Journal of Electronic Testing (2024) 40:199–214
https://doi.org/10.1007/s10836-024-06115-9

Syntactic and Semantic Analysis of Temporal Assertions to Support the Approximation of RTL Designs

Alberto Bosio[1] · **Samuele Germiniani**[2] · **Graziano Pravadelli**[2] · **Marcello Traiola**[3]

Received: 30 July 2023 / Accepted: 6 March 2024 / Published online: 23 April 2024

Abstract
Approximate Computing (AxC) aims at optimizing the hardware resources in terms of area and power consumption at the cost of a reasonable degradation in computation accuracy. Several design exploration approaches and metrics have been proposed so far to identify the approximation targets, but only a few of them exploit information derived from assertion-based verification (ABV). In this paper we propose an ABV methodology to guide the AxC design exploration of RTL descriptions; we consider two main approximation techniques: bit-width and statement reduction. Assertions are automatically mined from the simulation traces of the original design to capture the golden behaviours. Then, we consider the syntactic and semantic aspects of the assertions to rank the approximation targets. The proposed methodology generates a list of statements sorted by their increasing impact on altering the functional correctness of the original design, when selected to be approximated. Through experiments on a case study, we show that the proposed approach represents a promising solution toward the automation of AxC design exploration at RTL.

Keywords Approximate computing · Assertion-based verification · Assertion mining · Fault injection

1 Introduction

Approximate Computing (AxC) paradigm was proposed to improve performance while lowering area and power overheads compared to classical computing systems at the cost of a degraded, but still acceptable, output accuracy [4]. During past years, AxC has been widely adopted since many popular applications we use, such as digital signal processing of images or audio, data analytics, machine learning, web search and wireless communications shown an inherent resiliency to input noise or errors due to imprecise computation [5, 8, 15, 22, 24]. AxC has the potential to be implemented at various abstraction levels within a computing system, ranging from circuits to algorithms [4]. However, this vast design exploration space becomes a significant bottleneck when deploying AxC.

Indeed, several works have been proposed so far to automatically trade-off between output accuracy and performances [19]. At HW level, most of previous works lack the capability to identify resilient elements (e.g, HW component, HDL statements, etc.) of the design to be approximated. Existing approaches generate approximate variants of the Design Under Exploration (DUE). Every variant is then executed/simulated in order to determine the accuracy degradation [2]. The problem is that the accuracy depends on the application, and thus it requires a specific metric to be computed (e.g., similarity index, hamming distance, etc.). Consequently, exploring the design for AxC generally results in a long and tedious procedure.

Recently, the authors of [18] have presented a method to automatically identify resilient elements of the DUE as

Responsible Editor: L.M.B Poehls

✉ Alberto Bosio
alberto.bosio@ec-lyon.fr

Samuele Germiniani
samuele.germiniani@univr.it

Graziano Pravadelli
graziano.pravadelli@univr.it

Marcello Traiola
marcello.traiola@inria.fr

[1] Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, Lyon, France

[2] Department of Engineering for Innovation Medicine, University of Verona, Verona, Italy

[3] University of Rennes, Inria, CNRS, IRISA, UMR6074, Rennes, France

candidates for approximation. The evaluation of the accuracy degradation leverages on the use of assertions, where an assertion is a logic formula that captures a specific functional behaviour implemented in the design [6]. Therefore, assertions can measure how much the approximated design alters the functionality. In other words using assertion to estimate the output accuracy replace the use of functional metrics. On the other hand, assertions have never been used to identify the DUE resilient elements (i.e., HDL statements).

Verification engineers have been employing for decades assertion-based verification (ABV) techniques to verify the functional correctness of a design w.r.t. its formal specifications. The ABV can be described in two phase: 1) temporal assertions, i.e., logics formulas written by means of a temporal logic like, for example, Linear Time Logic (LTL) or Computation Tree Logic (CTL) are identified to express correct functional behaviours. 2) Temporal assertions are verified either statically by model checking, or dynamically by synthesizing assertion checkers that verify if the corresponding assertions are true or not during the simulation of the design.

The contribution of this work is a methodology for guiding the approximation of the DUE leveraging on ABV with two different exploration approaches exploiting the syntactic and semantic aspects of the assertions. In both approaches, assertions are automatically generated from the original implementation of the DUE to capture its functionalities. Furthermore, we consider two main approximation techniques: bit-width and statement reduction [4].

The first approach analyses the syntactic structure of the assertions and the behavioral description of the DUE itself. In particular, it identifies design statements that can have an impact on the evaluation of the assertions. The analysis is based on the syntactic study of the assertions and their activation frequency. Moreover, the impact is computed by leveraging on the distance between variables belonging to a statement and the primary outputs of the DUE. The distance is computed on the Variable Dependency Graph (VDG) and corresponds to the minimum number of arcs connecting them.

The second approach analyses the semantic aspect of the assertions. In particular, it exploits the combination of fault injection and ABV. Fault injection is used to mimic the approximation effect on the DUE, and assertions are then re-evaluated on the faulty design (i.e. representing the approximated design) to analyse the variations of their truth values with respect to the original implementation. Such variations are then used to rank the different approximation alternatives, according to their estimated impact on the functionality of the target design.

In both the above approaches, the output of a list of DUE elements (either bits of signals/registers or statements) ranked by increasing levels of severity: the higher the severity, the higher the output accuracy degradation. Moreover, the second approach introduces a clustering procedure to identify approximations that can be applied simultaneously in order to maximise the overheads reduction.

The produced list of targets is further used to apply approximation. Designers can start the approximation from the top-ranked statements (i.e., having the lowest severity) and stop as soon as either they have reached the desired saving in terms of overheads, or the degradation of the design functionality exceed a given threshold.

The authors already proposed a work on this topic in [3]. Differently from [3], in this paper, we improve the methodology by analysing the semantic aspect of the assertions.

The rest of the paper is organised as follows. Section 2 presents the state of the art; Section 3 gives the basics to understand the technical parts of the paper; Section 4 presents our methodology; Section 8 details the results obtained on the case study; finally, in Section 9 we draw our conclusions.

2 Related Work

As stated before, one of the most challenging problems in AxC is the identification of resilient portion of the application to be approximated. The portion depends on the abstraction level: at software level it can be an instruction of the source code, while at hardware level it can be an HDL statement or signal. Previous approaches allow the designer to explicitly mark resilient portions and specify how to apply approximation (i.e., which technique). For example, a programmer can annotate through pragmas that a given loop has to be approximated by applying the loop perforation technique [23]. Without annotations, each code line has to be considered as a potential approximation target, thus leading to virtually infinite possibilities of approximations.

At the hardware level, ABACUS [21] directly works with RTL implementations (i.e., HDL code). The proposed design-space exploration leverages a greedy algorithm to find a trade-off between output accuracy and power consumption. In [1, 2], design space exploration exploits a genetic approach. Other approaches manually identify resilient portions of a design, mainly focusing on arithmetic components [12, 14, 20]. Recently, authors of [17] proposed WOAxC, a workload-aware framework to automatically select pre-existing approximate units (adders and multipliers), minimizing energy consumption while meeting the quality requirements of the application.

Since quality requirements may change at run-time, authors of [28] propose an automated framework for the design of dynamically reconfigurable circuits. The framework modifies the circuit netlist by replacing the wires with *approximate switches* to reduce circuits' dynamic power consumption through a reduction of the switching-activity.

Although, the added circuitry increases the static power consumption, a high reduction in total power consumption is achieved since the dynamic power is predominant in Application-Specific Integrated Circuit (ASIC) technologies considered by the authors. Unfortunately, the approach is tightly coupled with the target technology, hence it is not guaranteed that it could be used effectively on a different technology.

A different approach is presented in [13], where authors take into account information about the underlying hardware architecture, such as the use of over-scaling techniques.

A recent work claims approximate computing cannot be fully exploited by only considering hardware or software; therefore, hardware/software co-design should be considered in order to achieve better trade-off between accuracy and efficiency, in machine learning applications in particular [10].

On the other hand, even if they aim at different goals, different approaches have been proposed for applying verification techniques to approximate computing. In [26], the authors present an AxC-based approach to achieve a fast and accurate enough repeated execution for security verification. They implement and evaluate the approximate computing-based security verification framework by conducting a case study on a CPU-FPGA based video motion detection system. The Authors in [11] present a novel approach for determining under what conditions a software verification result is valid when the software is executed on approximate hardware. To this end, they compute the allowed tolerances for AxC hardware from successful verification runs. More precisely, they derive a set of constraints which-when met by the AxC hardware-guarantees the verification result to carry over to AxC.

