Extracting Hardware Assertions Including Word-Level Relations over Multiple Clock Cycles

Mami Miyamoto[1], Kiyoharu Hamaguchi[2]

Interdisciplinary Graduate School of Science and Engineering
Shimane University, Matsue, Shimane, 690-8504 JAPAN
[1]E-mail: s169513@matsu.shimane-u.ac.jp
[2]E-mail: hama@cis.shimane-u.ac.jp

Abstract

Various mining approaches have been proposed for the automatic generation of temporal assertions from execution traces of hardware designs. These approaches can handle assertions based on LTL formulas or PSL, and many of them can represent word-level relations such as inequalities, additions, and so on. In the existing methods, however, such relations are searched only within a clock cycle. They cannot extract a property such that two values at inputs are added, and its result appears two clock cycles later at an output. We propose a method to extract relations over multiple clock cycles between variables as atomic propositions by analyzing execution traces and to generate assertions including the relations. Our method can also efficiently generate assertions by extracting frequent relations between atomic propositions over multiple clock cycles as propositions, that is, conjunctives of atomic propositions. The experimental results demonstrate the feasibility of the proposed method.

Keywords

Assertion mining;

1. Introduction

Assertions in hardware design verification are properties that the design should satisfy. They are mainly used for assertion-based verification. Assertion-based verification is a common approach for functional verification. The assertions are used as checkers in simulation and formal verification. Assertions are usually defined manually, but assertion definition is a process that requires much time and high expertise. Therefore, as a complementary approach to manual definition of assertions, many approaches have been proposed for automatically extracting assertions. Extracted assertions can be used for checking design evolutions, finding errors and documentation. In addition, by extracting the relations between input and output signals of the design as assertions, it can be reused for other designs.

There are two kinds of approaches for assertion mining: static approach and dynamic approach. Since the static approaches such as [1] [2] relies on formal analysis of designs, it has been known that it cannot scale well. Recently, the dynamic approaches, where assertions are mined from simulation traces, have been studied well, and have been successful in finding assertions that are good in terms of compactness, understandability and fault detection capacities. As shown in section 2, various mining approaches have been proposed for the automatic generation of temporal assertions from execution traces of hardware designs. However, the existing methods cannot extract properties such that "x and y as inputs at a clock cycle are added and its result appears two clock cycle later at output z." Such properties can be described using local variables in SVA (System Verilog Assertion). An SVA description for the above property can be "$(1, v_x = x, v_y = y)\ |{-}{>}\ \#\#2(z = v_x + v_y)$." However, since the existing methods use LTL (Liner Temporal Logic) and PSL (Property Specification Language), describing such a property is not straightforward.

In order to address this problem, in this paper, we propose a method to extract relations over multiple clock cycles between variables as atomic propositions by analyzing execution traces and to generate assertions including these relations. Our method can also efficiently generate assertions by extracting frequent relations between atomic propositions over multiple clock cycles as propositions, that is, conjunctives of atomic propositions. This process can prevent mining assertions with overly constrained antecedents. The features of the proposed method are as follows:

- Extract relations between word-level variables over multiple clock cycles as atomic propositions and generate assertions including such propositions.
- Prevent mining assertions with overly constrained antecedent by extract frequent relation between atomic propositions over multiple clock cycles as propositions using the frequent pattern mining.
- Consider the features of a digital circuit that often keep values unchanged over multiple clock cycles and perform assertion mining by focusing on the changes of the values in the execution traces.

2. Related Works

There have been many dynamic approaches proposed for hardware assertion mining, where temporal properties are to be mined. In [3], some typical patterns such as req-ack relations or state machine protocols are mined with data mining techniques. [4] provides a method for finding repeated patterns. [5] and [6] developed methods for mining patterns for interface protocols. Assuming temporal template patterns including two variables, [7] showed a practically fast mining algorithm which focus on the changes of signal values, and applied the obtained assertion sets to fault analysis. In [8] and [9], using a method based on the decision tree algorithm and static analysis, more general temporal patterns of form "$always\ (antecedent\ {-}{>}\ consequent\ )$" have been successfully extracted, where antecedents and consequents are conjunctions of Boolean atomic propositions preceded by more than or equal to zero "$next$" operators. This work was extended to cover assertions having uncertain delays within ranges in consequents for targeting transaction level models in [10].

In the above works, as atomic propositions, Boolean (or binary) variables were considered. Methods for handling word-level features, which describes some relations on variables of more than one bit-width, have been proposed in [11] and [12]. In [11], atomic propositions of form "$x = constant$" were extracted to construct assertions. In [12], to extract the word-level arithmetic relations such as "$z = a + b$" or "$x > y$", they used Daikon [13]. The latter work was improved through [14] and [15], and in [16], they have proposed a method for mining assertions which match given general LTL templates.

