Automatic Assume Guarantee Analysis for Assertion-Based Formal Verification

Dong Wang[†]
Synopsys Inc.
dongw@synopsys.com

Jeremy Levitt[†]
Mentor Graphics Corp.
jeremy_levitt@mentorg.com

***Abstract*— Assertion based verification encourages the insertion of many assertions into a design. Typically, not all assertions can be proven (or falsified). The indeterminate assertions require manual analysis in order to determine design correctness – a time-consuming and error-prone process. This paper shows how automatic assume guarantee reasoning can be used to reduce the amount of manual analysis. We present algorithms to automatically compute the assume guarantee relations between assertions. We extend circular assume guarantee reasoning to compute more proofs. And, we show how automatic assume guarantee reasoning can be used in practice to reduce the number of indeterminate assertions requiring manual analysis. We present the results of applying our algorithms to large industrial designs.**

I. Introduction

Assertion based verification (ABV) is a methodology for verifying correct design implementation. Assertions, in the form of safety properties, are added to the design as it is being coded. These assertions check for the correct behavior of the logic being coded as well as for correct behavior on the design interfaces. As the design and verification evolve, the number of assertions can become very large. It is typical for a design block to have hundreds of assertions. It is also typical that a considerable number of assertions are too difficult to be formally proven or falsified within the given resource limits. For crucial assertions, such as bus mutual exclusion (mutex) properties and logic equivalence checking (LEC) constraints, etc., manual inspection of each indeterminate assertion is frequently mandated. The consequence of these assertions failing is likely to be a completely non-functional chip. Manual inspection is clearly a tedious, time-consuming and error-prone process. Therefore, it is desirable to reduce the number of assertions that require manual analysis.

This paper describes automatic assume guarantee algorithms designed to exploit the high assertion density of ABV. Our algorithms automatically compute the assume guarantee relations between given sets of assertions. We exploit the automatically computed assume guarantee relations in two ways. Firstly, we use the assume guarantee relations to generate additional proofs. To increase the number of proofs obtained, we present an algorithm that extends circular assume guarantee reasoning to compute additional proofs. Secondly, we use the assume guarantee relations between the indeterminate assertions to reduce the number of indeterminate assertions that require manual analysis. An assume guarantee relation states that an assertion is implied (guaranteed) if a set of assumed assertions are valid. Since the correctness of the assertion is implied by a set of assumed assertions, only the assumed assertions need to be analyzed manually. We present a practical and simple method to organize this information so that fewer assertions need be manually analyzed.

[†]This work was done while the authors were at 0-In Design Automation.

A. Related work

The assume guarantee algorithms we present in this paper are different in important respects from those proposed by previous research. Most existing assume guarantee algorithms are manual [10] [9] [7]. Users are required to identify the assume guarantee relations between properties, and the algorithms then check the validity of the given relations and use them to generate proof obligations. In contrast, our algorithms automatically compute assume guarantee relations by enhancing existing SAT based temporal induction and abstraction refinement algorithms. Recently there has been some interesting research on automating assume guarantee, such as [1] [4]. [1] [4] focus on automatically synthesizing a suitable environment to guarantee a property, but this synthesis can be prohibitively expensive to perform. In contrast, our algorithms are designed for ABV where the number of assertions is usually large. Instead of constructing completely new assumptions, our algorithms efficiently identify useful assumptions within the set of user specified assertions.

Most previous research on assume guarantee is directed at generating more proofs. In this paper, the assume guarantee relations are also used to reduce the manual verification effort involved when assertions can not be formally proven or falsified. Other previous research uses dependencies between assertions to avoid redundant error reporting when design bugs are found. Our approach on the other hand, uses assume guarantee relations to help users verify their design even when no design bugs have been found yet.

B. Contributions

We have designed several new assume guarantee algorithms to not only compute proofs, but just as importantly to reduce the manual verification effort required for indeterminate assertions.