In [27], the authors propose a dynamic verification methodology to assess the quality of the approximated circuit through test patterns mutations and coverage information. However, to the best of our knowledge, none of the exsiting verification-oriented methodologies proposed to use assertions to guide the approximate design exploration. In [18], the authors introduce a verification-guided method to automatically identify program blocks suited for approximation, while ensuring "good enough" program correctness. The idea is to identify code sections that are less impacting on the computation of the program outputs and therefore, more approximable. In particular, they generate a statement ranking based on whether an instruction affects a particular primary output of the designs. Assertions are then used to evaluate the impact of the approximation on the functional behaviour of the final design. It is important to note that assertions are not directly used for statements identification (i.e., identify which portion of the code has to be approximated) as we propose in this paper.

On the contrary, in this work, we thoroughly explore the use of assertions to perform an effective design space exploration. Specifically, we syntactically analyse the characteristics of a set of assertions to estimate the approximability of the approximation targets in the DUE; the analysis is based on the syntax tree and the activation frequency of the assertions. Additionally, we exploit the concept of "distance between variables" to measure the influence on the outputs of the variables in the DUE. Finally, we analyse the semantics of the assertions by employing fault injection analysis.

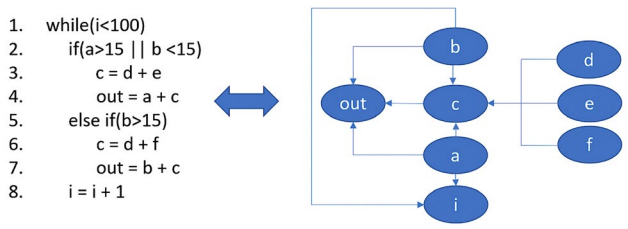

Fig. 1 VDG Example

3 Preliminary Definitions

We report hereafter the definitions of concepts used in the rest of the paper.

Definition 1 Let v_1 and v_2 be two variables belonging to a statement s, v_2 is **data dependent** on v_1 if v_2 belong to the left-hand side of s and v_1 belong to the right-hand side of s.

In the running example of Fig. 1, the statement at line 3 contains variable "c" (left-hand side) which is data dependent on variables "d" and "e" (right-hand side).

Definition 2 Let v_1 and v_2 be two variables and c a conditional statement, v_2 is **control dependent** on v_1 if:

- v_2 is part of the condition of c;
- v_1 belongs to the left-hand side of a statement included in the scope of c.

In the running example, the statement at line 3 contains variable "c" which is control dependent on variable "a" and "b" which are part of the condition of the if statement at line 2.

Definition 3 A **Variable Dependency Graph** (VDG) is a data structure composed of nodes and edges where each node represents a variable and each edge represents a data dependency or a control dependency between variables. Let n_1, n_2 be two nodes of a VDG, if n_2 has an incoming edge e_1 connecting n_2 with n_1, then the variable represented by n_2 is either data dependent or control dependent on the variable represented by n_1.

Table 1 Contingency table

	Cons. true	*Cons. false*	*Cons. unknown*
Ant. true	**ATCT**	**ATCF**	**ATCU**
Ant. false	**AFCT**	**AFCF**	**AFCU**
Ant. unknown	**AUCT**	**AUCF**	**AUCU**

A VGD example, which will be used in the rest of the paper, is shown in Fig. 1.

Definition 4 Given a finite sequence of time units $\langle t_1, ..., t_n \rangle$ and a set of variables $\{v^1, ..., v^m\}$, a **trace** is a sequence of tuples $(t_i, v_i^1, ..., v_i^m)$ such that v_i^j is the value assumed by variable v^j at time t_i.

Definition 5 An **assertion** is a LTL formula of the form *always*(*antecedent* → *consequent*).

Figure 3 shows a few examples of such assertions generated by our approach in the running example. In this paper, we make use of well-known LTL operators, such as *Next (X)* and *Always (G)* to exemplify our approach. We kindly refer the reader to [29] for a complete description of LTL operators and semantics according to the Property Specification Language (PSL).

Definition 6 Given a trace *tr* of length *n* and an assertion *as*, an **evaluation** of *as* with respect to *tr* is the sequence of evaluation units $\langle e_1, ..., e_n \rangle$, where e_i is the truth value of the assertion at instant t_i.

Definition 7 An assertion **holds** in a trace if and only if its evaluation does not contain any evaluation unit whose value is false.

Definition 8 Given a trace *tr*, and an assertion *as*, a **contingency table** is a 3×3 matrix displaying the frequency distribution of *true*, *false* and *unknown* evaluation units of the antecedent with respect to the consequent of *as* in *tr*.

For example, in Table 1, ATCT represents the number of time instants in a trace where both the antecedent and the consequent evaluate to *true*.

Definition 9 A **metric** is a numeric formula measuring the impact of an assertion's feature in the assertion ranking.

The more prominent the feature, the higher its impact on the final ranking of the assertion. The elements of the contingency table are examples of features of an assertion.

4 Methodology Overview

Our methodology is intended to provide the designer with an automatic way to explore AxC alternatives on RTL descriptions. Two approximation strategies have been considered:

- **Bit-width reduction:** fixing to a constant value one or more bits on a subset of design signals/registers;
- **Statement reduction:** removing one or more statements.

The overview of the entire methodology is reported in Fig. 2. The input is the RTL description of the DUE with a suitable testbench.[1] The output is a ranked list of DUE elements that can be approximated. In the rest of the paper, we refer to such elements with the term *approximation tokens* (AT). In particular, according to the addressed AxC strategies, we consider the following kinds of ATs:

- **Statement token:** an instruction appearing in the RTL description of the DUE;
- **Bit token:** a bit of a signal/register occurrence appearing in a statement token.

In this paper, we consider only two types of tokens as we want to perform design exploration for two well-known and easy-to-understand approximation techniques. Nonetheless, the methodology is applicable to other types of tokens and corresponding approximation techniques.

The core of the methodology comprises of two main approaches that analyse the syntax and semantics of temporal assertions. Since both approaches require a set of LTL assertions, the procedure starts by dynamically simulating the DUE to generate the golden trace (traces) by simulating the unaltered implementation of the DUE with the provided testbench (testbenches). Then, we employ an assertion miner to generate assertions holding on the golden trace. The assertions predicate on the inputs and outputs of the DUE and capture its golden behaviour.

Hereafter, we provide an overview of the three main steps of the syntax-based and semantic-based methodologies. A detailed description for each of them is then reported in the following sections.

4.1 Overview of the Syntax-based Method

1. **Syntactic data extraction:** in this step, we extract the required data. First, we generate the VDG from the

[1] The testbench affects the quality of the mined assertions, as it happens in any other simulation-based verification approach. It is reasonably to assume that in a simulation-based verification flow a high-quality test set is available at the time assertion mining is executed.

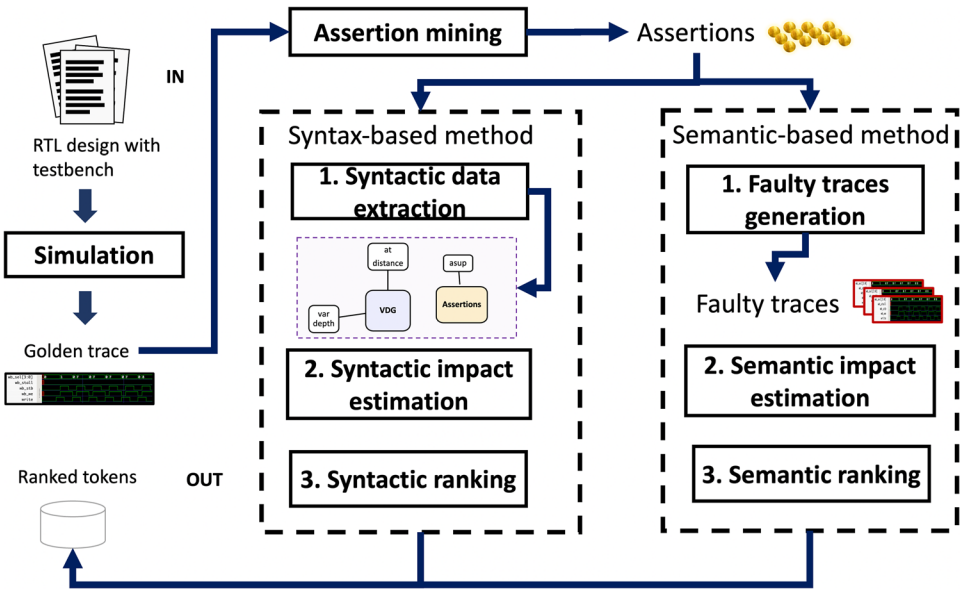

Fig. 2 Overview of the methodology

source code of the golden (original) design. Then, additional syntactic-related features are extracted from the VDG and the assertions.

2. **Syntactic impact estimation:** During this phase, the data obtained from the syntax trees of the extracted assertions and the VDG of the reference design are utilized to calculate a function that estimates the impact of approximating each statement within the design.
3. **Statement ranking:** In the last step, the impact function estimation allows to rank the design. Moreover, statements having similar impact (according to a similarity threshold) are further ranked by exploiting two supplementary procedures: i) a heuristic based on the VDG that considers the distances between the statements and the primary outputs; ii) the execution frequency of statements according to the simulation traces provided as input.

