	4208 s.	26400 s.
BUS-APB	2 s.	70 s.	940 s.

Table 8: Execution time for computing I and C.

uncovered by discarding assertions in $Q1_I$, while 2 mutants remain uncovered by discarding assertions in $Q1_C$.

It is worth noting also that in the case of *BUS-APB*, the number of mutants covered only by assertions belonging to group $Q4_C$ cannot be computed, because due to a particular distribution of covered mutants among assertions, the third quartile correspond exactly to the fourth (i.e., to the maximum number of mutants covered by the assertions with the highest mutant coverage). In particular, this happens because, by chance, 8 assertions on 24 cover the same (highest) number of mutants. In this situation, due to the low variability of mutant coverage among assertions there is no distinction between $Q3_C$ and $Q4_C$. This represents a drawback of the mutant-based analysis, which is instead outcome by the approach proposed in this paper that can effectively distinguish between $Q3_I$ and $Q4_I$. A further analysis has been conducted by measuring the time required for the computation of I and C.

Results are reported in Table 8. It is evident that measuring I (*I time*) requires a few seconds, independently from the complexity of the DUV. On the contrary, mutation analysis requires a longer verification time I (*C time*) to simulate DUV and checkers for each mutant. This is particularly evident for complex designs like *UART*, where assertions predicate on large time windows (up to 665 clock cycles). For sake of clarity, the time reported for I does not include the time spent for counting assertion occurrences in the execution traces, since the result of such a counting is already available when assertions are automatically generated through assertions mining. If this information was not available, or assertions were manually defined, the time for computing I would include the time spent for one simulation run to compute assertion occurrences on the execution traces (I + *sim time*), while computation of C always requires a number of simulation runs equal to the number of mutants.

From previous considerations we derive that the evaluation of the assertions according to the methodology proposed in Section 6 represents a faster and more effective approach for assertion qualification with respect to measuring the quality of assertions by using a mutant coverage-based approach.

8 Conclusions

The paper presents a mining algorithm for automatic generation of LTL temporal assertions and a qualification metric for the evaluation of the assertion interestingness.

On one side, the assertions generation technique relies on a time window-based analysis of execution traces that searches for behaviours that repeat peri-

odically capturing the relation between primary inputs and primary outputs of the DUV. The approach is particularly suited for mining assertions that describe the behaviour of the control signals of the DUV, which are used to implement the I/O communication protocol surrounding the computation of the DUV core functionality. In comparison with a state-of-the-art assertion miner proposed in [5], experimental results show that our approach generates a more compact set of assertions, which achieves a higher mutant coverage and requires shorter times for the simulation of the corresponding checkers.

On the other hand, the qualification approach re-adapts metrics typically adopted in data mining, i.e., support and correlation coefficient, to measure the importance of an assertion on the basis of both its activation frequency during simulation runs and the correlation between its antecedent and consequent. Experimental results showed that, compared to traditional mutant coverage-based techniques, our metrics provides a better estimation of assertion interestingness by ranking in the top positions assertions that cover the major number of mutants and in the lowest positions assertions that cover mutants detected also by better ranked assertions. Finally, concerning estimation time, we outperform the mutant coverage-based approach of one order of magnitude, by considering also the time required for computing the frequency of assertions by simulation. When such frequencies are already available (e.g., when provided by an assertion mining tool) the computation of the proposed metrics is almost negligible (a few seconds).

References

1. A. Gupta, "Assertion-based verification turns the corner." *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 131–132, 2002.
2. A. Pnueli, "Linear and branching structures in the semantics and logics of reactive systems," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, W. Brauer, Ed. Springer Berlin Heidelberg, 1985, vol. 194, pp. 15–32.
3. "Standard for property specification language (PSL)," *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pp. 1–184, 2012.
4. G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.
5. S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 32, no. 6, pp. 952–965, 2013.
6. A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *In proc. of ACM/IEEE DATE*, 2015.
7. "Jasper Activeprop," http://www.jasper-da.com.
8. http://www.atrenta.com/solutions/bugscope.htm5.
9. A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Proc. of ACM/IEEE DATE*, 2015.
10. S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertion with guidance from static analysis," *IEEE Trans. on CAD*, vol. 32, no. 6, pp. 952–965, 2013.

11. M. Bertasi, G. Di Guglielmo, and G. Pravadelli, "Automatic generation of compact formal properties for effective error detection," in *Proc. of ACM/IEEE CODES+ISSS*, 2013, pp. 1–10.
12. W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. of ACM/IEEE DAC*, 2010.
13. S. Katz, O. Grumberg, and D. Geist, "Have I written enough properties? — A method of comparison between specification and implementation," in *Proc. of ACM CHARME*, 1999, pp. 280–297.
14. H. Hoskote, T. Kam, P. H. Ho, and X. Zao, "Coverage estimation for symbolic model checking," in *Proc. of ACM/IEEE DAC*, 1999, pp. 300–305.
15. N. Jayakumar, M. Purandare, and F. Somenzi, "Dos and don'ts of CTL state coverage estimation," in *Proc. of ACM/IEEE DAC*, 2003, pp. 292–295.
16. A. Fedeli, F. Fummi, and G. Pravadelli, "Properties incompleteness evaluation by functional verification," *IEEE Trans. Comput.*, vol. 56, no. 4, pp. 528–544, 2007.
17. P.-N. Tan, V. Kumar, and J. Srivastava, "Selecting the right interestingness measure for association patterns," in *Proc. of ACM/SIGKDD KDD*, 2002, pp. 32–41.
18. P.-N. Tan and V. Kumar, "Interestingness measures for association patterns: A perspective," in *Proc. of Workshop on Postprocessing in Machine Learning and Data Mining*, 2000.
19. D. Lo and S. Maoz, "Specification mining of symbolic scenario-based models," in *Proc. of ACM PASTE*, 2008, pp. 29–35.
20. D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *Proc. of ACM KDD*, 2007, pp. 460–469.
21. J. Henkel and A. Diwan, "Discovering algebraic specifications from java classes," in *Proc. of ECOOP*, 2003, pp. 431–456.
22. M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.
23. D. Sheridan, L. Liu, H. Kim, and S. Vasudevan, "A coverage guided mining approach for automatic generation of succinct assertions," in *Proc. of IEEE VLSI Design*, 2014, pp. 68–73.
24. M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli, "Dynamic property mining for embedded software," in *Proc. of ACM/IEEE CODES+ISSS*, 2012, pp. 187–196.
25. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.
26. http://www.atrenta.com/about-bugscope.htm5.
27. C. M. Antunes and A. L. Oliveira, "Temporal data mining: An overview," in *Proc. of Workshop on Temporal Data Mining*, 2001.
28. R. J. Bayardo Jr and R. Agrawal, "Mining the most interesting rules," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 1999, pp. 145–154.
29. M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring.* Springer Publishing Company, Incorporated, 2008.
30. K. Pearson and L. N. G. Filon, "Mathematical contributions to the theory of evolution. IV. on the probable errors of frequency constants and on the influence of random selection on variation and correlation," *Philosophical Transactions*, vol. 191, pp. 229–311, 1898.

31. http://www.fp7-smac.org.
32. http://opencores.org/project,a_vhd_16550_uart.
33. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.
34. J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.
35. A. Fin, F. Fummi, and G. Pravadelli, "Amleto: A multi-language environment for functional test generation," in *Proc. of IEEE ITC*, 2001, pp. 821–829.
36. https://www.research.ibm.com/haifa/projects/verification/focs/.