With ABV, it is impractical for users to specify the assume guarantee relations between assertions. Designers and verification engineers have neither the knowledge nor the inclination to perform such analysis. Thus, we show how to enhance existing SAT based temporal induction and abstraction refinement algorithms to automatically compute assume guarantee relations. Our algorithms are based on SAT unsatisfiability proofs and abstract counterexample simulation. Similar to the use of constraints, assumptions are used to strengthen induction hypothesis for temporal induction, and to generate smaller abstractions for abstraction refinement.

We present an algorithm, called `Guarantee`, that uses the automatically computed assume guarantee relations to deduce guaranteed assertions from a given arbitrary set of assumed assertions. The traditional application of the circular assume guarantee rule [9] (described in Section II-B) is a special case of our algorithm, specifically when (1) the given set of assumed assertions is empty and (2) each property is implied by a single set of assumptions. Our algorithm allows a non-empty set of assumed assertions, enabling the deduction of general dependencies between assertions (in addition to proofs deduced from the empty set of assumed assertions). Our algorithm handles multiple sets of assumptions per property and this enables proofs to be deduced in the presence of zero-delay cycles (described in Section II-B).

We also present an algorithm, called `Guide`, that distills the information represented by all the automatically computed assume guarantee relations into an organized and actionable format, thereby reducing the manual effort required to analyze the indeterminate assertions. This is of practical importance, since a direct presentation of all the assume guarantee relations is more complicated than most designers and verification engineers are prepared to accept: the number of properties having assumptions can be large, an assumption of a property can itself have assumptions, the assume guarantee relations may contain loops, and there are different assumption types (zero-delay or unit-delay) to keep track of.

C. Organization

The remainder of the paper is organized as follows. Section II introduces notation and the circular assume guarantee rule. Section III describes automatic algorithms to compute assume guarantee relations. Section IV describes the *Guarantee* algorithm to compute guarantees for an arbitrary set of assertions. Section V describes the `Guide` algorithm to generate organized and actionable assume guarantee reports. Section VI presents experimental results. Finally, Section VII contains conclusions.

II. Background

A. Notations

In this paper, the design under verification is modeled by a machine $M = \langle F, X, I, R, P \rangle$, where F is the set of primary inputs, X is the set of registers, $I(X)$ is the initial state predicate, $R(X, F, X')$ is the next state predicate, and $P \subseteq X$ is the set of assertions. An assertion $p \in P$ is true under M if and only if $M \models \Box p$, i.e. p holds in all reachable states of M. In bounded model checking [2] (BMC) and temporal induction [11], machine M is unrolled to different times. Let F^k and X^k be the unrolled input and register variables at time k. Given a set of properties $C \subseteq P$, a single property p and an unroll k, the BMC and temporal induction for p under constraints C are defined as:

$$\begin{aligned}\texttt{BMC}_k(C,p) &= I(X^0) \wedge R(X^0,F^0,X^1) \wedge \\ &\quad C(X^1) \wedge R(X^1,F^1,X^2) \wedge \cdots \wedge \\ &\quad C(X^{k-1}) \wedge R(X^{k-1},F^{k-1},X^k) \wedge \\ &\quad C(X^k) \wedge \neg p^k \qquad (1)\\ \texttt{IND}_k(C,p) &= C(X^0) \wedge p^0 \wedge R(X^0,F^0,X^1) \wedge \cdots \wedge \\ &\quad C(X^{k-1}) \wedge p^{k-1} \wedge R(X^{k-1},F^{k-1},X^k) \wedge \\ &\quad C(X^k) \wedge \neg p^k \qquad (2)\end{aligned}$$

B. Circular Assume Guarantee Reasoning

Given a machine M, a property $p \in P$ and two disjoint sets of properties $A_p^+ \subseteq P$ and $A_p^0 \subseteq (P \setminus \{p\})$, circular assume guarantee reasoning [9] deals with triples of the form $\langle\langle A_p^+, A_p^0\rangle, M, p\rangle$, which asserts that in machine M, assuming A_p^+ up to time t–1 and A_p^0 up to time t can deduce p up to time t. A_p^+ is called the unit-delay assumptions for p, and A_p^0 is the set of zero-delay assumptions for p. We will use $A_p = A_p^+ \cup A_p^0$ to denote all the assumptions for p, and $\langle A_p, M, p\rangle$ to denote the triple. We say p is *implied* by assumptions A_p. With zero delay assumptions, care must be taken to avoid false "proofs". For example if two false assertions are equivalent, each assertion is implied by the other, yet neither is true. Circular assume guarantee reasoning requires non-zero-delay (i.e. unit-delay) cycles to generate proofs.