4.2 Overview of the Semantic-based Method

1. **Faulty traces generation:** in the first step of the semantic-based method, we dynamically simulate the DUE approximations to generate a set of execution traces. For each AT, we generate a trace that reflects the effect of activating a corresponding approximation (i.e., bit-width reduction for bit tokens, and statement reduction for statement tokens). These traces are obtained by simulating a faulty version of the DUE. In particular, stuck-at faults are injected in specific elements of the original implementation to exactly mimic the effect of approximating each AT.
2. **Semantic impact estimation:** In the second step, we re-evaluate the mined assertions on the faulty traces obtained in step 1. By comparing the contingency tables of the assertions evaluated on the golden and faulty traces, we automatically estimate the approximability of each single AT, in terms of how its approximation affects the functionality of the original design.
3. **Semantic ranking:** Finally, we rank the ATs in order of decreasing approximability. ATs with similar approximability are clustered.

N	Assertions	S	Causality	Support
1	G((a >= 58 && a <= 60) && (b >= 23 && b <= 63) |-> out >= 81 && out <= 126)	0.98	0.91	0.68
2	G((a >= 4 && a <= 63) && (b >= 61 && b <= 63) |-> out >= 65 && out <= 126)	0.96	0.64	1.00
...	...	...	...	...
38	G((a >= 28 && a <= 29) && (b >= 34 && b <= 35) |-> out >= 64 && out <= 73)	0.1	0.67	0.18

Fig. 3 Assertions mined for the running example of Section 7

5 Assertion Mining

Before diving in to the two approaches, we automatically mine assertions holding on the golden trace of the original DUE implementation. To extract them, we exploit HARM an open-source state-of-the-art assertion miner for LTL properties [7, 9].

The miner has been configured to automatically generate assertions on primary inputs and outputs of the DUE in the form $always(antecedent \rightarrow next[N](consequent))$, where N is the design depth, that is, the number of clock cycles necessary to propagate the effects of primary inputs toward primary outputs. This is done to ensure that the mined assertions represent meaningful I/O relations.

Both the *antecedent* and the *consequent* of each assertion are instantiated by the tool following the form $prop_1$ && $prop_2$ && ... && $prop_k$. Each proposition $prop_i$ is in the form $c_l <= var_j <= c_r$, or $var_j == c$, or $var_j <= c$, or $var_j >= c$, where var_j is an input or an output of the DUE if located, respectively, in the *antecedent* or the *consequent*, while c_l, c_r and c are numeric constants.

Finally, we rank the assertion set according to a score S, which is obtained, for each assertion a, by combining the following metrics (see Definitions 8 and 9):

- Metric 1: $Support(a) = ATCT/traceLength$;
- Metric 2: $Causality(a) = 1 - AFCT/traceLength$;

where, $ATCT$ and $AFCT$ are derived from the contingency table of a (see Definition 8).

Then, the overall score S is obtained, for each assertion a, through the followint formula

$$\prod_{i=1}^{2} calibrate(sm_i(a)/sm_i(a_{max\_i})), \quad (1)$$

where $sm_i(a)$ represents the score of a evaluated by using the metric i, a_{max_i} is the assertion obtaining the maximum score by using metric i, and *calibrate* is a procedure that "calibrates" the input score by using the following modified version of the Richards' curve, ranging from 0 to 1

$$R(x) = 1/(1 + e^{(3.3-10.62x)})^2. \quad (2)$$

The underlying idea of this ranking formula is to enable the utilization of two metrics simultaneously in a unified ranking process. Additionally, it aims to prioritize assertions that achieve higher scores in both sorting metrics, while discouraging assertions that perform well in only one or neither of them. Once the ranking process concludes, we retain only those assertions with a score S greater than 0. These selected assertions then embody the golden behavior of the original design.

Figure 3 shows some of the assertions mined for the running example of Section 7 and the corresponding values for *Support*, *Causality* and S.

In the running example, we generate assertions by providing the basic template $always(antecedent \rightarrow consequent)$ as the inputs are propagated to the outputs on the same clock cycle. The miner generates 383 assertions, although only 38 assertions end up providing a final ranking greater than zero. We report in Fig. 3, 3 of of the 38 generated assertions.

6 Syntax-based Method

In this section, we delve into the details of the 3 steps comprising the syntax-based method.

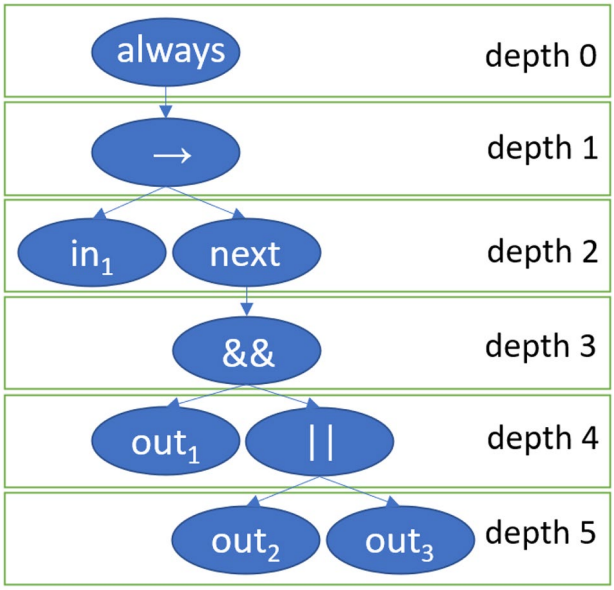

Fig. 4 Syntax tree corresponding to the assertion $always[(in_1) \longrightarrow next(out_1 \&\& (out_2 || out_3))]$

6.1 Syntactic Data Extraction

First, we generate the VDG of the DUE. The primary use of the VDG is to assess the likelihood that the impact of the approximation on a token is masked by other operations before it reaches the primary outputs of the design. After that, the following syntax-related values are extracted from the assertions and the generated VDG.

- The **variable depth in an assertion** (*var_depth*). This is defined as the depth of a variable involved in the consequent of an assertion $a \in A$ from the root of the syntax tree of a. For instance, consider the assertion in Fig. 4. The *always* operator is the root of the corresponding syntax tree located at depth 0; in_1 is situated at depth 2, out_1 at depth 4, while out_2 and out_3 at depth 5. Intuitively, ATs whose variables appear in the deepest part of an assertion consequent should allow a better approximation, in terms of functional accuracy, as their influence on its truth values is generally lower (due to masking effects); hence, their approximation produces a lesser effect on the corresponding functional behaviour of the DUE.
- The **AT distance from primary outputs** (*at_distance*). This is defined as the minimum number of edges separating an AT from an output of the DUE in the corresponding VDG. In the example of Fig. 1, the distance between variable e and output *out* is 2. Intuitively, ATs that are closer to the outputs can greatly influence the

functional behaviour of the DUE; conversely, ATs further from the outputs allow a better approximation, as their influence is less severe. To calculate the AT distance, we must proceed differently depending on whether the corresponding AT is related to an assignment or a conditional statement. If assignment case, the distance is the minimum number of edges separating its left-hand side variable from an output in the VDG. If conditional case, then the corresponding condition may contain multiple variables; the final value corresponds to the distance of the variable belonging to the condition at the shorter distance from one output. In the example of Fig. 1, statement 2 lays at distance 1 from variable *out*.

- The **assertion support** (*asup*). The support of an assertion is determined by counting the occurrences of simulation traces in which both the antecedent and the consequent of the assertion are true, indicating that the assertion is true in a meaningful way (non-vacuously). In simple terms, assertions with low support typically involve variables that have less significant influence on the functional behavior of the DUE. Consequently, if other ATs share the same variables with a low-support assertion, they are more likely to allow a considerable approximation because their effect on the DUE behavior is relatively smaller.

The information derived from the assertions and the VDG are then combined in the next step of this method to estimate the impact of approximation on the DUE behaviours.

6.2 Syntactic Impact Estimation

The evaluation of how approximating a token affects the functionality of the DUE is performed using the *estimateImpact* function, outlined in Algorithm 1. This function takes three inputs: the set of assertions *A* mined by HARM, the set of ATs of the DUE represented as *AT*, and the VDG denoted as *vdg*. The output value produced by the *estimateImpact* function for an AT indicates its undesirability for approximation. In other words, the higher the value returned by *estimateImpact* for a particular AT, the less favorable it is to choose that AT for the approximation process. The impact function exploits "variable depth in an assertion", "the assertion support" and "AT distance from primary outputs" extracted in the previous step.