Extracted atomic propositions with arithmetic relations in the existing methods, however, are searched only within a single clock cycle. To the best of our knowledge, there has not been any approaches available to handle word-level relations over multiple clock cycles. We propose a method to cover these features in this paper.

3. Preliminaries

This section introduces the definitions necessary for the proposed method and assertions to be mined.

In the following, we assume a model M as an RTL design. V is a set of variables on M. Assertions are mined over V. The variables in V are given manually. In this paper, they are arbitrary signals in M. They can be of bit-width 1 or more, and they can be primary inputs, primary outputs or internal signals. We give $V_a \subseteq V$ as a set of variables which value is assigned in the consequent of assertions. For example, variable z in relation $z = a + b$ is included in V_a.

Definition 1. (***Execution trace***) *Given a finite sequence of simulation instants $t_0 t_1 t_2 \dots t_{n-1}$ and a model M on a set of variables V, an execution trace of M is a finite sequence $T = V_0 V_1 V_2 \dots V_{n-1}$, where V_i is the evaluation of variables in V at simulation instant t_i, that is, $V_i(v)$ is the value of $v \in V$ at t_i.*

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

In this paper, in addition to relations within a single clock cycle such as $a = True$, $a = 5$, $a > 5$, $a > b$, $z = a + b$, we also consider relations between variables over multiple clock cycles such as $z[2] = a[0] + b[0]$, as atomic propositions. Such an atomic proposition refers to variables at multiple clock cycles. We assign clock cycle 0 to the variables referred to at the earliest clock cycle among them. Then, $x[i]$ is value of x referred to i clock cycles later. That is, $z[2] = a[0] + b[0]$ means "a and b as inputs at a clock cycle are added and its result appears two clock cycles later at output z." In this paper, we consider the following atomic propositions: **(i)** value assignment (e.g., $a = True$, $a = 5$), **(ii)** relation representing bit shift operations between two variables $z[i] = a[j]\ op\ n$, where $op = \ll$ or $\gg$, $i \geq j$, $z \in V_a$ and $n \in \mathbb{N}$, **(iii)** relation among three variables $z[i] = a[j]\ op\ b[k]$, where $op = +, -, \times, /, bitwize\text{-}and, bitwize\text{-}or$, $i \geq j$, $j \geq k$ and $z \in V_a$. Users can modify these atomic proposition templates.

Definition 3. An atomic proposition $z[i] = a[j]\ op\ n$ of (ii) and an atomic proposition $z[i] = a[j]\ op\ b[j]$ of (iii) in the above, is defined to *hold* at clock cycle of $min(i, j, k)$.

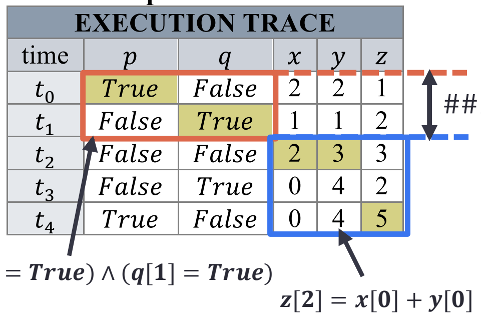
It can be defined in a similar way for propositions. For example, in Table 1, $(p[0] = True) \wedge (q[1] = True)$ hold at t_0, and $z[2] = x[0] + y[0]$ hold at t_2.

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

A proposition trace is similarly defined as in an execution trace. We use the following time window as used in [15].

Definition 5. (***Time window***) *Given a trace (execution trace or proposition trace) $\tau = A_0 A_1 A_2 \dots A_{n-1}$, and two simulation instants t_i and t_j such that $0 \leq t_i \leq t_j \leq n-1$, a time window $TW[i, j] = A_i A_{i+1} \dots A_j$ is the subsequence of contiguous elements of α included between t_i and t_j.*

Table1. Example of an execution trace



EXECUTION TRACE					
time	p	q	x	y	z
t_0	*True*	*False*	2	2	1
t_1	*False*	*True*	1	1	2
t_2	*False*	*False*	2	3	3
t_3	*False*	*True*	0	4	2
t_4	*True*	*False*	0	4	5