Formally, $M \models \Box(\wedge_{p_i \in P}\ p_i)$, if

- every property p_i is implied by assumptions, i.e. $\langle\langle A_{p_i}^+, A_{p_i}^0\rangle, M, p_i\rangle$,
- and there exists a well founded order $\preceq$ on P, such that $\forall q \in A_{p_i}^0\ (q \preceq p_i)$.

For formal hardware verification of safety properties without fairness constraints, proving $\langle\langle A_{p_i}^+, A_{p_i}^0\rangle, M, p_i\rangle$ is reduced to an induction problem or a reachability problem (see Section III for details).

During verification, each property $p \in P$ can be in one of four possible states representing the current verification result: *unknown*, *true*, *false* or *implied*. Initially all properties are unknown. A property becomes true or false when we find either a proof or counterexample respectively for it. A property becomes implied when we find a set of assumptions that imply it. We use $P_{\texttt{unknown}}, P_{\texttt{true}}, P_{\texttt{false}}, P_{\texttt{impld}}$ to represent the corresponding set of assertions. As discussed, we allow a single property p to have more than one set of assumptions $A_{p,j}$, where $1 \leq j \leq k_p$ and $\langle A_{p,j}, M, p\rangle \wedge (A_{p,j} \cap P_{\texttt{true}} = \emptyset) \wedge (A_{p,j} \cap P_{\texttt{false}} = \emptyset)$. We use A_p to denote any of the valid set of assumptions for p. Whenever a property $q \in A_p$ is proven, it is removed from A_p. If this makes A_p empty, p is proven to be true. On the other hand, if q is shown to be false, A_p is no longer a valid set of assumptions for p and the

entire set is removed. If it is the only set of assumptions for p, then p becomes unknown. Our algorithms may also directly prove or falsify p. A property is *indeterminate* if it is either unknown or implied.

For a set of properties $C \subseteq P$, we use $M \models \Box C$ as a shorthand for $M \models \Box(\wedge_{c_i \in C}\ c_i)$. We say p is *guaranteed* by C if and only if $M \models (\Box C \rightarrow \Box p)$. Intuitively, if C is true under M then any guaranteed property must be true under M. It is informative to note that $\langle C, M, p\rangle$ is a stronger statement than $M \models (\Box C \rightarrow \Box p)$. For example, let $M = \langle \emptyset, \{p, q\}, I, R, \{p, q\}\rangle$, such that:

$$\begin{aligned} I(p) = I(q) &= 1 \\ R(p) &= 0 \\ R(q) &= p \end{aligned}$$

It is easy to see that the only trace of M is:

time	p	q
$t = 0$	1	1
$t = 1$	0	1
$t \geq 2$	0	0

Thus, $M \models \Box q \rightarrow \Box p$ holds because there is no path where q holds globally. $\langle\langle \emptyset, \{q\}\rangle, M, p\rangle$ on the other hand does not hold, because at time $t = 1$, property q is 1, but p is 0 and so q cannot imply p.

III. Identify Assumptions

In this section, we show how to enhance existing formal verification algorithms to compute assumptions automatically.