Algorithm 1 Impact Function

```
1:  function getAssertionImpact(a, at, vdg)
2:      asup ← getSupport(a)
3:      c ← getConsequent(a)
4:      lhs ← getLHS(at)
5:      impact ← 0
6:      for all var in getVars(c) do
7:          var_depth ← getDepth(var, c)
8:          at_distance ← getDistance(var, lhs_var, vdg)
9:          impact+=asup/(var_depth ∗ at_distance)
10:     return impact
11:
12: function estimateImpact(A, AT, vdg)
13:     I ← ∅
14:     for all at in AT do
15:         at_imp ← ∅
16:         for all a in A do
17:             at_imp.push(getAssertionImpact(a, at, vdg))
18:         I.push(avg(at_imp))
19:     return I
```

The function *estimateImpact* consists of two main steps. First, the impact of a token *at* with respect to each assertion is computed and stored in variable *at_imp*. This is achieved by iterating over *A* (line 16) and by retrieving for each tuple (*at*, *a*) the corresponding assertion impact by calling the function *getAssertionImpact* (line 17). The resulting impact is then appended to *at_imp*. After that, the function computes the average of the impacts in *at_imp* and stores the result in *I* (line 18). Finally, the list of final impacts is returned (line 19). Function *getAssertionImpact* returns the impact of token *at* with respect to assertion *a*. First, the function retrieves three accessory elements that are used to compute the partial impact; such elements include the support of the assertion *asup* (line 2), the syntax tree of the assertion's consequent *c* (line 3) and the left-hand side variable of the AT *lhs_var* (line 4). Note that this algorithm is simplified to accept only AT related to assignment statements with a left-hand side variable. This is done to improve readability. Second, the impact is progressively computed for each variable in the assertion's consequent (lines 6-9). For each *var*, the partial impact is calculated through the formula $asup/(var\_depth * at\_distance)$, where *var_depth* and *at_distance* are the aforementioned "variable depth in an assertion" and "a distance from primary outputs". Finally, the computed impact is returned (line 10).

The left part of Fig. 5 shows the estimated impact for each statement token of the example of Fig. 1. The worst-time complexity of function *estimateImpact* is $O(|S| * |A|)$,

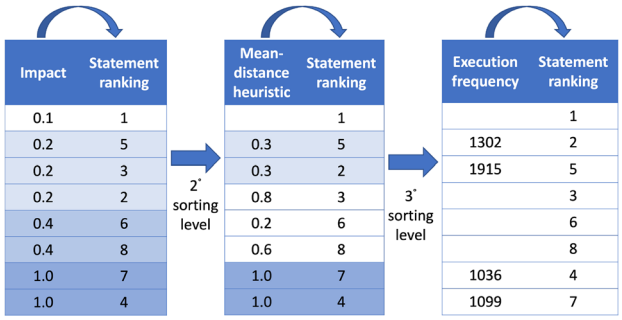

Fig. 5 Ranking for the example of Fig. 1

where |*S*| and |*A*| are the number of analyzed ATs and assertions, respectively. We omitted the cost of calling function *getAssertionImpact*, as in any practical scenario, it would only contain constant-time operations. This is true because the number of variables in each assertion is usually limited.

6.3 Syntactic Ranking

In the last step of syntax-based method, the output of the *estimateImpact* function is used to rank the AT of the DUE to create a *AT approximation list*. That list ranks the AT in descending approximation i.e., designers are suggested to start approximating the tokens on the top of the list and stop as soon as they reach the maximum functional degradation threshold. Therefore, AT with a higher impact on the DUE behaviour will be ranked in the bottom part of the AT approximation list compared to those with a lower impact. Additionally, two auxiliary criteria are employed to order the positions in the list for AT with a similar impact value (i.e., when the difference among impacts is lower than a predefined threshold):

- All tokens with a similar impact are further sorted by using a *mean-distance heuristic* that calculates the average distance of each AT from the design primary outputs. This distance is defined by counting the number of arcs in the path connecting two target nodes in the VDG. AT with a higher average distance are supposed to have a lesser impact on the design functionality when approximated; therefore, among those with similar impact, these are ranked lower than AT with a shorter average distance.
- In the case of AT having similar impact and average distance, we consider the number of times that the AT is executed during the design simulation (by using the same simulation traces adopted for the assertion mining). The intuition is that the higher the number of executions of a AT, the higher is the influence of its approximation on the design functionality.

Let us resort to the example of Fig. 1 and the ranking of the corresponding AT reported in Fig. 5. Initially, the ATs are ranked according to the result of the *estimateImpact* function. This creates 4 clusters of AT with similar impact values $\{(1), (2, 3, 5), (6, 8), (4, 7)\}$, as shown on the left side of Fig. 5. A second sorting level based on the *mean-distance heuristic* is then applied, thus reordering the ATs and reducing the number of AT belonging to unsorted clusters $\{(2, 5), (4, 7)\}$, as reported in the central part of Fig. 5. Finally, the third level of sorting, based on the AT execution frequency, is applied on the two remaining clusters, thus providing the results shown on the right of Fig. 5.

7 Semantic-based Method

In this section, we report the details of the 3 steps comprising the semantic-based method.

To simplify the detailed exposition of each step, in the following sections, we refer to the running example reported in Algorithm 2. It takes as input two 8-bit unsigned integers (signals *a* and *b*) and returns their sum as output (*out* port). We assume that the adder module is stimulated by an external test tench, such that the two least significant bits of signal *a* and *b* (i.e., *a*[7:6] and *b*[7:6]) remain unused.

Algorithm 2 Running example

```
1:  module adder(a, b, clk, out)
2:  input [7:0] a, b
3:  input clk
4:  output [8:0] out
5:  reg [8:0] sum
6:  always @(posedge clk)
7:  begin
8:        sum = a + b
9:  end
10: endmodule
```

7.1 Faulty Traces Generation

In the first step of semantic-based method, we simulate the DUE to generate a set of faulty execution traces. For each target AT, we generate a trace reflecting the effect of its approximation according to either the bit-width reduction or the statement reduction strategy. For each class of AT, we identify a fault model to mimic the effect of its approximation in the functional behaviour of the DUE as follows:

- A bit token is approximated by injecting a stuck-at 0/1 on the target bit. Stuck-at X has been not considered as the propagation of X values along the execution traces would prevent the evaluation of mined assertions in step 3.

- A statement token is approximated differently depending on the type of statement as follows:
  - assignment: the statement is removed; in case its left-hand side remains undefined, its value is assigned to 0 for bit-vectors and numeric types, stuck at 0/1 for a single bit/Boolean;
  - module instantiation: the statement is removed; in case the signals connected to the outputs of the module remain undefined, they are treated as in the case of assignments;
  - conditional statement: either the true or the false block is removed; in case any signal/register remains unsigned, it is treated as in the case of assignments.

For each considered fault, the design is altered by the fault and re-simulated to generate a faulty trace.

For instance, in the running example, the statement $sum = a + b$ at line 8 is a statement token, while the third bit of signal a (i.e., $a[2]$) is considered a bit token. To inject a fault for the first bit token of signal a, a stuck-at 0 fault is injected on the first bit of the signal (ex. a & 11111110). In this example, the only fault, injected to approximate the only available statement token, would consist of removing the instruction at line 8.

7.2 Semantic Impact Estimation

In the last step, the generated assertions are re-evaluated on the faulty traces obtained by perturbing the original RTL description of the DUE by adopting the bit-width and the statement reduction strategies, as reported in Section 7.1. In particular, for each AT, the corresponding fault is injected and the assertions are re-evaluated generating a new contingency table. It is worth noting that the value ATCF is 0 in the contingency table of any assertion for the original implementation (as all assertions are true on the golden trace), while it is very likely greater than 0 for assertions affected by the presence of a fault. Consequently, we can observe variations in the value of ATCT as well. The purpose of this procedure is then to analyse the effect of the different approximation alternatives (represented by ATs) on the functional behaviours (captured by the assertions) of the design.

The whole procedure is implemented in the *evaluate* function reported in Algorithm 3, whose behaviour is detailed hereafter.

Algorithm 3

```
1:  function EVALUATE(A, gt, FT)
2:      GCT ← ∅
3:      diff ← ∅
4:      for all a ∈ A do
5:          GCT[a] ← genContingency(gt, a)
6:      for all ft ∈ FT do
7:          for all a ∈ A do
8:              fct ← genContingency(ft, a)
9:              diff[ft] += abs(GCT[a] − fct)
10:     return diff
```

The function takes as inputs A, gt and FT, where A is the set of assertions mined in the previous step, gt is the golden trace and FT is the set of all faulty traces generated in the first step of the methodology. Note that for each $ft \in FT$ there exists a corresponding unique approximation token at and a unique fault f. The function returns a dictionary $diff$, which stores, for each ft, a matrix representing the sum of the differences between the contingency tables achieved on ft and gt for all assertions belonging to A.

