Efficient Verification of Multi-Property Designs (The Benefit of Wrong Assumptions)

Eugene Goldberg, Matthias Güdemann, Daniel Kroening
Diffblue Ltd.
Oxford, UK

Rajdeep Mukherjee
University of Oxford
UK

***Abstract*—We consider the problem of efficiently checking a set of safety properties $P_1, \ldots, P_k$ of one design. We introduce a new approach called JA-verification, where JA stands for "Just-Assume" (as opposed to "assume-guarantee"). In this approach, when proving a property P_i, one assumes that every property P_j for $j \neq i$ holds. The process of proving properties either results in showing that $P_1, \ldots, P_k$ hold without any assumptions or finding a "debugging set" of properties. The latter identifies a subset of failed properties that are the first to break. The design behaviors that cause the properties in the debugging set to fail must be fixed first. Importantly, in our approach, there is no need to prove the assumptions used. We describe the theory behind our approach and report experimental results that demonstrate substantial gains in performance, especially in the cases where a small debugging set exists.**

1. INTRODUCTION

The advent of powerful model checkers based on SAT [1]–[4] has created a new wave of research in property checking. This research has been mostly focused on algorithms that verify a single property for a given design. However, in practice, engineers write many properties for one design (sometimes hundreds and even thousands). This demands efficient and scalable techniques for automatic verification of multiple properties for one design.

More specifically, the problem we address is as follows. We are given a transition relation and a set of initial states, which specify the design. In addition, we are given a set of safety properties $P_1, \ldots, P_k$ that are *expected to hold*. (In Section 5, we consider the case where some properties are expected to fail.) We want to check if every property P_i holds. If not, we want to have an efficient way to identify failed properties that point to wrong design behaviors. (Thus, identification of *all* failed properties is not mandatory.) One way to solve this problem is to check whether the property $P := P_1 \wedge \ldots \wedge P_k$ holds. We will call P the *aggregate* property. If P holds, then all properties P_i are proven. Otherwise, the generated *counterexample* (CEX for short) identifies a subset of the failed properties, but no information is gained about the remaining properties. The latter can be verified by removing the failed properties from the set, and re-iterating the procedure with a new aggregate property.

In this paper, we study an alternative approach where the properties P_i are verified separately. We will refer to this approach as *separate verification* as opposed to *joint verification* of a set of properties. We will highlight three main reasons for our interest in separate verification. First, we want to study multi-property verification in the context of an IC3-like model checker [3]. Such a model checker will benefit from separate verification by generating proofs that take into account the specifics of each property. Besides, a property P_i is a weaker version of the aggregate property P. Thus, proving P_i should be easier than P. Second, each property P_i is an over-approximation of the same set of reachable states. Therefore, inductive invariants of already proven properties can be re-used. The third reason is as follows. If all P_i are true, there is a common proof of this fact, namely a proof that P holds. However, if some P_i properties fail, there may not be one universal CEX that explains all failures if the latter are property specific. Separate verification is more relevant in such a context.

In this paper, we introduce a version of separate verification called JA-verification. Here, JA stands for "Just-Assume", as opposed to "assume-guarantee". In JA-verification, one proves property P_i, assuming that every other property P_j, for $j \neq i$, holds, regardless whether it is true. We will call such a proof *local* as opposed to a *global* proof that P_i is true where no assumptions are made. JA-verification results in constructing the set of properties P_i that failed locally (if any). We show that if the aggregate property P fails, there is at least one property P_i that fails *both* globally and locally. Thus, if *all* properties P_i hold locally it also means they hold globally.

If P_i holds locally, it either holds globally as well or every CEX that breaks P_i fails some other property P_j *before* this CEX fails P_i. That is, a CEX for P_i contains a shorter CEX for P_j. This suggests that if P_i holds locally, its failure (if any) is most likely *caused by* failures of other properties. For this reason, we call the set of properties that *fail* locally a *debugging set*. This debugging set of properties points to design behaviors that need to be fixed in the first place. The approach guarantees that the failure of a property from the debugging set is not preceded by a failure of any other property. JA-verification constructs a debugging set as follows: when proving P_i it assumes that all P_j, for $j \neq i$, hold even when some of them fail. Thus, even the wrong assumption that all P_j, $j \neq i$ hold proves to be useful. For that reason, we use the term "Just Assume" to name our approach.