A. Temporal Induction

SAT based temporal induction [11] [5] complements BMC by establishing an upper bound on the length of the shortest counterexample. This bound is typically much smaller than other bounds, such as diameter or recurrence diameter. However, as is well known, temporal induction often fails due to inadequate induction hypotheses. We use the following simple SAT based verification algorithm as the basis of our enhancement.

```
// m is the given bound
Algorithm Bmc_Ind(M, p, m)
1 for k = 0, ..., m − 1
2   if BMC_k(∅, p) is satisfiable
3     return false
4 for k = 1, ..., m
5   if IND_k(P_true, p) is unsatisfiable
6     return true
7 return unknown
```

Fig. 1. Bounded model checking and temporal induction

Let $\overline{P_{\texttt{false}}} = P \setminus P_{\texttt{false}}$. Next we show how to use $\overline{P_{\texttt{false}}}$ to strengthen induction without sacrificing soundness. The following example is used to motivate our approach. Let machine M have 8-bit inputs in_1, in_2, 8-bit registers $cntr_1, cntr_2$, and property registers w_1, w_2. The initial state and next state functions are:

$$\begin{aligned} I(cntr_1) &= 17 \\ I(cntr_2) &= 13 \\ R(cntr_1) &= w_1 == w_2\ ? \quad cntr_1 * 3 - in_1 * 2\ :\ cntr_1 + in_1 \\ R(cntr_2) &= w_1 == w_2\ ? \quad cntr_2 * 5 - in_2 * 2\ :\ cntr_2 + in_2 \\ I(w_1) &= 1 \\ I(w_2) &= 1 \\ R(w_1) &= cntr_1[0] \\ R(w_2) &= cntr_2[0] \end{aligned}$$

It is easy to see that the two assertions w_1, w_2 are true since values in $cntr_1$ and $cntr_2$ are always odd numbers. But, algorithm Bmc_Ind fails to prove w_1 and w_2 regardless of the induction depth. The induction, $\texttt{IND}_k(C, p)$, can be easily falsified because it ignores the initial state of the machine. For example, Table I shows a failed induction for w_1 with 4 unrolls. However, if both w_1 and w_2 at time $t-1$ and time t are assumed, we can easily prove w_1 at time $t+1$ and similarly for w_2. Based on the circular rule in Section II-B, both w_1 and w_2 are proven.

time	w_1	$cntr_1$	in_1	w_2	$cntr_2$	in_2
0	1	1	2	0	0	0
1	1	3	2	0	0	0
2	1	5	1	0	0	0
3	1	6	0	0	0	0
4	0	6	0	0	0	0

TABLE I
Induction counterexample for w_1

Figure 2 shows the modifications of Line 4 to Line 6 in Figure 1. For each unroll k, when proving p we use as constraints all the properties not known to be false. In the case of unsatisfiability, we extract the unsatisfiable core [3] from the SAT solver (Line 2). Let $U = \{p_i^j \mid (p_i \neq p) \wedge (p_i^j \in \text{unsat\_core}) \wedge p_i \in (P_{\texttt{unknown}} \cup P_{\texttt{impld}})\}$ be a subset of property variables that appear in the unsatisfiable core. U is the set of assumed assertions responsible for the success of the inductive hypothesis. If U is empty (Line 3), p is proven without assumptions and the algorithm terminates with a proof (Line 4). Otherwise, an assume guarantee relation $\langle\langle A_p^+, A_p^0\rangle, M, p\rangle$, where $A_p^0 = \{p_i \mid p_i^k \in U\}$ and $A_p^+ = \{p_i \mid p_i \notin A_p^0 \wedge \exists j <$

```
1 for k = 1, ..., m
2   if IND_k(P_false, p) is unsatisfiable
3     if no assumptions are required for unsat
4       return true
5     else
6       add properties in unsat proof as assumptions for p
```

Fig. 2. Temporal induction with assumptions

k $(p_i^j \in U)\}$ has been computed. Even if p is implied by assumptions at unroll k (Line 6), it may be possible to prove p at a larger unroll using fewer or no assumptions because larger unrolls will apply constraints in more time frames. In general, proofs from SAT solvers are highly affected by the state of the SAT database and SAT heuristics used. We purposely vary the decision heuristics. As a result, assumptions identified in different unrolls may not subsume each other. We record every set of assumptions for each property p, unless it contains an existing set of assumptions or p is proven without assumptions.

