Automatic extraction of assertions from execution traces of behavioural models

Alessandro Danese*, Tara Ghasempouri* and Graziano Pravadelli*†
* Department of Computer Science, University of Verona, Italy. Email: {name.surname}@univr.it
†EDALab s.r.l., Italy. Email: graziano.pravadelli@edalab.it

***Abstract*—Several approaches exist for specification mining of hardware designs. Most of them work at RTL and they extract assertions in the form of temporal relations between Boolean variables. Other approaches work at system level (e.g., TLM) to mine assertions that specify the behaviour of the communication protocol. However, these techniques do not generate assertions addressing the design functionality. Thus, there is a lack of studies related to the automatic mining of assertions for capturing the functionality of behavioural models, where logic expressions among more abstracted (e.g., numeric) variables than bits and bit vectors are necessary. This paper is intended to fill in the gap, by proposing a tool for automatic extraction of temporal assertions from execution traces of behavioural models by adopting a mix of static and dynamic techniques.**

I. Introduction

Specification mining is an automatic approach for extracting assertions from the implementation of the design under verification (DUV). Mined assertions are, generally, not intended as an alternative for their (manual) definition. Indeed, the primary goal is to improve the verification and documentation process by helping verification engineers with a way for evaluating and extending the basic manually-defined set of assertions. While both static (e.g., in [1]) and dynamic (e.g., in [2]) approaches have been proposed, dynamic specification mining (DSM) provides better scalability, and relying on the analysis of execution traces, it can be applied also when the source code of the DUV is not available. Figure 1 describes the general idea about DSM. The DUV model can be described at different abstraction levels targeting, for example, register transfer level (RTL) or transaction level model (TLM) hardware descriptions as well as software protocols and embedded software. Execution traces, generated by simulating the DUV, pass through an assertion miner tool, whose output is a set of candidate assertions that capture the behaviours exercised during simulation, according to a set of temporal patterns. At the end, the quality of the obtained candidate assertions can be evaluated to estimate how good they are in describing the behaviours of the DUV.

Overall, specification mining can be classified in two different domains i.e, Boolean and Non-Boolean. Boolean-domain approaches are referred to specifications for low level HW descriptions [2], [3] or SW protocols [4]. They can only deal with bits and bit vectors (for HW) or they can capture the sequence of function calls (for SW protocols), but they are not able to mine relations through arithmetic expressions on numeric data types such as, for example, $a > b + c$. On the other hand, non-Boolean techniques like [5], are able to infer arithmetic invariants on data types such as string, float and integer. However, they generally do not have the ability to mine specifications which define temporal behaviours of the design, such as "x happens *until* y occurs". First approaches that automatically generate temporal assertions involving arithmetic/logic expressions for behavioural descriptions have been proposed in [6] and [7]. These techniques have contrasting drawbacks. The approach in [6] tends to generate too complex assertions (i.e., composed by large expressions involving several variables together in the same formula). For example, mined assertions may overlap two or more behaviours in the same formula or they can capture too specific situations. In the first case, assertions are difficult to understand by humans. In the second, assertions capture several conditions happening consequently in the considered execution trace, which are strictly related to the peculiarity of the actual values stressed by the testbench rather than to the real symbolic behaviour implemented in the DUV. On the contrary, in [7] compact assertions are extracted in the form of implications where the antecedent is represented by a proposition composed only of two simple operands (e.g, variables or constants) and one logic operator. In this case, only simple assertions are extracted, which could fail to capture interesting implications whose antecedent is composed by more complex formulas.

This work has been partially supported by the EU large-scale integrating project CONTREX (FP7-2013-ICT-10-611146).

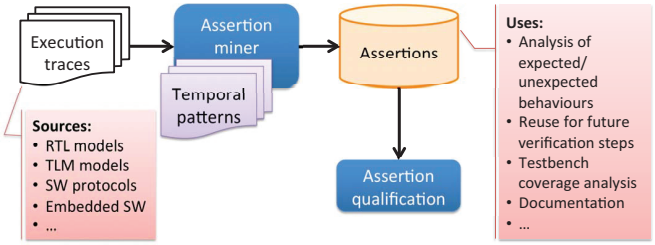

Fig. 1. Dynamic specification mining.

This paper proposes a new approach for temporal specification mining of behavioural descriptions with the intent of overcoming limitations of existing works. The proposed approach implements a new assertion miner that exploits both static and dynamic techniques to capture in a more effective way the behaviours implemented in the DUV in the form of PSL assertions [8]. In particular, main characteristics of the proposed approach with respect to similar existing techniques are:

- implementation of techniques to improve the quality of the mined assertions relying on (i) analysis of the cones of influence of the DUV, (ii) more efficient and accurate extraction and composition of atomic propositions, and (iii) classification of candidate propositions All these techniques pursue the goal of mining neither "too simple" nor "too complex" assertions to avoid drawbacks suffered by [6], [7];
- extension of the temporal patterns considered for assertion mining;
- optimization of the overall execution time.