To improve the efficiency of JA-verification, we exploit the fact that IC3 proves a property by strengthening it to make this

Supported by ERC project 280053 "CPROVER", the H2020 FET OPEN 712689 SC^2 and SRC contracts no. 2012-TJ-2269 and 2016-CT-2707.

property inductive. Specifically, we show that the strengthening clauses generated by IC3 can be re-used when making any other property inductive if the same transition relation and initial states are used. Thus, in JA-verification, clauses generated to make P_i inductive are re-used when proving P_j, $j \neq i$.

Our contribution is threefold. First, we describe a new method of multi-property verification called JA-verification (see Section 4). It is based on the machinery of local proofs (Sections 2 and 3) and re-use of strengthening clauses (Section 6). Second, we show that JA-verification generates CEXs only for a special subset of failed properties called a debugging set. This is very important since computing a CEX can be very expensive (e.g., if a counter is involved). Third, we provide implementation details (Section 7) and experimental results showing the viability of JA-verification (Section 8).

2. Local And Global Proofs

Separate verification is based on the assumption that, in general, proving P_i is easier than $P_1 \wedge \ldots \wedge P_k$ because P_i is a weaker property. In this section, we discuss how one can make a proof of a property simpler if this proof is only needed in the context of proving a stronger property.

A. Definitions

We will denote the predicates of the *transition relation* and the *initial states* as $T(S,S')$ and $I(S)$ respectively. Here S and S' are sets of present and next state variables respectively. An assignment s to variables S is called a *state*. We will refer to a state satisfying a predicate $Q(S)$ as a *Q-state*. A property is just a predicate $P(S)$. We will say that a P-state (respectively $\overline{P}$-state) is a *good* (respectively *bad*) state. A property P is called *inductive* with respect to T if $P(S) \wedge T \rightarrow P(S')$ holds.

We will call a sequence of states $(s_j,\ldots,s_k)$ a *trace* if $T(s_i,s_{i+1})$ is true for $i = j,\ldots,k-1$. We will call the trace above *initialized* if s_j is an I-state. Given a property $P(S)$ where $I \rightarrow P$, a CEX is an *initialized trace* $(s_0,\ldots,s_k)$ where s_i, $i = 0,\ldots,k-1$ are P-states and s_k is a $\overline{P}$-state. We will refer to a state transition system with initial states I and transition relation T as an *(I,T)-system.* Given an (I,T)-system, checking a property P is to find a CEX for P or to show that none exists.

B. Hardness of proving strong and weak properties

Let P and Q be two properties where Q is weaker than P, i.e., $P \rightarrow Q$. On the one hand, verification of Q should be easier because one needs to prove unreachability of a smaller set of bad states. On the other hand, P can be inductive even if Q is not. In fact, the essence of IC3 is to turn a non-inductive property into an inductive one by adding strengthening clauses. This makes the modified property easier to prove despite the fact that it is stronger from a logical point of view.

The reason for the paradox above is as follows. The set of traces one needs to consider to prove Q is not a subset of those one considers when proving P. To prove P, one needs to show that there is no initialized trace of P-states leading to a $\overline{P}$-state. Thus, one does not consider traces where two $\overline{P}$-states occur. Proving Q is reduced to showing that there is no initialized trace of Q-states leading to a $\overline{Q}$-state. Since $P \rightarrow Q$, a $\overline{Q}$-state is a $\overline{P}$-state as well. On the other hand, a Q-state can also be a $\overline{P}$-state. Thus, in contrast to the case when we prove P, to prove Q one has to consider traces that may include two or more different $\overline{P}$-states.

We will refer to a regular proof of Q (where one shows that no initialized trace of Q-states leads to a $\overline{Q}$-state) as a *global* one. In the next subsection, we discuss reducing the complexity of proving Q using the machinery of local proofs.

C. Local proofs

The intuition behind local proofs is as follows. Suppose that one needs to prove a property Q as a step in proving a stronger property P. Then it is reasonable to ignore traces that do not make sense from the viewpoint of proving P. Proving *Q locally in the context of P*, or just *locally* for short, is to show that there does not exist an initialized trace of P-states (rather than Q-states) leading to a $\overline{Q}$-state.