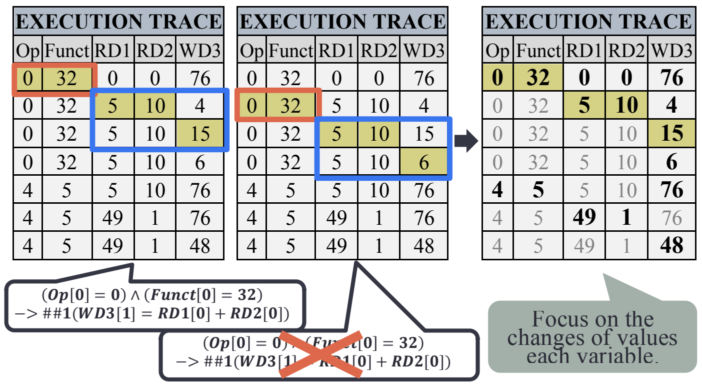
##2
$(\boldsymbol{p}[\mathbf{0}] = \boldsymbol{True}) \wedge (\boldsymbol{q}[\mathbf{1}] = \boldsymbol{True})$
$\boldsymbol{z}[\mathbf{2}] = \boldsymbol{x}[\mathbf{0}] + \boldsymbol{y}[\mathbf{0}]$

We extract word-level relations between variables over multiple clock cycles that hold in execution traces as atomic propositions, and mine implications between two propositions as assertions. For example, the property "if $p = True$ holds in a clock cycle and $q = True$ holds in the next clock cycle, x and y at one clock cycle later are added and its result appears two clock cycle later at z" holds in the execution trace shown in Table 1. This can be described in SVA as follows: $(p = True)$ ##1 $(q = True)$ |−> ##1 $(1, v_x = x, v_y = y)$ ##2 $(z = v_x + v_y)$. In SVA, "*always*" is assumed implicitly. In the following, that property is simply described as $(p[0] = True) \wedge (q[1] = True)$ $->$ ##2 $(z[2] = x[0] + y[0])$. "##n" represents the elapse of n clock cycles. "$->$" is the ordinary implication operator, which means $(p[0] = True) \wedge (q[1] = True)$ and ##2 $(z[2] = x[0] + y[0])$ have the same starting clock cycle t_0. Note that $z[2] = x[0] + y[0]$ is regarded as $True$ at the clock cycle in which x and y are referred to. The formula means that if $(p[0] = True) \wedge (q[1] = True)$ holds, $(z[2] = x[0] + y[0])$ will hold 2 clock cycles later. We refer to "##n" as an *offset* of a consequent.

In this paper, the assertions are mined within each time window of preset length. By determining the maximum number of clock cycles of an assertion, we can avoid mining relations between propositions that are too far apart. The propositions are also extracted within each time window.

Moreover, we consider the features of a digital circuit that often keep values unchanged over multiple clock cycles, and propose assertion mining by focusing on the changes of the values in execution traces. As a result, the unchanged values over multiple clock cycles can be considered as one value and we can mine assertions that cannot be mined in the existing method that consider all values. Table 2 shows a part of an execution trace of a multi-cycle MIPS processor. In the MIPS processors, the operation to be performed is determined by the values of the signal Op and $Funct$ representing the type of instruction. Multi-cycle processor executes one instruction over multiple clock cycles. For an example, if $Op = 0$ and $Funct = 32$ are read in the decode stage, the values of the specified registers $RD1$ and $RD2$ at one clock cycle later are added and its result appears one clock cycle later at $WD3$ and is written to the register. Such a property seems to be able to describe as $(Op[0] = 0) \land (Funct[0] = 32) \text{->} \#\#1\ (WD3[1] = RD1[0] + RD2[0])$. However, in actuality, this assertion does not hold in the execution trace, because the same values are kept in Op and $Funct$ until the next instruction is read. See the middle table of Table 2.

Table 2. Execution traces of multi-cycle MIPS processor


$(Op[0] = 0) \land (Funct[0] = 32)$ $\text{->} \#\#1(WD3[1] = RD1[0] + RD2[0])$
$(Op[0] = 0) \land (Fu\cancel{nct}[0] = 32)$ $\text{->} \#\#1(WD3[1 \cancel{] = RD}1[0] + RD2[0])$
Focus on the changes of values each variable.

EXECUTION TRACE				
Op	Funct	RD1	RD2	WD3
0	32	0	0	76
0	32	5	10	4
0	32	5	10	15
0	32	5	10	6
4	5	5	10	76
4	5	49	1	76
4	5	49	1	48

EXECUTION TRACE				
Op	Funct	RD1	RD2	WD3
0	32	0	0	76
0	32	5	10	4
0	32	5	10	15
0	32	5	10	6
4	5	5	10	76
4	5	49	1	76
4	5	49	1	48

EXECUTION TRACE				
Op	Funct	RD1	RD2	WD3
0	**32**	**0**	**0**	**76**
0	32	**5**	**10**	**4**
0	32	5	10	**15**
0	32	5	10	**6**
4	**5**	5	10	**76**
4	5	**49**	**1**	76
4	5	49	1	**48**