The rest of the paper is organized as follows. Section II provides further details on the state of the art. Section III introduces some preliminary definitions and shows an overview of the methodology, which is then detailed in Sections IV, V, and VI. Section VII reports experimental results. Finally, Section VIII is devoted to conclusions.

II. 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 [9], [10] instrument the source code to mine linear sequence charts. However, these approaches are not aimed at discovering the complete behaviour of the DUV, but only the collaboration among its components. Other works mine the specifications of the DUV in form of algebraic equation [11] or Hoare-style equations of pre and post-conditions [12], but the temporal behaviours are not considered. Temporal assertion mining is described in [3], [13], [14], where a mining tool, Goldmine, is proposed for extracting respectively, Boolean-level, word-level and system-level assertions. Recently, Goldmine has been further improved to extract more succinct assertions [15]. In general, Goldmine extracts PSL assertions that involve only the *next* temporal operator. The only two approaches that generate temporal assertions considering arithmetic/logic expressions among the variables of the DUV are ODEN [6] and the work described in [7]. However, the drawbacks summarized in the previous section limit their effectiveness. Commercial tools are also available for automatic assertion generation at RTL, e.g., Atrenta BugScope [16] and Jasper ActiveProp [17]. The first generates SVA or PSL assertions where only the *next* temporal operator is considered. The second generates both structural and behavioural SVA *next*-based assertions, but no arithmetic/logic expressions are considered.

III. Preliminaries and overview

Before showing an overview of the proposed strategy, some definitions are reported to create the necessary background.

Definition 1. ***(Atomic proposition)*** *An atomic proposition is a formula that does not contain logical connectives.*

In this paper, we consider atomic propositions of the form: *num_var_1 op num_var_2*; *bool_var = True*; *bool_var = False*; where *num_var_1* and *num_var_2* are numeric data type variables or constants (e.g., natural, integer, float, etc.), *bool_var* is a Boolean variable, and *op* is one of the following operators: $=, <, >, \leq, \geq, \neq$.

Definition 2. ***(Proposition)*** *A proposition is a composition of atomic propositions through logic connectives. An atomic proposition itself is a proposition.*

In this paper, we consider the connectives $\vee$ and $\wedge$ to compose propositions.

Definition 3. ***(Temporal assertion)*** *A temporal assertion is a composition of propositions through temporal operators.*

In this paper, we consider $always$, $next$, $until$ and $before$ operators of PSL language to compose temporal assertions.

The proposed approach consists of three main steps: (i) *identification of cones of influence* (Section IV), (ii) *mining of propositions* (Section V), and (iii) *mining of temporal assertions* (Section VI). Initially, the source code of the DUV and execution traces are analysed to extract the cone of influence for each primary outputs of the DUV. This step is necessary to prevent the assertion miner from generating assertions that mix variables belonging to different cones of influence. On the contrary, longer assertions could be generated that overlap unrelated behaviours, degrading both the readability and the quality of the mined assertions. Then, execution traces are partitioned according to the cones of influence and provided to the *proposition miner*. Each execution trace describes the values assigned to primary inputs (PIs) and primary outputs (POs) of the DUV at each simulation instant. For every cone of influence, the proposition miner is in charge of extracting propositions representing interesting relations between PIs and POs that appear frequently in the analysed execution traces. Finally, such propositions are combined to create temporal assertions by the *assertion miner*. Mined assertions are in the form $antecedent \rightarrow consequent$, where $antecedent$ and $consequent$ are temporal assertions reflecting temporal patterns described in Section VI. Considered variables are PIs and POs of the DUV, since we are interested in capturing system behaviours at the boundary of the DUV. However, the methodology could be applied without changes to consider internal variables too. Details on the three steps of the proposed methodology are reported in the next sections.