B. Abstraction Refinement

Existential abstraction refinement is an effective technique for combating the state explosion problem associated with model checking. The objective of abstraction-refinement algorithms is to generate a small abstraction that both preserves the validity of a property and is amiable to direct model checking. We show how assumptions can be used to help reduce the size of these abstractions.

We first introduce a BDD based reachability algorithm which computes $\langle A_p, M, p\rangle$ and separates the unit-delay assumptions in A_p from the zero-delay assumptions in A_p. The algorithm is based on the standard forward reachability algorithm:

Algorithm Reach(M, A_p, p)
  $reach = I$
  $new = reach$
  while $new \neq \emptyset$
    $to = img[M](new) \wedge A_p$
    $new = to \wedge \neg reach$
    $reach = reach \vee to$
  return $reach$

Essentially, reachability is restricted to the set of states that satisfy all properties in A_p. $\langle A_p, M, p\rangle$ holds when the reachable states do not violate p. Zero-delay assumptions are identified during reachability when p is violated in a state s that is excluded because s also violates $q \in A_p$. All the other assumptions are unit-delay assumptions.

We have enhanced the counterexample guided localization reduction algorithm in [12], and the proof-based without counterexample algorithm in [8], [6] to identify useful assumptions. The refinement algorithm in [12] is based on simulating the abstraction counterexamples on the concrete machines using 3-value simulation. Our enhancement of this algorithm is very simple. A property $q \in \overline{P_{\texttt{false}}}$ is added to the abstraction, whenever the 3-value simulation value for q becomes 0. Therefore by assuming q, the abstract counterexample is invalidated.

Abstraction without counterexamples [8] [6] uses unsatisfiable BMC instances to construct abstraction. The circuit nodes that appear in the unsatisfiable core of $\texttt{BMC}_k(\emptyset, p)$ are extracted and model checked to see whether they are also valid for an unbounded proof. Our enhancement is to create a smaller abstraction based on the unsatisfiability proof of $\texttt{BMC}_k(\overline{P_{\texttt{false}}}, p)$. Assuming that the properties in $\overline{P_{\texttt{false}}}$ are impossible to violate in time frames 1 to k, the two SAT problems $\texttt{BMC}_k(\emptyset, p)$ and $\texttt{BMC}_k(\overline{P_{\texttt{false}}}, p)$ are equivalent in that they are both unsatisfiable. However, the size of the unsatisfiability core for the latter can be much smaller than the former. A trivial example is $\texttt{BMC}_k(\{p\}, p)$, which does not use any circuit information to determine unsatisfiability.

IV. Compute Guarantees

In this section, we show how to compute guaranteed assertions for a given arbitrary set of assumptions using the computed assume guarantee relations. Our new algorithm is based on a proof graph similar to an AND-OR graph:

Definition 4.1: A *proof graph* $\mathcal{G}$ is a directed cyclic graph with 5-tuple $\langle \mathcal{T}, \mathcal{A}, \mathcal{O}, \mathcal{E}, \mathcal{E}^+\rangle$, where

- $\mathcal{T}$ is the set of terminal vertices. Each property $p \in P_{\texttt{unknown}} \cup P_{\texttt{impld}}$ has a corresponding terminal vertex $\tau_p \in \mathcal{T}$.
- $\mathcal{A}$ is a set of AND vertices. Each set of assumptions $A_{p,j}$ for $p \in P_{\texttt{impld}}$ corresponds to an AND vertex α_p^j. For each assumption $q \in A_{p,j}$, there is an edge $\langle \tau_q, \alpha_p^j\rangle \in \mathcal{E}$.
- $\mathcal{O}$ is a set of OR vertices. Each implied property $p \in P_{\texttt{impld}}$ has a corresponding OR vertex o_p. There is an edge $\langle o_p, \tau_p\rangle \in \mathcal{E}$. If p has k_p sets of assumptions, there are k_p edges $\langle \alpha_p^j, o_p\rangle \in \mathcal{E}$, for $j \in \{1, \ldots, k_p\}$.
- $\mathcal{E}^+ = \{\langle \tau_q, \alpha_p^j\rangle \mid q \in A_{p,j}^+\}$ represents the unit-delay edges in $\mathcal{E}$. All the other edges in $\mathcal{E}$ have zero-delay.