First, we initialize variables GCT and $diff$, where GCT is intended to store the contingency tables of the golden trace (lines 2-3). Then, we iterate over the assertions in A such that the function genContingency(gt, a) evaluates the assertion a on the trace gt to retrieve the corresponding contingency table (line 5). After that, for each couple $\langle ft, a\rangle \in FT \times A$, a "faulty" contingency table is generated and stored in fct (line 8). ftc is then compared with the golden contingency table $GTC[a]$; this is done by returning the absolute difference between the two matrices (line 19). The result of this operation is stored in $diff[ft]$, which contains the impact of fault f on all the considered assertions. Finally, the sum of differences $diff$ is returned (line 10).

At this point, we employ the dictionary of differences $diff$ to sort the ATs in order of decreasing approximability. Since each matrix belonging to $diff$ contains 9 fields (3x3 matrix), there are several ways to estimate the impact of a fault f on the functional behaviour of the design. In this work, we decided to rank each approximation token at by counting the number of times in which the corresponding fault f made the assertions in A fail. That is, each at is ranked by using the ATCF field (first row, second column) of the corresponding matrix $diff[ft]$. This choice is plausible because the higher the increment of ATCF, the worse the impact on the functional behaviour, and as a consequence, the lower

Table 2 Final ranking of ATs for the running example

Bit token	a[6]	a[7]	b[6]	b[7]	b[0]	b[1]	a[0]	a[1]	a[2]	b[2]	a[3]	...
ATCF	0	0	0	0	3	3	4	4	22	23	49	...
Rank	0	1	2	3	4	5	6	7	8	9	10	...
Cluster	0	0	0	0	1	1	2	2	3	3	4	...

the approximability of the corresponding AT. In particular, all the ATs are sorted by increasing order of ATCF.

7.3 Semantic Ranking

Finally, we apply a clustering algorithm to group ATs exposing a similar approximability; this is done to help the designer in the process of simultaneously approximating multiple ATs. To achieve that, we apply Lloyd's version of the *k*-means algorithm [16]. Since the optimal value of *k* is not known beforehand, we employ the elbow method: a well-known cluster analysis heuristic used to determine the number of clusters in a data set. The *k*-means algorithm is executed multiple times: the idea consists of measuring how the variance inside the generated clusters diminishes for an increasing number of clusters *k*. In most cases, if we plotted the variance for every value of *k*, we would observe an "elbow-like" line. The value *k*, at which the reduction of variance plateaus, is considered a good candidate.

Table 2 reports the result obtained by applying the ranking methodology to some ATs of the running example. In this case, we considered as ATs the bits of signals *a* and *b* of the statement at line 8 of Algorithm 2. As expected, the table shows that the two least significant bits of both *a* and *b* are ranked first. This is not surprising since these bits remain unused during simulation; therefore, the corresponding faults do not produce any effect on the functional behaviour of the design (ATCF is zero). Then, after the clustering, our methodology suggests that the designer should simultaneously apply the bit-width reduction strategy to *a*[7:6] and *b*[7:6].

8 Case Study

We conducted an evaluation of our proposed methodology using the RTL description of a commonly used Sobel edge-detection filter. The evaluation aimed to assess the approximation impact of different ATs in terms of functional accuracy and power/area reduction. The primary goal is to demonstrate the effectiveness of our approach in presenting the designer with approximation options that minimally affect functional accuracy while ensuring significant savings in terms of area and power. To evaluate the functional accuracy of the approximated designs, we used the Structural SIMilarity (SSIM) index [25]

For the sake of completeness, we also adopted PSNR and Correlation Coefficient as alternative metrics to SSIM to evaluate the difference between two images. The correlation coefficient is expressed as a number between −1.0 and 1.0. A correlation coefficient of 1.0 means that the pixel values in the two images are perfectly matched. A correlation coefficient of –1.0 means that the pixel values in the two images are perfectly mismatched. A correlation coefficient of 0 means that pixel values in the two images are randomly different.

For our evaluation, we utilized eight images as the workload for the Sobel filter. In the following sections, we report the results of applying the syntax-based and the semantic-based method respectively.

8.1 Results for the Syntactic-based Approach

We applied the bit width reduction to the statements of the HDL design, according to the statement ranking produced

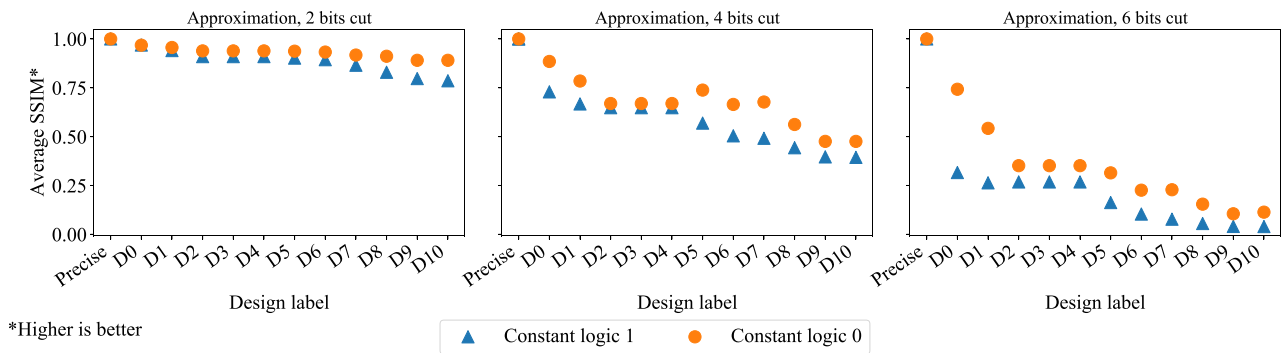

Fig. 6 Sobel results in terms of average SSIM

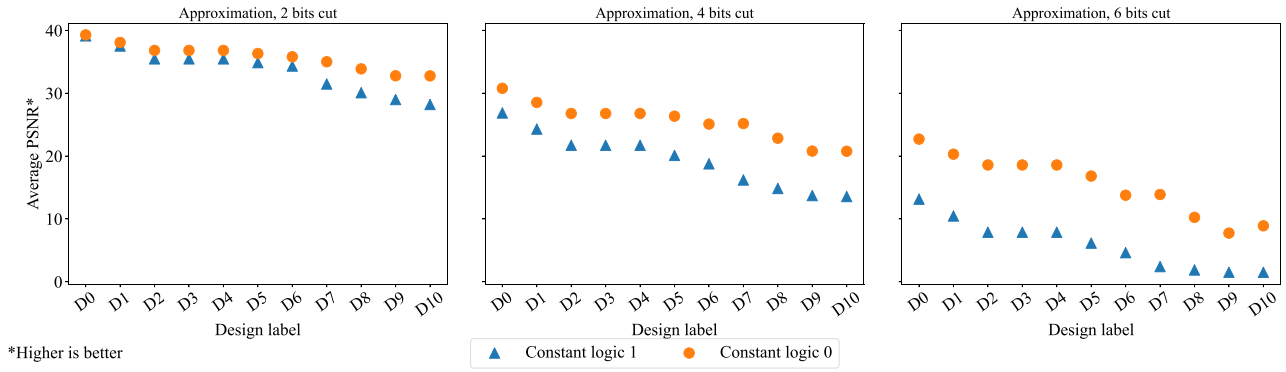

Fig. 7 Sobel results in terms of average PSNR

by the methodology of Section 6. Hence, we proceeded to simulate the workload using the obtained approximate designs. Subsequently, we calculated the SSIM, PSNR and Correlation Coefficient between the output images generated by the approximated designs and the precise ones, which were produced by the non-approximated design. Regarding the bit width reduction, we conducted two distinct experiments when the bit-width reduction is applied as approximation technique. In the first experiment, the bit-width reduction is implemented by consistently setting the cut bits to logic '0'; in the second experiment, we set the cut bits to logic '1'. Regarding the two mentioned variants, we create an approximate design *i* by applying all the ATs with a "criticality" value less than or equal to *i*. Here, "criticality" refers to the position of the statement in the ranked list. In other words, we begin approximating with the first token and gradually proceed, adding each approximation to the subsequent tokens. As we move to the next token, the level of approximation increases accordingly.

Figures 6, 7 and 8, illustrate the results we obtained, showing respectively the average SSIM, PSNR and Correlation Coefficient (y-axis) across the eight images. The x-axis represents the labels of the approximate designs, arranged in ascending order based on their criticality. As highlighted in the graphs, the proposed methodology's ranking accurately predicts the impact of approximation (i.e., higher criticality leads to lower average SSIM) for both cases of setting the cut bit to logic '0' and '1'.

It can be noted that the three metrics (i.e., SSIM, PSNR and Correlation Coefficient) show the same trend: the more approximation, the lower the metric. Once again, we decided to further focus only on the SSIM since it has been widely adopted in the literature and, in particular, in the image process research community [25]. Secondly, we have to mention the difference in terms of quality between setting the removed bits to logic '0' or logic '1'. The latter shows a lower quality (for all three metrics), reflecting the fact that adding removed bits at logic '1' introduces high errors (higher values) that impact the final results.