IV. Identification of cones of influence

The first step of the methodology consists in the identification of PIs that belong to the cone of influence of each DUV's PO. Given a target variable v, its cone of influence is represented by the set of variables that affect the value of v. This task is fundamental to better characterize the behaviours of the DUV avoiding the risk of generating assertions that capture unrelated behaviours in the same formula. We apply two complementary modalities to extract the cones of influence. When the DUV source code is available, tools based on static approaches can be adopted like, for example, CodeSurfer [18]. Currently, we have interfaced our methodology with CodeSurfer, since it provides a better support for the C++ language, which can be automatically generated starting from popular hardware description languages like, VHDL, Verilog and SystemC by means of HIFSuite [19]. When the DUV source code is not available, the extraction of cones of influence can rely only on the analysis of the execution traces by adopting heuristics techniques like, for example, the solution proposed in Tane [20], which has the ability of producing a list of likely correlations among two or more columns of a table of values. When the DUV source code is not available, our methodology provides Tane with tables representing execution traces to extract functional dependences among PIs and POs. Independently from the adopted strategy, at the end of this phase, each execution trace is partitioned in different *slices* according to the extracted cones of influence.

V. Proposition miner

The purpose of the proposition miner is to generate formulas according to Def. 2, that will be used as antecedents and consequents by the assertion miner. The proposition miner takes the slices of the execution traces[1] as input, and it works in two steps. It first analyses each slice to extract atomic propositions (Section V-A) that describe simple relations between DUV variables, like, for example, $var1 > var2$, $var3 = True$, etc. Then, it composes atomic propositions (Section V-B) to create more complex propositions that could represent antecedents or consequents of the final assertions, like, for example, $(var1 > var2) \wedge (var3 = True)$.

A. Mining of atomic propositions

Mining of atomic propositions is performed by calling an external tool, i.e., Daikon [5], which is able to dynamically extract arithmetic/logic expressions among variables of the DUV by analysing execution traces. In Daikon's terminology, atomic propositions are called *invariants*, since they hold throughout the analysed trace. However, no temporal behaviour can be observed by composing such invariants. Thus, execution traces are *tokenized* in sub-traces. Then, Daikon is called to extract invariants of such sub-traces. These invariants, being true only on some parts of the original execution traces, represent atomic proposition candidates for creation of temporal assertions, as described in the following steps of the methodology (Section V-B and Section VI). Invariants that are true for all sub-traces are instead discarded.

[1] In the following of the paper, to not overload the writing, we use the term execution trace instead of explicitly referring to its slices. Indeed, each of the next steps is executed on the slices derived from the extraction of DUV cones of influence.

Despite of the fact that the next steps of the proposed methodology are independent from the way atomic proposition candidates are extracted, we have chosen Daikon to this purpose since it is one of the most powerful tools for such kind of inference. However, since Daikon execution represented the major bottleneck for the approach described in [6], in this proposal the use of Daikon inside the proposition miner has been optimized as follows.

Invocation on a reduced set of short sub-traces. In order to extract the most complete set of atomic proposition candidates, execution traces have to be tokenized in an exhaustive way by creating all sub-traces of length 2, 3, 4, etc.. This means we should generate $\sum_{i=2}^{n-1} i$ sub-traces from an n-length execution trace. In order to avoid the generation of this huge quantity of sub-traces, we studied how many invariants are generally mined changing the sub-trace sizes. After an empirical analysis on a large set of case studies, we indeed observed that sub-traces longer than 6 simulation instants very rarely provide new candidates w.r.t. shorter sub-traces. Thus, in the current methodology only sub-traces whose length is between 2 and 6 simulation instants are considered. This way, given an execution trace composed of n instants, the total number of sub-traces provided to Daikon is $\sum_{i=1}^{min(n-2,5)}(n-i)$.

Analysis of a reduced set of invariant patterns. The list of invariant patterns considered by the Daikon's inference engine is very rich [21]. However, most of them are not interesting to mine temporal assertions for behavioural descriptions of hardware components or embedded SW and they can be removed to save execution time. For example, invariants typically occurring in software programs like $x.field\ is\ null$, *array A is sorted*, etc., are irrelevant in our context. Thus, we restricted Daikon to search only for arithmetic/logic expressions involving the most common relational (e.g., $=, \neq, \leq, \geq, <, >$) and arithmetic (e.g., $+, -, *, \div$) operators. Moreover, we imposed also that constants are not allowed as operands of relational operators, with the only exception represented by the Boolean constants *True* and *False*. In most of cases, it is unlikely that atomic propositions like $variable = constant$ play a decisive role for the functionalities of the design. Instead, it is generally more important to capture relations between variables.