Thus, we perform extraction of propositions and mining of assertions by focusing on the changes of the values. In the right-most table of Table2, the values changed from the previous values are shown in bold. By considering only such values, we can extract the property $(Op[0] = 0) \land (Funct[0] = 32) \text{->} \#\#1\ (WD3[1] = RD1[0] + RD2[0])$ from the execution trace. Such a property becomes the assertion that holds in the execution trace by interpreting as $((Op[-1] != 0) \lor (Funct[-1] != 32)) \land (Op[0] = 0) \land (Funct[0] = 32) \text{->} \#\#1\ (WD3[1] = RD1[0] + RD2[0])$. On the other hand, we should not consider the changes of values for consequents of assertions in the assertion mining phase. Suppose that, in the left-most table of Table 2, the first line happens to be $0, 32, 5, 10, 4$ instead of $0, 32, 0, 0, 76$. Then, the assertion cannot be found, because $5, 10, 4$ in the second line is ignored, if we focus only changes of values.

4. Methodology

The proposed method consists of four phases:

1) **Extracting of atomic propositions:** Extract frequent word-level atomic propositions that expressing value assignment and relation between variables by analyzing the execution trace.
2) **Mining of propositions:** Mining frequent propositions over multiple clock cycles from atomic proposition trace in order to obtain candidate antecedents of assertions.
3) **Mining of assertions:** Mining assertions from the trace of frequent propositions obtained at phase 1 and 2.
4) **Pruning and combining of assertions:** Prune unnecessary assertions from the mined assertion set and combine consequents of assertions having the same antecedent with "∧".

We can consider that propositions which frequently hold in the execution trace can represent the behaviors of the DUV. In other words, it is possible to generate assertions with high coverage for DUV behavior by mining the relation between frequent propositions.

We can choose whether to focus on the changes of values or to handle all the values in the traces. Whether or not the values are kept over multiple clock cycles may be judged from the execution traces or the documentations. The above procedure 1 - 4 can be performed independently from these two approaches. In the following, we explain the approach in which we handle all the values in the traces.

4.1. Extracting of frequent atomic propositions

In the first phase of the proposed method, we analyze the execution trace of DUV and extract word-level atomic propositions (described in Section 3) that frequently hold.

In order to make assertion mining more efficient and obtain a high quality assertion set, we classify frequent atomic propositions as follows:

(A) Candidate of antecedents: a set of the atomic propositions representing value assignment within a single clock cycle (e.g., $a[0] = constant$).

(B) Candidate of consequents: a set of the atomic propositions that assign a value to the variables in V_a (e.g., $z[2] = a[0] + b[0]$, $z[1] = a[0] << 3$, $z[0] = constant$ $(z \in V_a)$).

Algorithm 1. Extraction of frequent atomic propositions

```
1:  Function $GetAtomicProps(T, max\_len, ant\_th, con\_th)$
2:    $ant\_a\_props = \{\}$
3:    $con\_a\_props = \{\}$
4:    $candidates = \{\}$
5:    $t_i = 0$
6:    while $t_i \leq (length(T) - max\_len)$ do
7:      $ap\_list = extractionAP(TW[t_i, t_i + max\_len - 1])$
8:      $candidates.extend(ap\_list)$
9:      $t_i = t_i + 1$
10:   $counted\_ap = Counter(candidates)$
11:   for all $< ap, m >\in counted\_ap$ do
12:     if $((ap\ is\ candidate\_of\_antecedents)$
                                and $(m \geq ant\_th))$ then
13:       $ant\_a\_props = ant\_a\_props \cup \{ap\}$
14:     if $((ap\ is\ candidate\_of\_consequents)$
                                and $(m \geq con\_th))$ then
15:       $con\_a\_props = con\_a\_props \cup \{ap\}$
16:   return $ant\_a\_props, con\_a\_props$
```

The procedure of frequent atomic proposition extraction is shown in Algorithm 1. The function $getAtomicProps$ takes as arguments an execution trace T, the length of time window for mining assertions max_len and two thresholds ant_th and con_th. ant_th is the thresholds of frequency for atomic propositions (A), con_th is the thresholds of frequency for atomic propositions (B). First, the function $GetAtomicProps$ extracts all atomic propositions that hold on each time window of length max_len (line 6-9). At each iteration, the time window is analyzed by the function $extractionAP$, and a list of atomic propositions ap_list that hold on the time window is extracted (line 7). In order not to extract the same atomic proposition that holds at the same time, the function $extractionAP$ extracts only atomic propositions including the variables that refer to values at time t_i, that is, the starting clock cycle of the time window. The atomic propositions in ap_list are added to $candidates$ (line 8).