The importance of local proofs is twofold. First, to prove Q locally, one needs to consider only a subset of the traces to prove P (because the set of $\overline{Q}$-states is a subset of that of $\overline{P}$-states). Thus, in terms of the set of traces to consider, a weaker property becomes also "easier". Second, as we show in Section 4, to prove the aggregate property $P := P_1 \wedge \ldots \wedge P_k$, it suffices to prove all properties P_i *locally* with respect to P.

It is convenient to formulate the notion of a local proof in terms of a modified transition relation T. We will call this modification *the projection of T onto property P* and denote it as T^P. It is defined as follows.

- $T^P(s,s') = T(s,s')$, if s is a P-state.
- $T^P(s,s') = 0$ if s is a $\overline{P}$-state and $s \neq s'$.
- $T^P(s,s') = 1$ if s is a $\overline{P}$-state and $s = s'$.

Informally, T^P is obtained from T by excluding any transitions from a $\overline{P}$-state other than a transition to itself. Hence, a trace in (I,T^P)-system cannot have two different $\overline{P}$-states. Thus, a local proof of Q with respect to property P, as we introduced above, is just a regular proof with respect to T^P. (In turn, proving Q globally is done with respect to T.)

Proposition 1: Let P be inductive with respect to transition relation T. Then any property Q weaker than P (i.e., $P \rightarrow Q$) is inductive with respect to T^P.

(The proofs of the propositions are given in [5].) Proposition 1 states that in terms of proofs by induction, proving Q locally with respect to a stronger property P is *at most* as hard as proving P itself.

3. Local Proofs And Debugging

In this section, we explain how the machinery of local proofs can be used to address the following problem. Given a property P that failed, find a weaker property Q that is false as well and can be viewed as an *explanation* for failure of P. The subtlety here is that not every failed property Q where $P \rightarrow Q$ can be viewed as a reason for why P fails. We will refer to this problem as the *debugging* problem.

To address the debugging problem one first needs to clarify the relation between local and global proofs.

Proposition 2: Let $P \rightarrow Q$. If property Q holds with respect to T (i.e., globally), it also holds with respect to T^P (i.e., locally). The opposite is not true. If Q holds with respect to T^P, it either holds with respect to T or it fails with respect to T and every CEX contains at least two $\overline{P}$-states s' and s'' where $s' \neq s''$.

Informally, Proposition 2 means that proving Q locally is almost "as good as" proving globally modulo CEXs that do not make sense from the viewpoint of proving P. These CEXs have at least two $\overline{P}$-states.

One can use Proposition 2 for solving the debugging problem as follows. Suppose that Q does not hold locally. This means that there is a CEX of P-states leading to a $\overline{Q}$-state. Since $P \rightarrow Q$, a $\overline{Q}$-state is a $\overline{P}$-state as well. So this CEX is also a regular CEX for P. In other words, the fact that Q fails locally means that Q can be viewed as a reason for failure of P.

Suppose that Q holds locally. Assume that Q fails globally and $(s_0,\dots,s_m)$ is a CEX where s_m is a $\overline{Q}$-state. From Proposition 2, it follows that this CEX has at least two $\overline{P}$-states. One of these states is s_m (because $P \rightarrow Q$). Another $\overline{P}$-state is one of Q-states s_i, $i = 1,\dots,m-1$. This means that failure of Q is not a reason for failure of P. Indeed, in every CEX for Q, property P fails *before* Q does.

Summarizing, if property Q fails (respectively holds) locally with respect to P, failure of Q is a reason (respectively cannot be a reason) for failure of P.

4. JA-VERIFICATION

In this section, we present a version of separate verification called "Just-Assume" or JA-verification. As before, P denotes the aggregate property $P_1 \wedge \dots \wedge P_k$ and T^P denotes the projection of T onto P (see Subsection 2-C). Since every property P_i is a weaker version of P, one can use the results of Sections 2 and 3 based on the machinery of local proofs.

We now provide a justification of proving weaker properties locally in the context of multi-property verification. By using the transition relation T^P to prove P_i, one essentially assumes that every property P_j, $j \neq i$ holds. While this may not be the case, nevertheless it works for two reasons. The first reason is that if the aggregate property P fails, there is a time frame where P (and hence some property P_i) fails for the first time. Let this be time frame number m. For every time frame number p where $p < m$, the assumption that every property P_j, $j \neq i$ holds *is true*. Thus, if P fails, there is at least one property (in our case P_i) that fails even with respect to T^P.