More efficient invocation on sub-traces. Daikon's execution flow is composed of three steps: (i) initialization of internal data structures according to the selected invariant patterns and the data types of considered variables, (ii) mining of invariants, and (iii) printing of results. By profiling the three phases on several case studies and different lengths of execution sub-traces, we derived interesting observations. In particular, we observed that the third phase is independent from the length of the sub-traces analysed by Daikon and definitely negligible from the execution time point of view. On the contrary, the second phase strictly depends on sub-trace length. However, execution time related to the second phase is almost irrelevant (few milliseconds) for the very short sub-traces extracted by the tokenization procedure. The real bottleneck is represented by the first phase, which costs, in average, almost one second for each analysed sub-trace independently from its length. To reduce this cost, we modified the workflow of Daikon to allow a more efficient integration with the proposition miner. In particular, a single Daikon process is call for each execution trace, and, consequently, the initialization step is performed once per execution trace instead of once per sub-trace, as done in the original workflow. Given the sub- traces set provided by the tokenizer for each execution trace, data structures are initialized for the first sub-trace, then they are saved by deep copy and refreshed before moving to the next sub-trace. The time saved with this optimization is significant and it greatly reduces the impact of Daikon on the overall mining flow, as reported in the experimental results.

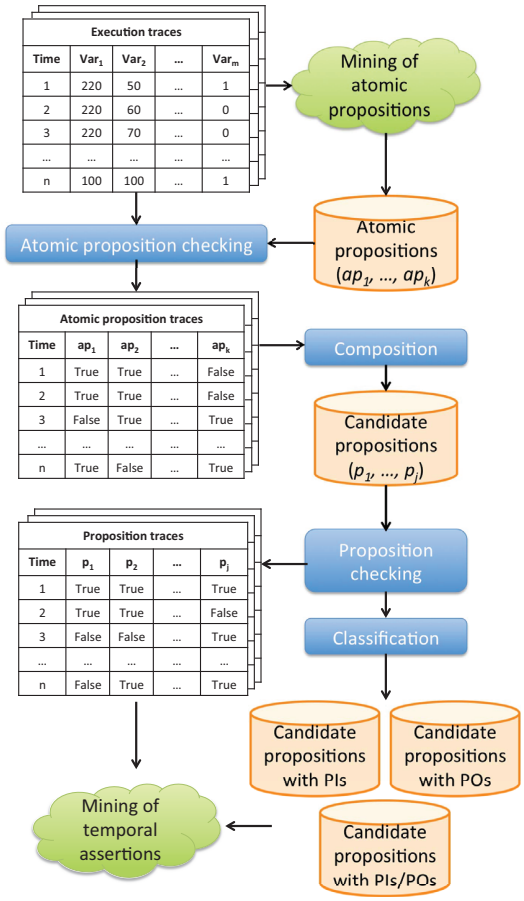

Fig. 2. Generation of candidate propositions.

B. Generation of candidate propositions

The goal of this phase is to compose atomic propositions in a set of candidate propositions according to Def 2 (Fig. 2). Such propositions will represent candidate *antecedents* and *consequents* for the final phase of the methodology. The set of atomic propositions is evaluated with respect to the execution traces. A checking procedure (*atomic proposition checking*) is executed to identify which atomic propositions are true in each instant of each execution trace. The output of this phase is represented by a table (*atomic proposition trace*) for each execution trace whose format is as follows. The first column refers to the time instants. Then, there is a column for each atomic proposition reporting its truth value for each time instant of the execution trace. Subsequently, a *composition procedure* generates a candidate proposition from each row of the atomic proposition trace by composing in an *AND* formula all atomic propositions that are marked as true. For example, in Figure 2, the first and the second atomic propositions (i.e., ap_1 and ap_2) are true at time instant 1, then a candidate proposition is created by composing ap_1 and ap_2 in the formula $p_1 := ap_1 \wedge ap_2$.

The next step (*proposition checking*) creates a new table (*proposition trace*) for each execution trace to identify which candidate propositions are true in each instant. The utility of this tables will be clear in Section VI.

Finally, candidate propositions are classified according to the kind of variables (primary inputs, primary outputs or both) they involve. Such a classification is used to restrict the work space of the assertion mining algorithm and generate high-quality assertions. In particular, we can distinguish among:

PI propositions: they involve only primary inputs of the DUV. They capture the behaviours of the testbenches used to simulate the DUV, while they cannot express anything about the behaviour of the DUV. They are good candidates to be antecedents of temporal assertions.

PO propositions: they involve only primary outputs of the DUV. They observe conditions occurring as a consequence of the DUV execution. They are definitely good candidates to be consequents of temporal assertions. However, they can be used also as antecedents when we are interested in capturing temporal implications between expected results of a DUV.

PIPO propositions: they involve both PIs and POs of the DUV. They can be considered good candidates for both antecedents and consequents. However, when a PIPO proposition is used as a consequent, it could be appropriate to prune its atomic propositions that predicate only on PIs.