To give an idea of the image quality associated to a given SSIM, Fig. 10 depicts four images showcasing the expected output of the filter (Fig. 10a), along with the output of three approximate versions resulting from the experiments. These approximate versions have corresponding SSIM values of 0.96, 0.56, and 0.04 (Fig. 10b, c, and d, respectively). The caption of each figure provides information about the approximated statement, the number of cut bits, and the logic value used to set them.

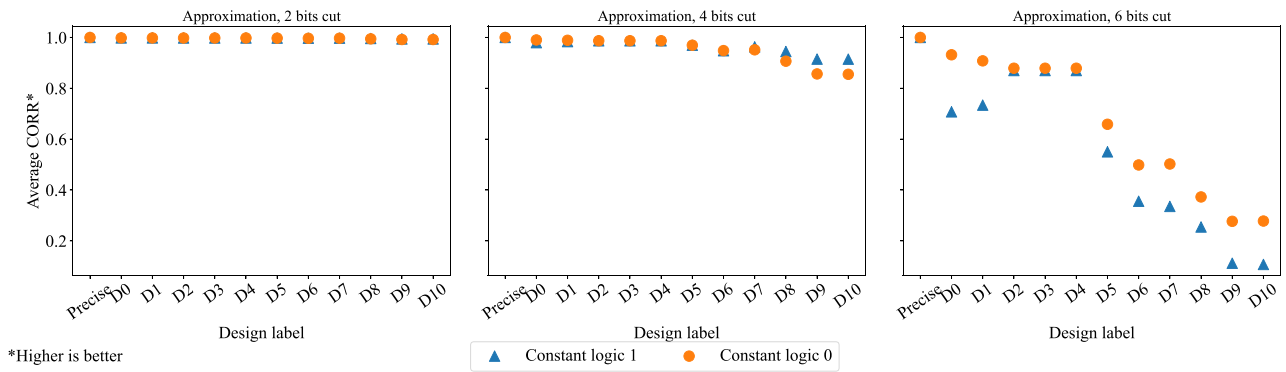

Fig. 8 Sobel results in terms of average correlation

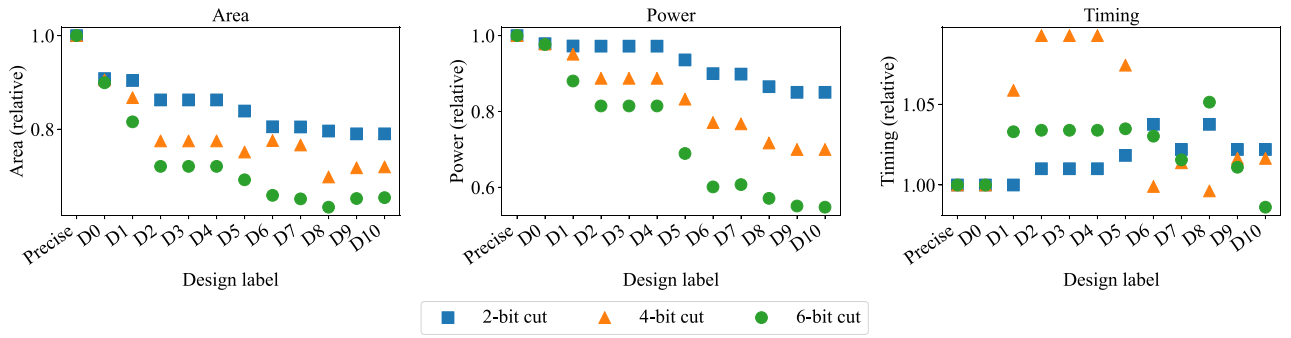

Fig. 9 Area, power consumption, and timing of approximate Sobel design syntheses (setting cut bits to constant logic '0')

Lastly, we conducted the synthesis of the approximate designs obtained from the experiment, where the cut bits were set to a constant logic value of '0' (represented by● in Fig. 6). The decision to choose logic '0' was based on its superior performance in terms of power consumption and area reduction. For the synthesis process, we utilized the FreePDK45 45-nm standard cell technology library as our target technology. Figure 9 presents the results in terms of relative area, power consumption, and timing, which were calculated as $1 - \frac{Precise - D_i}{Precise}$, where the *Precise* variant has a value of 1. The graphs clearly illustrate that both area and power consumption decrease in accordance with the criticality of the approximated tokens. For example, let's examine the results for approximating up to token D6 in Fig. 9, where 4 bits are removed. The area reduction is 25%, and the power reduction is 23%, while still maintaining an acceptable output image quality, as depicted in Fig. 10(c). This indicates that by following the ranking, direct application of approximation (in this case, approximating the first 7 tokens, from D0 to D6) leads to significant overhead reduction while maintaining an acceptable output accuracy. In contrast, without the ranking, the designer would have to consider all possible combinations of HDL tokens. This approach thus holds great promise in terms of its benefits.

Finally, we also noticed that Timing results do not follow the same trends as area and power. This can be explained by the fact that we configured the synthesis tool for area optimization. A lower area means lower activity (thus less power consumption), but there is no guarantee on the timing.

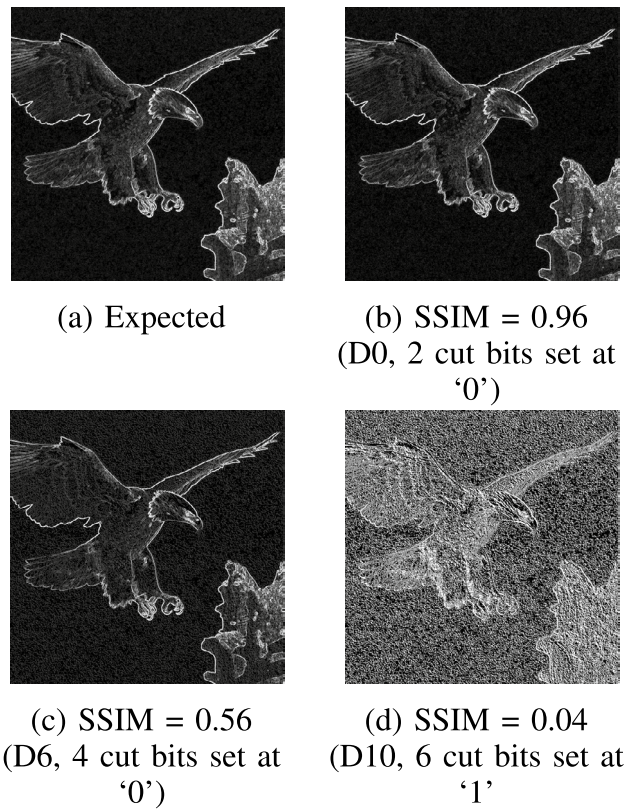

Fig. 10 Visual example of approximate outputs

8.2 Results for the Semantic-based Approach

We applied both the bit-width and the statement reduction approximation strategies on a set of ATs (i.e., bit tokens and statement tokens) by injecting faults as reported in Section 7.1. We then ranked and clustered the ATs, as detailed in Section 7.2. We considered a total number of 236 bit tokens and 38 statement tokens.

We conducted two distinct experiments for bit-width reduction: one by setting the target bit token to 0 and the other to 1. However, as the outcomes were quite similar for both scenarios, we present the results solely for the case of fixing the target bit token to 0.

The diagrams in Fig. 11(a, b) illustrate the SSIM on the *y* axis, achieved by the designs resulting from applying two reduction techniques: bit-width reduction (a) and statement reduction (b) to each individual AT. In the case of (a), the reduction is applied to each bit token, while in (b), it is applied to each statement token. On the *x* axis, the ATs are arranged in descending order, following the ranking