We also let $\mathcal{V} = \{\mathcal{T} \cup \mathcal{A} \cup \mathcal{O}\}$.
For example, Figure 3 shows a proof graph. In this graph, there

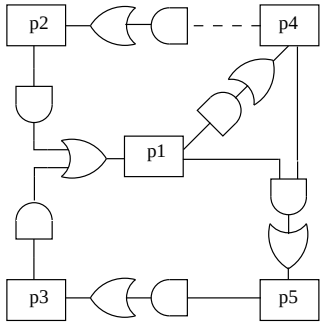

Fig. 3. An example proof graph

are 5 implied properties p_1 to p_5. The dashed line between p_4 and p_2 represents a unit-delay edge; all the other edges have zero-delay. p_1 is guaranteed assuming either p_2 or p_3. Thus p_1 has two sets of assumptions. p_5 is guaranteed if p_1 and p_4 both assumed. Based on model checking results, we know there is a t, such that all five properties hold from time 0 to time $t-1$. From Figure 3, we have

$$p_4^{t-1} \rightarrow p_2^t \rightarrow p_1^t \rightarrow p_4^t$$
$$\downarrow$$
$$p_5^t \rightarrow p_3^t$$

Therefore, by induction all five properties are true. Although there is a zero-delay cycle p_1, p_5, p_3, due to the existence of more than one set of assumptions for p_1 it is possible to generate real proofs.

Figure 4 shows the pseudo-code for `Guarantee`. For a given proof graph $\mathcal{G}$ and a subset $C \subseteq (P_{\texttt{unknown}} \cup P_{\texttt{impld}})$, it can be shown that $M \models \Box C \rightarrow \Box\texttt{Guarantee}(\mathcal{G}, C)$. $\texttt{Guarantee}(\mathcal{G}, \emptyset)$ generates real proofs without assumptions. In this algorithm, the possible truth values for a graph vertex are $0, 1, X$, where 0 represents `false`, 1 represents `true`, and X could be either `true` or `false`. We define function $\phi_w : \mathcal{V} \rightarrow \{0, 1, X\}$ as a mapping from vertices in iteration w (of the code block starting on Line 7) to truth values. At iteration 0, $\phi_0(v)$ is 1 for every vertex v. From Line 7 to Line 27, ϕ_w is calculated using event driven simulation. The code between Line 20 to Line 24 warrants detailed explanation. For a unit-delay assumption, values for the previous iteration are used; otherwise, values from the current iteration are used. Using lattice $1 \succeq X \succeq 0$, it is possible to prove that this algorithm performs a greatest fixed point computation $\langle 1, \ldots, 1\rangle = \phi_0 \succeq \phi_1 \succeq \cdots \succeq \phi_\infty$. When the fixed point is reached (Line 28), the set of guaranteed properties is $\{p \mid p \in (P_{\texttt{impld}} \setminus C) \wedge \phi_\infty(\tau_p) == 1\}$.

```
// G is the proof graph
// C is a given set of assumed properties
Algorithm Guarantee(G, C)
 1  φ_0(v) = 1, ∀v ∈ V
 2  w = 0
 3  while (1)
 4    w = w + 1
 5    φ_w(v) = X, ∀v ∈ V
 6    push every vertex into event queue
 7    while the event queue is not empty
 8      pop vertex v out of the event queue
 9      if v is τ_p, where p ∈ C
10        φ_w(v) = 1
11      elsif v is τ_p, where p ∈ P_unknown
12        φ_w(v) = 0
13      elsif v is τ_p, where p ∈ P_impld
14        let φ_w(v) have its input o_p's value
15      elsif v is an OR vertex
16        let φ_w(v) be the OR of its inputs' values
17      else
18        v must be an AND vertex
19        φ_w(v) = 1
20        foreach input edge e = ⟨τ_q, v⟩
21          if e ∈ E+
22            φ_w(v) &= φ_{w-1}(τ_q)
23          else
24            φ_w(v) &= φ_w(τ_q)
25      if φ_w(v) has changed
26        push all its fanout vertices into event queue
27    endwhile
28    if φ_w == φ_{w-1}
29      return {p | p ∈ (P_impld \ C) ∧ φ_w(τ_p) == 1}
30  endwhile
```