VI. Assertion Miner

In the last phase of the methodology, the candidate propositions are combined according to a set of temporal patterns to create candidate temporal assertions. Given a candidate proposition p_a of type PI, PO or PIPO that acts as antecedent, and a set of candidate propositions $P = (p_c^1, ..., p_c^k)$ of type PO or PIPO that act as consequences, the considered patterns are the following:

1) *Next:* $always(p_a \rightarrow next\ p_c^i)$;
2) *N-next:* $always(p_a \rightarrow next[N]\ p_c^i)$;
3) *Until:* $always(p_a \rightarrow p_a\ until\ p_c^i)$;
4) *Alternating:* $always(p_a \rightarrow next\ (p_c^i\ before\ p_a))$.
5) *Next_or:* $always(p_a \rightarrow next\ (p_c^1 \vee p_c^2 \vee ... \vee p_c^k))$;
6) *N-next_or:* $always(p_a \rightarrow next[N]\ (p_c^1 \vee p_c^2 \vee ... \vee p_c^k))$;
7) *Until_or:* $always(p_a \rightarrow p_a\ until\ (p_c^1 \vee p_c^2 \vee ... \vee p_c^k))$.

These patterns allow to capture interesting behaviours between PIs and POs of the DUV according to the classification proposed in [22] that describes frequently used assertions for representing design specification. Patterns similar to number 1, 3, and 4 have been considered also in [2], [6], [7]. On the contrary, patterns 2, 5, 6 and 7 have never been considered by other temporal mining tools. Approaches based on Goldmine [3], [13], [14], [15] are instead oriented to capture chain of next events, like $p_1 \wedge next\ p_2 \wedge ... \wedge next[i]p_{i-1} \rightarrow next[i+1]p_i$. Such a kind of pattern is not considered in this work. We think it is more suited to predicate over internal variables of the DUV rather than PIs and POs, which are, instead, our target.

The assertion mining algorithm works as shown in Figure 3. For each of the considered patterns, a corresponding accepting automaton has been implemented. Given one of the automata and given a couple of candidate propositions, the automaton is traversed by following each proposition trace generated by the proposition miner. If the error state is never reached for all the proposition traces, a candidate assertion is generated and stored by composing the two candidate propositions according to the considered temporal pattern. On the contrary, reaching the error state for at least one proposition trace is a sufficient condition to discard the candidate assertion. The proposed approach can be easily extended to support further temporal patterns by defining the corresponding automata and composing propositions accordingly.

The collected candidate assertions are then converted in checkers, by using, for example, IBM FoCs[23], and connected to the DUV. A different and very larger set of testbenches, with respect to the set initially used to generate the execution traces, is applied to stress the DUV and the candidate assertions searching for counterexamples. Each time a checker fails, the corresponding candidate assertion is discarded. Only assertions

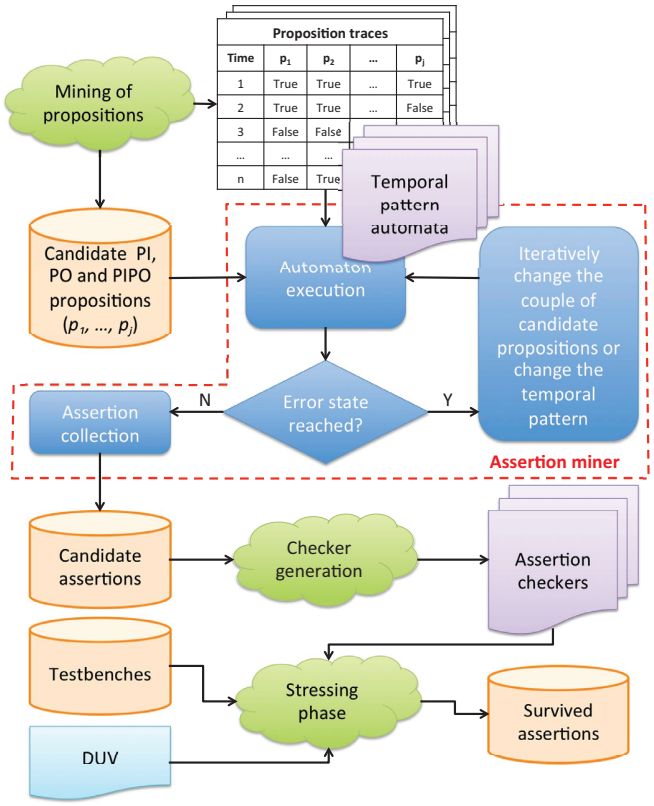

Fig. 3. Generation of temporal assertions.

that survive to this stressing phase are definitely collected. The stressing phase is applied to increase the likelihood that the surviving assertions are satisfied by the DUV independently from the execution traces adopted for their extraction. Being a dynamic, not exhaustive, approach, we cannot be completely guaranteed, but larger is the testbench set higher is the probability of collecting assertions that are satisfied by the DUV without the risk of escaping counterexamples. Since the mining procedure is much more expensive than simulating the DUV connected with checkers, it makes sense to use a reduced set of testbenches for the mining phase and a larger set of testbenches for the stressing phase.