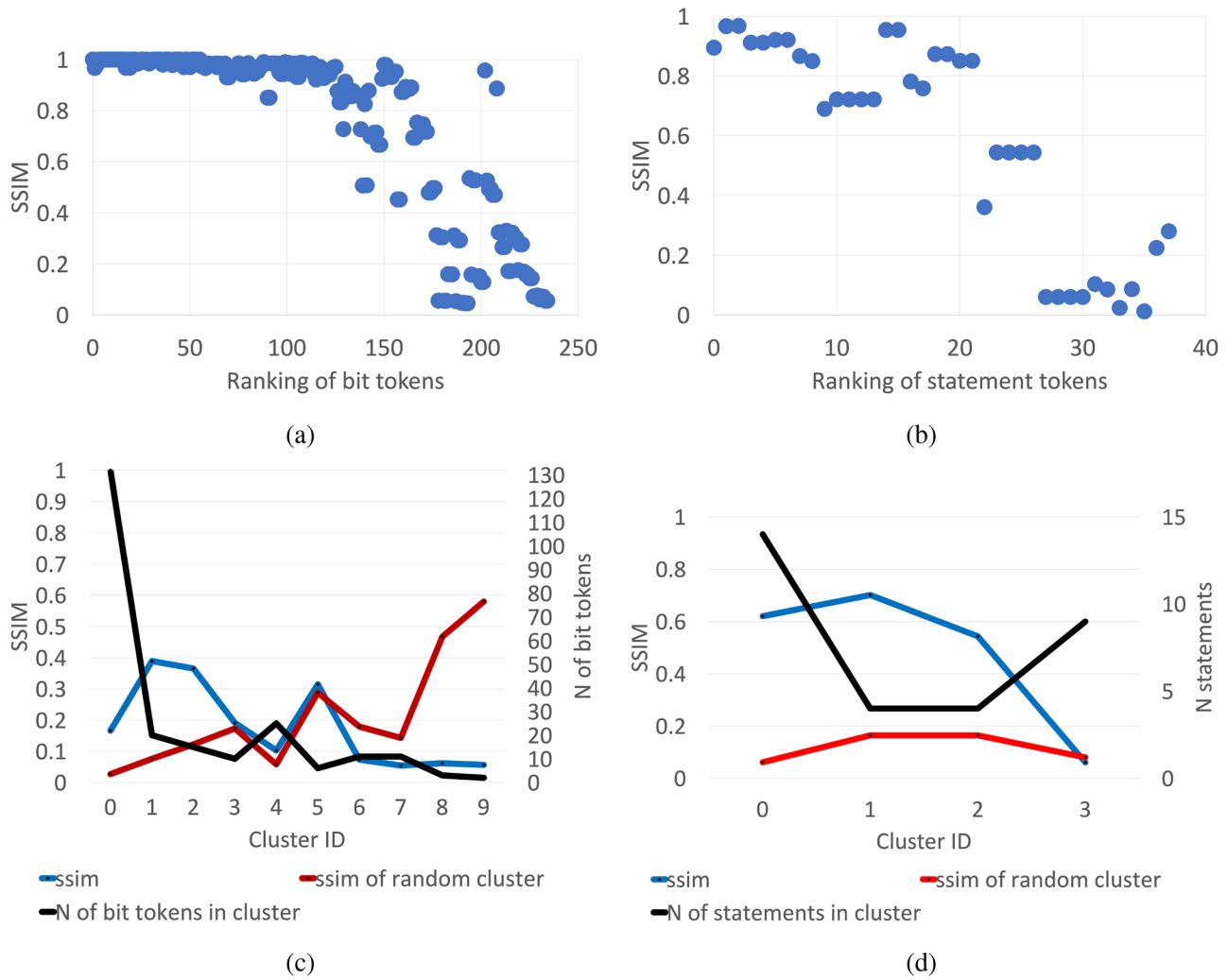

Fig. 11 (**a** and **b**) Impact of AT approximation alternatives on the functionality of the Sobel. Bit tokens (**a**) and statement tokens (**b**) are ordered on the x axis according to the ranking metrics defined in Section 7.2. (**c** and **d**) Impact on the functionality of the Sobel by simultaneously applying the approximations belonging to each AT clusters returned by the methodology proposed in Section 7.2 Clusters are ordered on the x axis from the top-ranked (cluster 0) to the worst-ranked

methodology proposed in Section 7.2. The results show a clear pattern where the ATs that provide the highest SSIM when approximated are the ones ranked at the top according to our approach. This observation applies to both bit tokens and statement tokens.

However, since approximating individual ATs does not yield significant area and power savings, we propose an alternative approach of clustering ATs and applying multiple approximations simultaneously. Consequently, in Fig. 11(c, d), we present the impact of concurrently applying approximations to AT clusters identified using the k-means approach outlined in Section 7. The dark line on the graph represents the size of these AT clusters, while the blue line indicates the SSIM values achieved by the clusters based on our methodology. Additionally, the red line showcases the SSIM attained by a randomly selected set of ATs, whose size corresponds to that of the corresponding cluster in the blue line. An initial observation indicates that the randomly formed clusters result in the lowest accuracy. The second observation pertains to the quality of the ranked clusters produced by our methodology. In the case of the statement reduction strategy (Fig. 11(d)), the cluster containing the ATs ranked at the top (ID 0) demonstrates that applying the approximations from this cluster simultaneously results in nearly the best SSIM. However, for the bit-width reduction strategy (Fig. 11(c)), the situation is different. Larger clusters (e.g., ID 0) achieve unsatisfactory SSIM values, whereas smaller ones (e.g., IDs 1 and 2) generally perform better. This illustrates that the SSIM in the bit-width reduction strategy is influenced by the cluster size, whereas this factor is not significant for the statement reduction strategy. Furthermore, upon examining the composition of clusters containing statement tokens compared to those

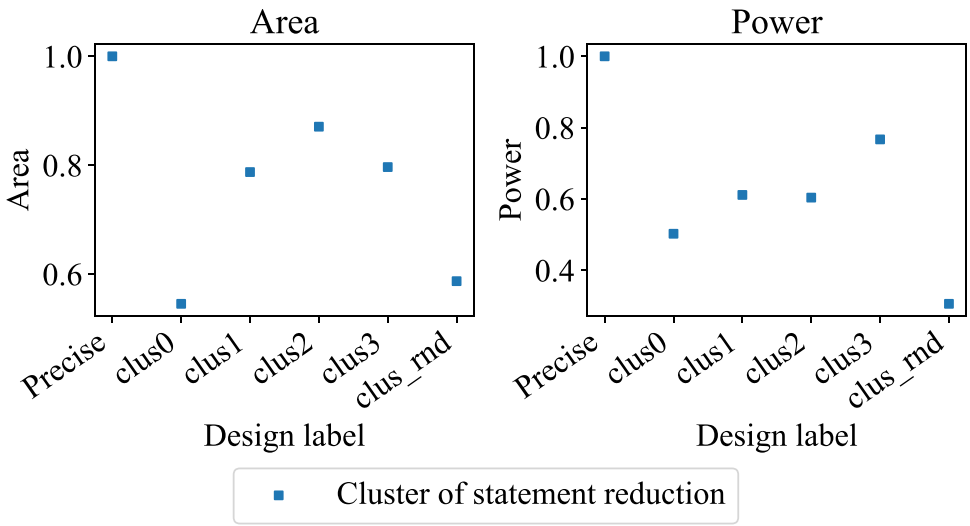

Fig. 12 Saving in terms of area and power by considering the statement token clusters. *Precise* refers to the original design, *clus0*, *clus1*, *clus2* and *clus3* are related to the designs approximated according to the four clusters returned by the our methodology in decreasing order of functional accuracy, *clus_rnd* indicates the average result for the design approximated by using a set of randomly chosen ATs

involving bit tokens, we noticed that the former tend to group ATs from the same cone of logic, whereas the latter do not exhibit this pattern. As a result, the impact of the approximation on functional accuracy becomes more pronounced when a large set of unrelated bit tokens is clustered, while this effect is mitigated when statement tokens belonging to the same cone of logic are grouped together.

As a result, we can deduce that the ranking and clustering approach introduced in this paper for statement tokens represents an effective methodology for assisting designers in exploring the statement reduction approximation strategy with regard to functional accuracy. The subsequent section will demonstrate its performance in terms of area and power savings as well.

We performed design synthesis by approximating the statement tokens based on the clusters generated using our proposed methodology. The FreePDK45 45-nm standard cell technology library was the target for these experiments. The outcomes, in terms of relative area and power consumption, were calculated using the formula $1 - \frac{Precise - clus_i}{Precise}$, where the *Precise* variant has a value of 1. These results are depicted in Fig. 12. The graph highlights that the cluster with ID 0, which was ranked highest by our methodology, exhibits the most significant area and power reduction compared to the *Precise* design, with reductions of 54.53% and 50.24%, respectively. Interestingly, even with such substantial reductions, this cluster (ID 0) maintains a relatively high SSIM value of 0.62, as indicated in Fig. 11(d). Conversely, approximating a random set of statements of the same size as cluster 0 (14 statements) resulted in a design with an average area reduction of 58.69% and an average power reduction of 30.60%. However, this design also exhibited a much lower SSIM value of 0.061. The results for the *clus_rnd* design were derived by synthesizing multiple designs obtained by randomly approximating the same number of statements as in cluster 0; subsequently, the average value was reported.

9 Conclusion

In this paper, we presented two novel approaches to identify the elements (signal/register bits or statements) to be approximated into RTL descriptions through the bit-width and the statement reduction strategies.

In the syntactic-based approach, statements are ranked based on a metric calculated using an impact function that utilizes the syntax tree and the support metric of the mined assertions. Additionally, the function takes into account information from the variable dependency graph of the DUE and the execution frequency of the statements. The presented outcomes demonstrate that this ranking methodology can significantly simplify space exploration, resulting in a remarkable 23% reduction in overheads, all while maintaining an acceptable level of output accuracy.