Here is the second reason why assuming P_j, $j \neq i$ works. To get some debugging information when proving property P_i, one is interested in traces where P_i fails *before* any other property does. By assuming P_j, $j \neq i$ is true, one drops the traces where P_i fails *after* some P_j, $j \neq i$ has failed.

The propositions below formalize the relation between local proofs and multi-property verification.

Proposition 3: Property P holds with respect to T iff every P_i, $i = 1,\dots,k$ holds with respect to T.

Proposition 4: Property P holds with respect to T iff P holds with respect to T^P.

Taking into account that Proposition 3 can also be formulated in terms of T^P (instead of T), Propositions 3 and 4 entail the proposition below.

Proposition 5: Property P holds with respect to T iff every P_i, $i = 1,\dots,k$ holds with respect to T^P.

We will refer to the subset of $\{P_1,\dots,P_k\}$ that consists of properties that fail with respect to T^P, i.e., locally as a *debugging set*. The following proposition justifies this name.

Proposition 6: Let the aggregate property P fail. Let D denote the debugging set of properties. Then the failure of properties of D is the reason for the failure of P in the following sense. For each CEX $(s_0,\dots,s_m)$ for property P, the state s_m that falsifies P also falsifies at least one property $P_i \in D$.

Example 1: The Verilog code below gives an example of an 8-bit counter. This counter increments its value every time the *enable* signal is true. Once the counter reaches the value of *rval* it resets its value to 0. We also want to reset the counter when signal *req* is true (*regardless* of the current value of the counter). However, the code contains a buggy line (marked in blue), which prohibits a reset *only* unless *req* is true.

```
module counter(enable,clk,req);
   parameter  rval = 1 << 7;
   input      enable, clk, req;
   reg [7:0]  val;
   wire       reset;

   initial val = 0;

   assign reset = ((val == rval) && req);

   always @(posedge clk) begin
      if (enable) begin
         if (reset) val = 0;
         else val = val+1;
      end
   end

   P0: assert property (req == 1);
   P1: assert property (val <= rval);

endmodule
```

Let us consider verification of properties P_0 and P_1 specified by the last two lines of the module `counter`. Property P_0 fails globally in every time frame because *req* is an input variable taking values 0 and 1. Property P_1 fails globally due to the bug above. Note however, that only property P_0 fails *locally*. Indeed, P_0 fails even under assumption $P_1 \equiv 1$. However, P_1 becomes true if one assumes $P_0 \equiv 1$. The latter means that $req \equiv 1$ and so the counter always resets on reaching value *rval*. So the debugging set consists only of P_0. The fact that P_1 holds locally means that either P_1 is true globally or that any CEX failing P_1 first fails P_0. The latter implies that the failure of P_1 is caused by incorrect handling of variable *req*.

Note that proving P_1 false globally is hard for a large counter because a CEX consists of all states of the counter from 0 to *rval*. On the contrary, proving P_1 true under assumption $P_0 \equiv 1$ is trivial because P_1 is inductive under this assumption. In Table I, we compare proving properties P_0 and P_1 above globally and locally. The first column gives the size of the counter. The next four columns give the results of solving

TABLE I
Example with a counter. Time limit is 1 hour

#bits	solving globally				solving locally
	ABC (bmc)		ABC (pdr)		
	#time frames	time	#time frames	time	
8	128	0.3 s	10	0.1 s	0.01 s
12	2,048	723 s	51	1.7 s	0.02 s
14	$*$	$*$	118	9.9 s	0.02 s
16	$*$	$*$	269	113 s	0.02 s
18	$*$	$*$	315	1,278 s	0.02 s
20	$*$	$*$	$*$	$*$	0.02 s

P_0 and P_1 globally by ABC, a mature tool developed at UC Berkeley [6]. The first pair of columns gives the results of Bounded Model Checking [1] (the largest number of used time frames and run time). The next pair of columns provides results of PDR (i.e., IC3). Finally, we give the results of solving P_0 and P_1 locally by our tool (see Section 7).