Next, the frequency of each atomic proposition in ap_list is counted, and the set of pairs <*atomic proposition* p, *frequency of* p > is obtained by the function $Counter$ (line 11). Then, each proposition is classified as (A) or (B), and only propositions having a frequency exceeding the threshold value are selected (line 11-15). "$ap\ is\ candidate\_of\_antecedents$" in line 12 means "the atomic proposition ap is classified as (A)" and "$ap\ is\ candidate\_of\_consequents$" in line 14 means "the atomic proposition ap is classified as (B)." ant_a_props is the set of frequent atomic propositions that are candidate of antecedent of assertions and con_a_props is the set of frequent atomic propositions that are candidate of consequent of assertions.

Table 3. Example of frequent atomic propositions

EXECUTION TRACE						
time	p	q	r	x	y	z
t_0	0	32	$False$	5	1	0
t_1	0	34	$True$	10	5	0
t_2	43	5	$False$	20	8	6
t_3	0	34	$False$	3	30	5
t_4	0	32	$True$	6	7	35
t_5	4	2	$False$	4	4	-27
t_6	8	32	$False$	1	2	13

FREQUENT ATOMIC PROPOSITIONS

(A)	(B)
$p[0] = 0$	$z[2] = x[0] + y[0]$
$q[0] = 32$	$z[2] = x[0] - y[0]$
$q[0] = 34$	$z[0] = 0$
$r[0] = True$	
$z[0] = 0$	

For example, consider the execution trace shown in Table 3 (left). When $\{z\} = V_a$, ant_th and con_th are 2, the atomic propositions shown in Table 3 (right) are extracted. In this example, we do not consider "$False$" of Boolean variable.

The sets of frequent atomic propositions can be edited by users. It is possible to extract other type of atomic propositions specified by the users.

The frequent atomic propositions of antecedents are used as candidate propositions of antecedents, and also used for generating more complex propositions in the next phase. The frequent atomic propositions of consequents are used as candidate propositions of consequents as they are.

The time window or similar method is used in [15][9]. However, in these existing methods, the idea of the time window is used to obtain temporal relations between atomic propositions that hold in a clock cycle, and it is not used to obtain atomic propositions over multiple clock cycles.

4.2. Mining of frequent propositions

The purpose of this phase is to mine frequent propositions over multiple clock cycles from frequent atomic propositions as antecedents of assertions. By not considering all combinations of the atomic propositions as candidates of antecedents, it is possible to prevent mining the assertions overly constrained antecedents. In other words, it is possible to avoid extracting the assertions that hold accidentally in the execution trace.

In existing method [15], frequent relations among the frequent atomic propositions are not considered in assertion mining. All temporal patterns between frequent atomic propositions are considered as candidate antecedents of assertions. Therefore, it seems that this method needs to limit the maximum number of clock cycles of the assertions or the antecedents to some extent. In [16], frequent relations among atomic propositions that hold in each clock cycle are considered as propositions, but frequent relations over multiple clock cycles are not considered in assertion mining.

Mining of the frequent propositions is performed as follows:

i) Obtain the atomic proposition trace ω from the execution trace T and the set of frequent atomic propositions which are candidate antecedents obtained in previous phase.

ii) Consider each time window $TW[t_i, t_i + max\_len - 1]$ of length max_len for ω and generate an item list listing atomic propositions that hold in each time window.

iii) Extract frequent itemsets with a frequency greater than or equal to the minimum support min_sup from the item list by frequent pattern mining, and generate frequent propositions by connecting the atomic propositions in each large itemset with "∧". In order to prevent duplication of propositions, we extracted only propositions including the atomic propositions that hold at time t_i.

Table 4. Example of an item list and frequent propositions

Item List	
$TW[t_0, t_2]$	$p[0] = 0, q[0] = 32, z[0] = 0, p[1] = 0, q[1] = 34, r[1] = True, z[1] = 0$
$TW[t_1, t_3]$	$p[0] = 0, q[0] = 34, r[0] = True, z[0] = 0, p[2] = 0, q[2] = 34$
$TW[t_2, t_4]$	$p[1] = 0, q[1] = 34, p[2] = 0, q[2] = 32, r[2] = True$
$TW[t_3, t_5]$	$p[0] = 0, q[0] = 34, p[1] = 0, q[1] = 32, r[1] = True$
$TW[t_4, t_6]$	$p[0] = 0, q[0] = 32, r[0] = True, q[2] = 32$

Frequent proposition	support
$p[0] = 0$	0.8
$q[0] = 32$	0.4
$q[0] = 34$	0.4
$r[0] = True$	0.4
$z[0] = 0$	0.4
$(p[0] = 0) \land (q[0] = 32)$	0.4
$(p[0] = 0) \land (q[0] = 34)$	0.4
$(p[0] = 0) \land (z[0] = 0)$	0.4
$(p[0] = 0) \land (r[0] = True)$	0.4
$(p[0] = 0) \land (r[1] = True)$	0.4
$(p[0] = 0) \land (p[1] = 0)$	0.4
$(p[0] = 0) \land (p[1] = 0) \land (r[1] = True)$	0.4

Consider the example shown in Table 3. The item list obtained when $max\_len = 3$ is shown in Table 4. The propositions obtained from the item list by performing the frequent pattern mining with $min\_sup = 0.4$ are shown in the lower side of Table 4.