In the semantic-based approach, approximation alternatives are implemented by fault injection. Their impact on the functional accuracy of the DUE is then estimated accordingly to a metric that evaluates the effect of the approximation on the generated assertions. A ranking and clustering procedure is then proposed to guide the designer in the identification of the best cluster of elements to be approximated. The experimental analysis shows the following major achievements: (1) Our procedure, for both bid-with reduction and statement reduction, is able to rank the approximation alternatives such that they are decreasingly ordered with respect to their effect on functional accuracy. (2) The clustering performs very well by considering the statement reduction approximation, i.e., the functional accuracy is generally higher for the designs obtained by simultaneously applying the approximations corresponding to the top-ranked AT clusters than those synthesized by considering the lower-ranked clusters or by randomly selecting a set of statement tokens. (3) The clustering of bits is affected by the size of the cluster, which is more determinant than the

ranking of its elements for mitigating the amplification of the errors caused by simultaneous bit-width reductions. (4) The saving in terms of area and power achieved by approximating the clusters of statements identified by the proposed approach is generally proportional to the functional accuracy: the higher the saving, the higher the accuracy.

In conclusion, the two approaches can be considered complementary. The Syntax one can quickly provide a first set of approximation targets, while the Semantic one can identify clusters of targets, leading to higher savings at the cost of a more time-consuming analysis due to the fault injection.

Funding This work has been partially supported by the INdAM GNCS, and it was carried out within the PNRR research activities of the consortium iNEST (Interconnected North-Est Innovation Ecosystem) funded by the European Union Next-GenerationEU (PNRR - Missione 4 Componente 2, Investimento 1.5 - D.D. 1058 23/06/2022, ECS-00000043).

Data availability The implementation of methodology presented in the paper is fully accessible from https://github.com/SamueleGerminiani/harm/tree/main/src/dea. The repository contains all the used data.

Declarations

Conflict of interest The authors declare no conflict of interests.

References

1. Barbareschi M, Barone S, Bosio A, Han J, Traiola M (2022) A genetic-algorithm-based approach to the design of DCT hardware accelerators. ACM J Emerg Technol Comput Syst 18(3):1–25
2. Barone S, Traiola M, Barbareschi M, Bosio A (2021) Multi-objective application-driven approximate design method. IEEE Access 9:86975–86993
3. Bosio A, Bragaglio M, Germiniani S, Mori S, Pravadelli G, Traiola M (2022) Assertion-aware approximate computing design exploration on behavioral models. In: 2022 IEEE 23rd Latin American Test Symposium (LATS), pp 1–6
4. Bosio A, Menard D, Sentieys O (eds) (2022) Approximate computing techniques, 1st edn. Cham, Switzerland, Springer Nature
5. Chippa VK, Chakradhar ST, Roy K, Raghunathan A (2013) Analysis and characterization of inherent application resilience for approximate computing. In: Proceedings of ACM/IEE DAC
6. Foster H, Lacey D, Krolnik A (2003) Assertion-based design, 2nd edn. Kluwer Academic Publishers, USA
7. Germiniani S, Pravadelli G (2022) Harm: a hint-based assertion miner. IEEE Trans Comput Aided Des Integr Circ Syst 41(11):4277–4288
8. Han J, Orshansky M (2013) Approximate computing: an emerging paradigm for energy-efficient design. In: Proceedings of IEEE ETS
9. https://github.com/SamueleGerminiani/harm. Accessed 22 Jan 2024
10. Huang P, Wang C, Liu W, Qiao F, Lombardi F (2021) A hardware/software co-design methodology for adaptive approximate computing in clustering and ANN learning. IEEE Open J Comput Soc 2:38–52
11. Isenberg FPT, Jakobs MC, Wehrheim H (2018) Validity of software verification results on approximate hardware. In: IEEE Embedded Systems Letters
12. Jiang H, Santiago FJH, Mo H, Liu L, Han J (2020) Approximate arithmetic circuits: a survey, characterization, and recent applications. Proc IEEE 108(12):2108–2135
13. Lee S, John LK, Gerstlauer A (2017) High-level synthesis of approximate hardware under joint precision and voltage scaling. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp 187–192
14. Liu W, Cao T, Yin P, Zhu Y, Wang C, Swartzlander EE, Lombardi F (2018) Design and analysis of approximate redundant binary multipliers. IEEE Trans Comput 68(6):804–819
15. Liu W, Lombardi F, Shulte M (2020) A retrospective and prospective view of approximate computing. Proc IEEE 108(3):394–399
16. Lloyd S (1982) Least squares quantization in PCM. IEEE Trans Inf Theory 28(2):129–137
17. Ma D, Thapa R, Wang X, Hao C, Jiao X (2021) Workload-aware approximate computing configuration. In: Design, Automation Test in Europe Conference Exhibition (DATE) (in press), pp 258–261
18. Mitra S, Das M, Banerjee A, Datta K, Ho T-Y (2016) A verification guided approach for selective program transformations for approximate computing. In: Proceedings of IEEE ATS
19. Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4):62:1-62:33
20. Mrazek V, Sekanina L, Vasicek Z (2020) Libraries of approximate circuits: Automated design and application in CNN accelerators. IEEE J Emerg Select Topics Circ Syst 10(4):406–418
21. Nepal K, Li Y, Bahar R, Reda S (2014) Abacus: a technique for automated behavioral synthesis of approximate computing circuits. In: Proceedings of ACM/IEEE DATE
22. Sampson A, Baixo A, Ransford B, Moreau T, Yip J, Ceze L, Oskin M (2015) Accept: a programmer-guided compiler framework for practical approximate computing. University of Washington Technical Report UW-CSE-15-01 (vol. 1, no. 2)
23. Sidiroglou-Douskos S, Misailovic S, Hoffmann H, Rinard M (2011) Managing performance vs. accuracy trade-offs with loop perforation. In: Proceedings of ACM ESEC/FSE
24. Venkataramani S, Chakradhar ST, Roy K, Raghunathan A (2015) Approximate computing and the quest for computing efficiency. In: Proceedings of ACM/IEE DAC
25. Wang Z, Bovik AC, Sheikh HR, Simoncelli EP (2004) Image quality assessment: From error visibility to structural similarity. IEEE Trans Image Process 13(4):600–612
26. Ye XFM, Wei S (2019) Runtime hardware security verification using approximate computing: a case study on video motion detection. In: Proceedings of IEEE AsianHOST
27. Yoshisue YMK, Ishihara T (2021) Dynamic verification of approximate computing circuits using coverage-based grey-box fuzzing. In Proceedings of IEEE IOLTS
28. Zervakis G, Amrouch H, Henkel J (2020) Design automation of approximate circuits with runtime reconfigurable accuracy. IEEE Access 8:53522–53538
29. (2010) Standard for property specification language (PSL). IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005), 1–182. https://doi.org/10.1109/IEEESTD.2010.5446004

Alberto Bosio received his MSc (2003) and PhD (2006) in Computer Engineering in the area of digital systems dependability at the Politecnico di Torino (Italy). He is now a Full Professor at Ecole Centrale de Lyon, Institue of Nanotechnology (France). His research activities are related to the design and test of advanced digital circuits and systems. He served as committee and organizing member in several international conferences including DATE (Track Chair) and ETS (Program Chair) as well as guest editors for many international journals. He is a member of the IEEE and the Vice-Chair of the European Test Technical Technology Council.

Samuele Germiniani Samuele Germiniani received the PhD degree in Computer Science from the University of Verona, Italy, in 2023, where he is currently a post-doc research fellow at the Department of Engineering for Innovation Medicine. His main research interests are related to semi-formal verification and embedded security of cyber-physical systems. He is a member of the IEEE.

Graziano Pravadelli PhD in computer science, IEEE senior member, IFIP 10.5 WG chair, is full professor of information processing systems at the Department of the Engineering for Innovation Medicine at the University of Verona (Italy) since 2018. In 2007 he co-founded EDALab s.r.l., an SME working on the design of IoT-based monitoring systems. His main interests focus on system-level modeling, simulation and semi-formal verification of embedded systems, as well as on their application to develop virtual coaching and telemedicine platforms for people with special needs.

Marcello Traiola received the Ph.D. degree in Computer Engineering in 2019 from the University of Montpellier, France and the Laurea degree (MSc) in Computer Engineering in 2016 from the University of Naples Federico II, Italy. Currently, he is a tenured Research Scientist with the Inria Research Institute, at the IRISA laboratory in Rennes, France, in the TARAN research team. Previously, he was a postdoctoral researcher at the Lyon Institute of Nanotechnology, École Centrale de Lyon, in France. His main research topics are emerging computing paradigms (approximate computing, in-memory computing) with special interest in hardware design, test, and reliability. He served as committee and organizing member in several international conferences as DATE (Review Chair). He is an IEEE and ACM member and responsible for the Test Technical Technology Community (TTTC) website. More informations at https://people.rennes.inria.fr/Marcello.Traiola/.