Fig. 4. Calculate the guarantees from assumptions

V. Generate Assume Guarantee Report

For the remaining indeterminate properties, we generate a report that restricts the manual analysis of these properties to a potentially much smaller set of properties. An example of such a report is shown in Figure 5. In this figure, there are four implied properties listed under "Prove". To guarantee these implied properties, there are four assumptions whose correctness need to be established. If the first of the four assumptions is proven, two (i.e. `2` in Figure 5) implied properties are guaranteed to be correct. If the second assumption is proven, one more (i.e. the first `+1` in Figure 5) implied property is guaranteed to be correct. Finally, if the last two assumptions are proven, one more (i.e. the last `+1` in Figure 5) implied property is guaranteed to be correct. Note that the last property requires all the four assumptions in the table. Using such a report, users can follow the suggested order to discharge all the required assumptions. For users who wish to see the assumptions for each implied property listed separately, we also provide the detailed assume guarantee relations.

```
----------------------------------------
Implied Properties (4)
----------------------------------------
Assume: custom zi_pci_io.slv_mon.BS13
Prove (2)
  custom checker_control.ps01110_1100
  custom checker_control.ps01110_0110
Assume: custom checker_control.ts0010
Prove (+1)
  custom cs3232.zi_pci_io.mas_mon.BS04
Assume: custom zi_pci_io.slv_mon.BS12
Assume: custom zi_pci_io.slv_mon.BS10
Prove (+1)
  custom cs3232.zi_pci_io.slv_mon.BS04
----------------------------------------
```

Fig. 5. An assume guarantee report

Figure 6 shows the pseudo-code for `Guide` to generate an assume guarantee report. In this figure, C is the set of selected assumptions and G is the set of implied assertions that can be guaranteed by C. Between Line 2 and Line 9, assumptions are identified and propagated using `Guarantee` until all implied properties are guaranteed. R associates each assumption with the number of properties that become guaranteed. At Line 10, R is sorted so that assumptions with more guaranteed properties are ordered first. The assumptions and their guarantees are then printed between Line 12 and Line 17. The `DFS` algorithm at Line 4 finds an assertion q in the transitive fanin of τ_p that is not in $C \cup G$. It is easy to see that by adding q into C, p is closer to being guaranteed. C can include both unknown properties and implied properties. Note that it is also possible to improve the ordering of assumptions by iterating this algorithm more than once.

// $\mathcal{G}$ is the proof graph
Algorithm Guide($\mathcal{G}$)
1 $C = G = \emptyset$
2 **while** $((C \cup G) \not\supseteq P_{\texttt{impld}})$
3 pick any $p \in P_{\texttt{impld}} \setminus (C \cup G)$
4 τ_q = DFS($\mathcal{G}, \tau_p, (C \cup G)$)
5 $C = C \cup q$
6 N = Guarantee($\mathcal{G}, C$)
7 $R = R \cup \{\langle q, |N| - |G| \rangle\}$
8 $G = N$
9 **endwhile**
10 sort R in descending order of the second elements
11 $C = G = \emptyset$
12 **foreach** ($\langle q, n \rangle \in R$)
13 $C = C \cup q$
14 $N =$ Guarantee($\mathcal{G}, C$)
15 **print** (q, $N \setminus G$)
16 $G = N$
17 **endfor**