To clarify how automata work, let us consider the *next-based* patterns. Details on the other temporal patterns are omitted for lack of space. Next-based patterns number 1 and 2 rely on the automata shown in Fig. 4. The only difference is represented by the number of states to be traversed before reaching the accepting state (*ant*) after the activation of the antecedent. In case the error state is reached the candidate assertion is discarded and a different couple of antecedent/-consequent candidates is analysed. The automata for patterns number 5 and 6 are similar, but in case the error state is reached at simulation instant t, on the assumption that p_a and p_c have been activated at least once before reaching t, an alternative searching procedure is activated. This procedure analyses the proposition trace to see if a different candidate proposition p_j is true at time t instead of p_c. If p_j is found, it is collected and the automaton restarts from the initial state searching for a new activation of the antecedent p_a in the rest of the proposition trace. When all proposition traces are completely traversed, collected propositions are composed in an OR formula together with p_c. Such a formula becomes the consequent of a next_or or N-next_or assertion where p_a is the antecedent. To avoid the risk a huge number of propositions are included in the OR formula, the error state can be reached

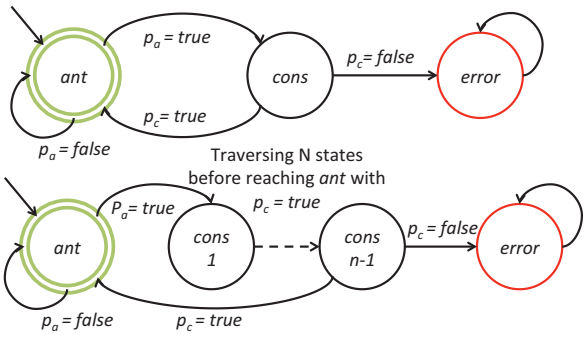

Fig. 4. Next (upper part) an N-next (lower part) pattern automata.

a maximum number of times defined by the user. When this threshold is overcome the automaton stops and the couple of candidates p_a, p_c is definitely discarded. From our experience reasonable thresholds are between 2 and 4.

VII. Experimental results

Experimental results have been carried out on an Intel Core2 Duo 2.2 GHz processor equipped with 2.0 GByte of RAM running Linux OS. Efficiency and effectiveness of the proposed mining methodology has been evaluated by considering the benchmarks reported in Table I. For *B06* and *BMaker* two cones of influence have been identified, only one for the other benchmarks. A set of execution traces for a total number of 10,000 simulation instants has been generated for all benchmarks.

Table II reports the number of atomic propositions (*AP*), the number of propositions (*P*), and the number of temporal assertions (*Assertions*) extracted by the proposed approach before running the stressing phase described in Section VI. In particular, the number of assertions have been divided among next-based (*X*), until-based (*U*), and alternating (*A*). Concerning the N-next pattern, values 2 and 3 are considered for the parameter *N*. Finally, in the last two columns, the total execution time of the mining approach (*Time*) and the percentage of this time spent by Daikon (*D*) are shown.

As expected, the most time-expensive step of the miner is the extraction of atomic propositions performed by using Daikon. Daikon time is not so much affected by the number of considered variables. In fact, considering *B06* and *BMaker*, we observe that there is a low difference between the execution time related to the single cones of influence (where the considered variables are a subset of the total), and the execution time of the DUV without differentiating the cones of influence. Indeed, Daikon time is dominated by the time spent to initialize internal data structures, which depends mainly on the data type of the considered variables. For a Boolean variable only two invariant patterns have to be considered (i.e., $var = true$, $var = false$), which are very simple to be inferred. On the contrary, for numeric data types the number of invariant patterns is higher and their inference is more difficult. This justifies why benchmarks where only Boolean variables (e.g., bit and bit vectors) are involved (i.e., *B03*, *B06* and *Uart*) have an execution time lower than the other benchmarks, which are implemented by using integer (i.e., *Dig_proc*) or real data types (i.e., *BMaker* and *Thermostat*).