The assertions in the proposed method consist of pairs of a frequent proposition of candidate antecedents obtained in phase 1, 2 and a frequent proposition of candidate consequents obtained in the previous phase 1.

4.3. Mining of assertions

The purpose of this phase is to extract assertions of length less than max_len that hold in the execution trace T. An assertion is an implication from a proposition of candidate antecedents to a proposition of candidate consequents. The length of an assertion is the number of clock cycles necessary for the assertion to hold

Algorithm 2. Mining of assertions

```
1: Function $GetAssert(T, max\_len, Ants, Cons)$
2:   $Assert = \{\}$
3:   $FailedAssert = \{\}$
4:   $t_i = 0$
5:   while $t_i \leq (length(T) - max\_len)$ do
6:     $AT = GetPropTrace(TW[t_i, t_i + max\_len - 1], Ants)$
7:     $CT = GetPropTrace(TW[t_i, t_i + max\_len - 1], Cons)$
8:     $a\_set = GetProps(AT, Ants)$
9:     $c\_set = GetProps(CT, Cons)$
10:    for all $< a, 0 > \in a\_set$ do
11:      for all $c \in Cons$ do
12:        for each $offset$ in $max\_len$ do
13:          if $< a, c, offset >$ in $FailedAsert$ do
14:            $continue$
15:          if $< c, offset >$ in $c\_set$ do
16:            if $lenth(a) \leq offset + length(c)$ do
17:              $Assert = Assert \cup \{< a, c, offset >\}$
18:              $continue$
19:          if $< c, offset >$ not in $c\_set$ do
20:            if $< a, c, offset >$ in $Assert$ do
21:              $Assert = Assert \backslash \{< a, c, offset >\}$
22:            $FailedAssert$
                      $= FailedAssert \cup \{< a, c, offset >\}$
23:    $t_i = t_i + 1$
24:  return $Assert$
```

The procedure of assertion mining is shown in Algorithm 2. The function $GetAssert$ takes as arguments an execution trace T, the length of time window for mining assertions max_len and two sets $Ants$ and $Cons$. $Ants$ is the set of frequent propositions as candidate antecedent of assertions obtained in phase 1 and 2. $Cons$ is the set of frequent propositions as candidate consequent of assertions obtained in phase 1. First, the function $GetAssert$ obtains the proposition trace AT and CT of length max_len by the function $GetPropTrace$ (line 6-7). Next, the sets of propositions $< p, offset >$ that hold in each proposition trace are obtained (line 8-9). p is a proposition in $Ants$ or $Cons$, and $offset$ is the time at which the proposition p holds regarding t_i to be 0. Then, at each iteration, the function $GetAssert$ obtains two sets $Assert$ and $FailedAssert$ (line 10-23). In order to prevent duplication of assertions, we consider only the propositions with $offset = 0$ for the antecedents (line10). $Assert$ is the set of implications $< a, c, offset >$ that hold in the execution trace before t_i, and $FailedAssert$ is the set of implications $< a, c, offset >$ that does not hold in the execution trace before t_i. In line 16, in order not to mine inconsistent assertions about time such as $(p[0] = True) \wedge (q[1] = True) \; -> \; \#\#0 \; (z[0] = 1)$, the lengths of the propositions a and c are considered. $< a, c, offset >$ represents "$a -> \#\#offset\; c$." The above processes are performed for each time window. Finally, we can obtain the set $Assert$ of assertions that hold in the execution trace T.

4.4. Pruning and combining of assertions

The purpose of this phase is to improve the quality of the assertion set by pruning and combining. After pruning as follows, we combine the consequent of assertions having the same antecedent into one assertion with "∧". First, if there is an assertion α as below, we prune the assertion α.

(i) α assigns a value to the same variable $v[i]$ in antecedent and consequent.

Next, if there are assertions $\alpha1$ and $\alpha2$ as below, we prune assertion $\alpha2$. The pruning (iii) is performed to improve the readability of the assertion set by adopting the only simplest assertion when several assertions are extracted for one behavior.