Fig. 6. Algorithm to generate an assume guarantee report

VI. Experiments

In this section, we report experimental results for eight industrial designs of various sizes and complexities. The largest design has over 135k registers and 900k gates. The smallest design has 600 registers and 2k gates. The numbers of assertions per design range from 82 to 6k. Our experiments were run on a 3.2Ghz Pentium4 with 3.4GB memory running RedHat Linux 7.2. Table II shows the results for designs D1 to D8. The second and third columns contain the number of register and assertion for each design. Columns 4 and 5 compare the number of assertions that remain for further analysis (1) using and (2) without using our automatic assume guarantee algorithms respectively. When the algorithms in this paper are disabled, we report the number of unknown properties; otherwise, we report the sum of the unknown properties and the number of assumptions (identified by Algorithm Guide in Section V). Based on this table, automatic assume guarantee can reduce the number of assertions remaining for manual analysis by $(899 - 664)/899 = 26\%$ on average. This represents a significant saving of users' time and effort. Columns 6 and 7 compare the runtime (1) using and (2) without using assume guarantee. The overhead of our assume guarantee algorithms is pretty low. The average slowdown is $(6763 - 5976)/5976 = 13\%$. The last column reports the number of real proofs computed by the Guarantee algorithm in Section IV. Note that except for 1 proof in design D4 and 2 proofs in design D5, these proofs are also proven without using assume guarantee. However, finding these proofs through direct brute force methods was typically much more computationally expensive than using our assume guarantee reasoning and leveraging proofs already found for other properties. This in part explains the low overhead of our assume guarantee algorithms, despite the amount of analysis they perform.

Design	Reg	Prop	Remaining		Time (secs)		Proof
			With	W/O	With	W/O	
D1	2k	363	35	150	56	52	14
D2	50k	219	50	74	795	532	0
D3	68k	739	93	126	2472	2296	46
D4	55k	6476	40	54	1475	1407	1/96
D5	65k	375	215	245	571	367	2/3
D6	17k	132	79	87	71	43	1
D7	135k	418	101	109	1212	1169	15
D8	600	82	51	54	111	110	0
Total		8804	664	899	6763	5976	3/175

TABLE II
Experiments on eight industrial designs.

VII. Conclusions

We have presented automatic assume guarantee algorithms for assertion based formal verification. Our algorithms exploit the large number of assertions in an ABV methodology to automatically compute the assume guarantee relations. Experiments with industrial designs demonstrate that a significant number of assertions can be excluded from manual analysis. Our approach is easily introduced into an existing ABV methodology. No manually supplied assume guarantee relations are required. Users are provided with an ordered list of assertions to analyze. Users do not need to know the detailed assume guarantee relations.

References

[1] R. Alur, L. Alfaro, T. Henzinger, and F. Mang, "Automating modular verification," in *CONCUR*, 1999, pp. 82–97.
[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999.
[3] P. Chauhan, E. Clarke, S. Sapra, J. Kukula, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using sat based conflict analysis," in *FMCAD*, 2002.
[4] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu, "Learning assumptions for compsotional verification," in *TACAS*, 2003.
[5] N. Een and N. Sorensson, "Temporal induction by incremental sat solving," in *First International Workshop on Bounded Model Checking*, 2003.
[6] A. Gupta, M. Ganai, P. Ashar, and Z. Yang, "Iterative abstraction using sat-based bmc with proof analysis," in *ICCAD*, 2003.
[7] T. Henzinger, S. Qadeer, and S. Rajamani, "You assume, we guarantee: methodology and case studies," in *CAV*, 1998, pp. 440–451.
[8] K. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *TACAS*, 2003.
[9] K. L. McMillan, "Circular compositional reasonining about liveness," in *CHARME*, 1999.
[10] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logic and Models of Concurrent Systems*, 1984, pp. 123–144.
[11] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a sat-solver," in *FMCAD*, 2000.
[12] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano, "Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines," in *DAC*, 2001.