For *B03* a high number of propositions has been generated. This is due to the nature of the DUV that, being an arbiter among 4 devices, presents a high number of possible combinations among the four request signals and the four grant signals. Such different combinations give rise to a high number of next-based assertions according to the received requests. On the contrary, the sequential length of *B03* is too short to reflect until-based behaviours, and no evident alternating behaviour is implemented by the arbiter.

DUV	*Typology*	*Cones*	*PIs*	*POs*	*Lines*
B03	RTL	1	4	4	141
B06	RTL	2	2	6	128
cone 1	RTL	-	1	4	-
cone 2	RTL	-	2	2	-
BMaker	ESW	2	4	4	552
cone 1	ESW	-	3	1	-
cone 2	ESW	-	1	3	-
Dig_proc	RTL	1	2	8	2580
Thermostat	ESW	1	2	1	56
Uart	TLM	1	9	5	14815

TABLE I. Characteristics of benchmarks.

DUV	*AP*	*P*	*Assertions*				*Time (s.)*	*D*
			X	*U*	*A*	*Total*		
B03	28	80	240	0	0	240	551	93%
B06 w/o cones	29	13	27	4	0	31	577	93%
B06 w/ cones	23	11	15	6	0	21	1078	93%
cone 1	14	6	12	2	0	14	540	93%
cone 2	9	5	3	4	0	7	538	93%
BMaker w/o cones	37	19	0	9	0	9	1641	89%
BMaker w/ cones	17	9	0	8	0	8	2949	89%
cone 1	8	4	0	3	0	3	1476	89%
cone 2	9	5	0	5	0	5	1473	89%
Dig_proc	15	4	12	0	0	12	1916	93%
Thermostat	4	3	0	3	0	3	1297	94%
Uart	20	19	57	0	16	73	394	92%

TABLE II. Experimental results.

For *BMaker* and *Thermostat* only until-based assertions are mined. This is consistent with the fact that their evolution depends on real data-type variables that evolve in a continuous, rather than discrete, way. Typical behaviours captured by analysing these benchmarks are "command is off until temperature is higher than setpoint" or "engine turns clockwise and engine turns fast until input of the mixer becomes false".

Extracted assertions have been then subjected to the stressing phase by stimulating the corresponding checkers connected to the DUV with up to 1 million stimuli. Table III reports the number of assertions for which the stressing phase was unable to found counterexamples at varying of the number of stimuli. For most of benchmarks, we found very few counterexamples by increasing the number of stimuli. Generally, by using a number of stimuli which is double (20,000) with respect to the length of execution traces adopted for the mining phase (10,000), the number of "survived" assertions stabilizes and no new counterexample is found any more. The only benchmark that does not converge on the number of survived assertions is *Uart*. Indeed, *Uart*, after a set of input is provided, requires 670 simulation instants before the corresponding result is observable at primary outputs. By using an execution trace of length 10,000 it means we can simulate completely no more than 15 different operations, which are too few for mining a set of assertions with a high degree of survival. However, this is not a problem of the proposed methodology, but a characteristic of the benchmark.

Finally, we report a comparison between the proposed approach and the tool ODEN described in [6]. Concerning the total execution time, the comparison is reported in Table IV. The table reports execution time at varying of the total number of considered simulation instants in the execution traces. Actual values are reported for the proposed methodology; for ODEN actual values are reported only for 100 and 1,000 simulation instants, since reaching 10,000 simulation instants becomes practically intractable for most of the considered benchmarks. ODEN's time for 10,000 simulation instants has been estimated on the basis of the tendency observed for shorter execution traces. By looking at the table, it appears that the increasing in execution time for the proposed approach is linear, at varying of the length of execution traces. On the contrary, ODEN execution time increases polynomially, till becoming unacceptable for long execution traces. This

Design	Number of stimuli					
	10000	*20000*	*40000*	*80000*	*100000*	*1M*
B03	240	191	177	171	171	171
B06 w/o cones	31	30	29	29	29	29
B06 w/ cones	21	21	21	21	21	21
cone_1	14	14	14	14	14	14
cone_2	7	7	7	7	7	7
BMaker w/o cones	9	9	9	9	9	9
BMaker w/ cones	8	8	8	8	8	8
cone_1	3	3	3	3	3	3
cone_2	5	5	5	5	5	5
Dig_proc	12	12	12	12	12	12
Thermostat	3	3	3	3	3	3
Uart	73	71	71	67	67	57

TABLE III. SURVIVED ASSERTIONS AFTER THE STRESSING PHASE.