The results show that bounded model checking soon becomes impractical, as the number of time frames increases exponentially. ABC's PDR solves more cases, but to generate a CEX, it has to consider a quickly increasing number of time frames as well. For JA-verification, the size of the counter has no influence on the run time. While the counter is a purely synthetic example, in practice, one often has to find so-called *deep* counterexamples. A system with complex inner state might require a long sequence of steps to reach a buggy state.

5. Handling Properties Expected To Fail

When proving properties P_i, $i = 1,\dots,k$ in JA-verification, as introduced in Section 4, one excludes the traces where a property P_j, $j \neq i$ fails before P_i does. This is based on the assumption that the properties that are the first to fail indicate design behaviors that need to be fixed first. However, this assumption is unreasonable when a property P_j that fails before P_i is *Expected To Fail* (ETF). For instance, to ensure that a state s is reachable, one may formulate an ETF property P_j where s is a $\overline{P}_j$-state. In this case, excluding the traces where P_j fails before P_i is a mistake.

One can easily extend JA-verification to handle ETF properties as follows. Suppose that our objective is to prove every property P_i that is *Expected To Hold* (ETH). In addition, for every ETF property we want to find a CEX that does not break any ETH property. Then, to solve P_i, $i = 1,\dots,k$ *locally* one assumes that every ETH P_j, $j \neq i$ is true. Thus, we exclude the traces where ETH properties fail before P_i, even if the latter is an ETF property.

6. IC3 And Clause Re-using

So far, we discussed the machinery of local proofs without specifying the algorithm used to prove a property. In this section, we describe an optimization technique applicable if property checking is performed by IC3 [3]. The essence of this technique is to re-use strengthening clauses generated by IC3 for property P_i to strengthen another property P_j, $j \neq i$.

A. Brief description of IC3

Let Q be a property of an (I,T)-system where $I \rightarrow Q$. If Q holds, there always exists a predicate $F(S)$ such that $Q \wedge F$ is inductive with respect to T. Then $(Q \wedge F \wedge T) \rightarrow (Q' \wedge F')$, where a primed predicate symbol means that the predicate in question depends on next state variables S' (for instance, Q' denotes $Q(S')$). The fact that $Q \wedge F$ is inductive implies that $Q \wedge F$ is an over-approximation of the set of states reachable in the (I,T)- system in question. Therefore, for every state s reachable in (I,T)-system, $Q(s) \wedge F(s) = 1$.

In IC3, formulas are represented in conjunctive normal form and the predicate F is constructed as a set of *clauses* (disjunctions of literals). Let $Rch(I,T,j)$ denote the set of states reachable from I-states in at most j transitions. To construct the formula F, IC3 builds a sequence of formulas $F_0,\dots,F_m$ where $F_0 = I$ and $F_j, j = 1,\dots,m$ specifies an over-approximation of $Rch(I,T,j)$. That is, if a state s is in $Rch(I,T,j)$, then s satisfies F_j, i.e., $F_j(s) = 1$. A formula F_j, $j > 0$ is *initialized* with Q. Then F_j (and possibly some formulas F_i, $i < j$) are strengthened by adding so called *inductive clauses*. If $Q \wedge F_i$ becomes inductive for some value of $i, i \leq j$, property Q holds. Clause C is called *inductive relative to* F_j if $I \rightarrow C$ and $C \wedge F_j \wedge T \rightarrow C'$ hold. In this case, every $s \in Rch(I,T,j)$ satisfies C and so $F_j \wedge C$ is still an over-approximation of $Rch(I,T,j)$.

B. Re-using strengthening clauses

The idea of re-using strengthening clauses is based on the following observation. Suppose that F_Q is a set of clauses which makes Q inductive in the (I,T)-system. This means that $Q \wedge F_Q$ is an over-approximation of the set of all states reachable in the (I,T)-system. Hence a state $s \in Rch(I,T,j)$ satisfies F_Q, for any value $j \geq 0$. Suppose one needs to prove some other property R. Then, when constructing a formula over-approximating $Rch(I,T,j)$, one can initialize this formula with $R \wedge F_Q$, rather than with R.

7. Implementation

We use a version of IC3 developed in our research group. We will refer to it as *Ic3-db* where "db" stands for Diffblue. *Ic3-db* uses the front-end of EBMC [7]. We will refer to our implementation of JA-verification based on *Ic3-db* as *Ja-ver*. The latter is a Perl script that calls *Ic3-db* in a loop for proving individual properties.