(ii) $\alpha1$ and $\alpha2$ have the same consequent, and the antecedent of $\alpha2$ includes the antecedent of $\alpha1$.

(iii) $\alpha1$ and $\alpha2$ have the same antecedent and a consequent that assigns a value to the same variable $z[i]$ $(z \in V_a)$, the consequent of $\alpha1$ representing the value assignment and the consequent of $\alpha2$ representing the relation between variables.

5. Experimental Results

All experiments were performed on an Intel Corei5-4590@3.30GHz with 24GB RAM. We used four designs in our experiments. Single-cycle MIPS processor, Multi-cycle processor and Pipelined MIPS processor from [17]. CORDIC design from OpenCores [18]. CORIC (Coordinate Rotation Digital Computer) is an algorithm for computing transcendental functions like sine and cosine.

5.1. Mining of assertions

Table 5 reports the number of variables in V and V_a used for experiments ($|V|$, $|V_a|$), the average number of bits of variables in V ($size(var)$), the maximum clock cycle of mining assertions (max_len), the number of frequent antecedent / consequent propositions ($\#ant$, $\#con$), the number of mined assertions ($\#assert$), the average numbers of antecedent / consequent atomic propositions included in the mined assertions ($size(ant)$, $size(con)$) and the total time required for the mining assertion phase and pruning and combining phase ($time$). max_len was determined from the execution traces. We used execution traces of 10000 lines for the MIPS processors and 3000 lines for CORDIC design. We chose to focus on the changes of values for MIPS processors, and chose to handle all the values in the traces for CORDIC design. Each trace was obtained by random simulation. In the MIPS processors, simulations were performed using programs in which all instructions other than the Jump instruction appeared randomly. In the CORDIC design, the value of input variable θ was set randomly. In experiments on CORDIC design, "$theta < 0$" and "$theta \geq 0$" are added manually as atomic propositions. The thresholds in the proposed method were set to filter propositions which are obviously less frequent.

Table 5. Experimental results of assertion mining

DUV	|V|	|Va|	size(var)	max_len	#ant	#con	#assert	size(ant)	size(con)	time
Single-cycle MIPS processor	6	1	23.33	3	50	58	7	1.57	1.00	4.39s
Multi-cycle MIPS processor	6	1	23.33	5	55	34	14	1.29	1.21	3.08s
Pipelined MIPS processor	6	1	23.33	5	84	58	8	1.63	1.00	3.33s
CORDIC	6	2	14.33	2	20	29	9	1.10	2.60	1.24s

Table 5 shows that the proposed method can extract assertions in a few seconds for a given set of word-level variables. The following shows examples of the assertions extracted in the experiments on pipelined MIPS processor and CORDIC design.

(i) Pipelined MIPS processor

$$((OpD[-1] != 0) \vee (FunctD[-1] != 32)) \wedge$$
$$(OpD[0] = 0) \wedge (FunctD[0] = 32)$$
$$-> \#\#1\ (ResultW[2] = SrcAE[0] + SrcBE[0])$$

(ii) CORDIC

$$(init[0] = 0) \wedge (theta[0] \geq 0)$$
$$-> \#\#0\ (y[1] = y[0] + x\_shifted[0]) \wedge (x[1] = x[0] - y\_shifted[0])$$

The assertion (i) captures the behavior that add the values of registers according to the given instruction $(OpD[0] = 0) \wedge (FunctD[0] = 32)$, and the assertion (ii) captures the calculation of CORDIC design that obtain $cos\theta$ and $sin\theta$ when $\theta \geq 0$ and the reset signal is 0.

5.2. Effectiveness of frequent proposition mining

Table 6 shows the results of extracting assertions without frequent proposition mining over multiple clock cycles (explained in phase 2 in Section 4). In this experiment, all temporal patterns of frequent propositions that hold at one clock cycle are considered as candidate antecedents of assertions. Comparing the Table 5 with Table 6, it can be seen that the frequent proposition mining improves the execution time of assertion mining. In addition, the number of assertions and the size of antecedent are larger than shown in Table 5. This is because assertions in which these antecedents are excessively restricted that hold accidentally at the execution trace are extracted. By this experiment, it was confirmed that frequent proposition mining over multiple clock cycles is effective.