Design	Length of execution traces					
	Proposed methodology			*ODEN*		
	100	*1000*	*10000*	*100*	*1000*	*10000*
B03	6	58	551	569	11202	643227
B06 w/o cones	6	56	577	500	7920	118539
B06 w/ cones	11	105	1043	na	na	na
cone_1	5	51	505	na	na	na
cone_2	6	54	538	na	na	na
BMaker w/o cones	33	167	1641	474	5209	323040
BMaker w/ cones	60	296	2949	na	na	na
cone_1	30	149	1476	na	na	na
cone_2	30	147	1473	na	na	na
Dig_proc	13	135	1916	625	12721	730651
Thermostat	13	116	1297	394	7102	252501
Uart	14	134	394	629	10934	425375

TABLE IV. COMPARISON BETWEEN EXECUTION TIME (IN SECONDS) OF THE PROPOSED APPROACH AND ODEN.

difference is mainly due to the different way Daikon is used in the extraction of atomic propositions, which represents the most expensive phase of both the proposed methodology and ODEN. By adopting the optimizations described in Section V-A, the cost of Daikon's invocation on a set of sub-traces is almost 40 times lower than ODEN's approach.

A different comparison is related to the number of assertions extracted by the proposed approach and by ODEN at varying the length of the execution traces. Results are reported in Table V. We observe that the number of assertions extracted by the current methodology has an horizontal asymptotic trend, while for ODEN the values generally keeps going to increase by augmenting the number of simulation instants. This highlights that the approach proposed in this paper is not dependent on the length of the execution traces, but on the number of different behaviours that execution traces expose. In fact, when the most of cases are covered by the execution traces, no new assertion is mined. On the contrary, in ODEN the number of assertions keeps to increase for longer traces. The reason is evident by analysing the assertions extracted by the two approaches. In ODEN, assertions are more related to the specific values assigned to PIs by the testbench. In the approach presented in this paper, assertions reflect symbolic relations between PIs and POs. For example, several assertions in ODEN include atomic propositions of the kind *variable* = *constant*. Clearly, if the value's range of a variable is very large, the number of possible atomic propositions of this kind increases rapidly by using different stimuli. On the contrary, to avoid such a problem, in this work we explicitly discarded the possibility of comparing a variable with a constant, as reported in Section V. As a result, we generate a smaller set of assertions that focuses more precisely on the relation among PIs/POs that derives from the DUV functionality, discarding specific conditions that are just an instance of more interesting and more general behaviours.

VIII. CONCLUSIONS

In this work we proposed a mining approach for behavioural descriptions that automatically extract temporal assertions from execution traces. Mined assertions capture arithmetic/logic relations between PIs and POs according to a set of temporal patterns that can be easily extended. With respect to similar existing techniques, the proposed methodology points

Design	Length of execution traces						
	Proposed methodology				*ODEN*		
	100	*400*	*1000*	*10000*	*100*	*400*	*1000*
B03	132	230	240	240	1554	15448	38385
B06 w/o cones	40	39	37	31	204	586	608
B06 w/ cones	32	23	21	21	na	na	na
cone_1	18	16	14	14	na	na	na
cone_2	15	7	7	7	na	na	na
BMaker w/o cones	9	9	9	9	287	1266	834
BMaker w/ cones	8	8	8	8	na	na	na
cone_1	3	3	3	3	na	na	na
cone_2	5	5	5	5	na	na	na
Dig_proc	6	7	6	12	24	26	55
Thermostat	3	3	3	3	35	89	88
Uart	34	34	43	73	156	162	341

TABLE V. COMPARISON BETWEEN THE NUMBER OF ASSERTIONS EXTRACTED BY THE PROPOSED APPROACH AND ODEN.

out an higher efficiency from the execution time point of view, and an higher effectiveness by considering the quality of mined assertions. Current limitations that will be part of future works are related to the incapability of capturing liveness assertions involving the *eventually* operator.

REFERENCES

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

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

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

[4] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *Proc. of ACM/IEEE ICSE*, 2006, pp. 282–291.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: automatic assertion generation using data mining and static analysis," in *Proc. of ACM/IEEE DATE*, 2010, pp. 626–629.

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

[15] 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.

[16] http://www.atrenta.com/about-bugscope.htm5.

[17] "Jasper Activeprop," http://www.jasper-da.com.

[18] http://www.grammatech.com/research/technologies/codesurfer.

[19] N. Bombieri, G. D. Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli, "Hifsuite: Tools for hdl code conversion and manipulation," *EURASIP J. Embedded Syst.*, pp. 4:1–4:20, 2010.

[20] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, "TANE: An efficient algorithm for discovering functional and approximate dependencies," *The computer journal*, vol. 42, no. 2, pp. 100–111, 1999.

[21] http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Invariant-list.

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

[23] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *Proc. of CAV*, 2000, pp. 538–542.