Table 6. Experimental results without frequent proposition mining over multiple clock cycles

DUV	#assert	size(ant)	size(con)	time
Single-cycle MIPS processor	324	3.17	1.43	68.56s
Multi-cycle MIPS processor	137	2.96	2.38	19.15s
Pipelined MIPS processor	3925	3.73	2.91	251.97s
CORDIC	31	1.71	5.23	1.78s

5.3. Mutant analysis

We performed mutant analysis to measure the quality of a set of assertions mined in the experiments of Table 5. A mutant is an artificial error of DUV, and the mutant coverage is the ratio between generated mutants and covered mutants. If an assertion fails on the DUV containing a mutant, the mutant is covered by the assertion. In our mutant analysis, the value of each output signal of the control unit in the MIPS processors, and the value of each signal of the set $V \backslash V_a$ in CORDIC are fixed at word-level, and the resulting DUVs are regarded as the mutants. The experimental results are shown in Table 7. $\#mutant$ is the number of mutants, $\#covered$ is the number of covered mutants and Avg is the average number of mutants covered by each assertion. The mutant coverage achieved for CORDIC design is 100% for the proposed method. The mutant coverage for the processors is not 100%, because there are mutants that do not affect V. Table 7 shows that the assertion sets mined by the proposed method can cover almost all the mutants affecting V.

Table 7. Results of mutant analysis

DUV	#mutant	#covered	Avg
Single-cycle MIPS processor	24	16	9.00
Multi-cycle MIPS processor	24	21	4.93
Pipelined MIPS processor	24	18	8.63
CORDIC	7	7	2.00

6. Conclusions

In this paper, we proposed a method that to extract relations over multiple clock cycles between variables as atomic propositions by analyzing execution traces and to generate assertions expressing the relations. The proposed method can also efficiently generate assertions by extracting frequent relations between atomic propositions over multiple clock cycles as propositions. The experimental results show that the proposed method can extract assertions including word-level relations over multiple clock cycles. In addition, the results of mutant analysis show that the assertions extracted by the proposed method can capture the behavior of DUV.

As future works, it is necessary to increase the form of the assertions that can be extracted, and to perform the experiments for larger designs.

References

[1] L. -C. Wang, M. S. Abadir, and N. Krishnamurthy, "Automatic generation of assertions for formal verification of powerpc microprocessor arrays using symbolic trajectory evaluation," Design Automation Conference (1998) 534-537.

[2] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," ACM Sigplan Notices, vol. 37, no. 1 (2002) 4-16.

[3] S. Hangal, S. Narayanan, N. Chandra and S. Chakravorty: "IODINE: a tool to automatically infer dynamic invariants for hardware designs," Design Automation Conference (2005) 775-778.

[4] G. Fey and R. Drechsler, "Improving simulation-based verification by means of formal methods," Asia South-Pacific Design Conference (2004) 640-643.

[5] B. Isaksen and V. Bertacco, "Verification through the principle of least astonishment," ICCAD (2006) 860-867.

[6] P. H. Chang and L. C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," Asia and South Pacific Design Automation Conference (2010) 607-612.

[7] W. Li, A. Forin and S. A. Seshia, “Scalable specification mining for verification and diagnosis,” Design Automation Conference (2010) 755-760.

[8] S. Vasudevan, D. Sheridan, D. Tcheng, S. Patel, W. Tuohy, and D. Johnson, “GoldMine: Automatic assertion generation using data mining and static analysis,” DATE (2010) 626-629.

[9] S. Hertz, D. Sheridan, S. Vasudevan, “Mining Hardware Assertions With Guidance From Static Analysis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,Volume 32 Issue 6 (2013) 952-965.

[10] L. Liu and S. Vasudevan, “Automatic Generation of System Level Assertions from Transaction Level Models,” J. Electronic Testing 29, 5 (2013), 669-684.

[11] L. Liu, C. H. Lin and S. Vasudevan, “Word level feature discovery to enhance quality of assertion mining,” IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2012) 210-217.

[12] M. Bonato, G. D. Guglielmo, M. Fujita, F. Fummi, G. Pravadelli, “Dynamic Property Mining for Embedded Software,” IEEE/ACM/IFIP international conference onf Hardare/softwere codesign and system synthesis (2012) 87-196.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao, “The daikon system for dynamic detection of likely invariants,” Sci. Comput. Program., 69(1-3) (2007) 35-45.

[14] A. Danese, T. Ghasempouri, G. Pravadelli, “Automatic extraction of assertions from execution traces of behavioral models,” Design, Automation & Test in Europe Conference & Exhibition (2015) 67-72.

[15] A. Danese, F. Filini and G. Pravadelli, “A time-window based approach for dynamic assertions mining on control signals,” IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC) (2015) 246-251.

[16] A. Danese, N. D. Riva, G. Pravadelli, “A-TEAM: Automatic template-based assertion miner,” Design Automation Conference (2017), Article No. 37.

[17] D. M. Harris, S. L. Harris, “Digital Design and Computer Architecture,” 2nd ed., Waltham, MA: Morgam Kaufmann (2012).

[18] OpenCores benchmarks. www.opencores.org.