Let $P_1,\dots,P_k$ be the set of properties to be proved. Proving P_i locally means showing that there is no initialized trace of P-states that leads to a $\overline{P_i}$-state, where P is the aggregate property $P_1 \wedge \dots \wedge P_k$. To guarantee that all present states satisfy P, *Ic3-db* adds constraints to the transition relation T that force P_j, $j \neq i$ to be equal to 1. Adding constraints, in general, adversely affects an important optimization technique of IC3 called *state lifting* [4], [8]. For that reason, *Ic3-db* has an option to make the state lifting procedure ignore constraints specified by P_j, $j \neq i$. The interaction of state lifting and property constraints in *Ic3-db* is described in more detail in a technical report [5]. This report also provides more information on how re-use of strengthening clauses is implemented in *Ic3-db*.

8. Experimental Results

In this section, we report experiments that show the viability of our approach to multi-property verification. We use

TABLE II
Designs with failed properties. Many properties of 6s258, 6s207, 6s254, 6s335, and 6s380 are false globally in joint verification but true locally in JA*-verification. 'mem' means running out of memory*

name	#latch	#pro pert.	Joint verification				JA-verification by *Ic3-db* with clause re-use		
			ABC		*Ic3-db*				
			#false (#true)	time	#false (#true)	time	time limit	#false (#true)	total time
6s104	84,925	124	1 (0)	10 h	1 (0)	mem	0.3 h	1 (123)	2.5 h
6s260	2,179	35	1 (0)	10 h	1 (0)	10 h	0.5 h	1 (34)	1,686 s
6s258	1,790	80	25 (0)	10 h	30 (0)	10 h	0.3 h	1 (72)	2.4 h
6s175	7,415	3	2 (0)	10 h	2 (0)	10 h	0.3 h	2 (1)	554 s
6s207	3,012	33	6 (0)	10 h	10 (0)	10 h	0.3 h	2 (31)	22 s
6s254	762	14	13 (1)	25 s	13 (1)	225 s	0.3 h	1 (13)	2 s
6s335	1,658	61	26 (35)	2 h	26 (35)	260 s	0.3 h	20 (41)	56 s
6s380	5,606	897	399 (0)	10 h	395 (0)	10 h	0.3 h	3 (894)	550 s

benchmarks from the multi-property track of the HWMCC-12 and 13 competitions We did two experiments, comparing joint and JA-verification based on *Ic3-db*. For both experiments, we picked eight designs to illustrate the point described in the corresponding subsection. (More experimental data can be found in [5].) We cross-check the results of *Ic3-db* in joint verification with those reported by the latest version of ABC [6].

As we mentioned in Section 7, JA-verification is implemented as a Perl script *Ja-ver* that calls *Ic3-db* to process individual properties sequentially. In this paper, we do not exploit the possibility to improve JA-verification by processing properties in a particular order.[1] Properties are verified in the order they are given in the design description.

Joint verification is also implemented as a Perl script called *Jnt-ver*, where *Ic3-db* is called to verify the aggregate property $P := P_1 \wedge \ldots \wedge P_k$. If P fails, the individual properties refuted by the generated CEX are reported false. *Jnt-ver* forms a new aggregate property by conjoining the properties P_i that are unsolved yet and calls *Ic3-db* again. This continues until every property is solved. As for ABC, joint verification is its natural mode of operation. However, in contrast to *Jnt-ver*, ABC does not re-start when a property is proved false and goes on with solving the remaining properties.

In both experiments, the time limit for joint verification was set to 10 hours. The time limit used by *Ic3-db* in JA-verification to prove one property is indicated in Tables II, III and IV. If a property of a benchmark was not solved by *Ic3-db*, the time limit was added to the total time of solving this benchmark.

A. Designs with failing properties

In the first experiment, we picked eight designs with failing properties to show that solving properties locally can be much more efficient than globally. Thus, the sacrifice one makes by looking only for the failed properties that form the debugging set pays off. Unfortunately, the HWMCC competitions do not identify properties of multi-property benchmarks that are expected to fail, if any (see Section 5). For that reason, we assumed that every property was expected to hold.

The results are given in Table II. The first column provides the name of the benchmark. The second and third columns give the number of latches and properties, respectively. The next two pairs of columns provide the results of joint verification performed by ABC and *Ic3-db*. The first column of the pair gives the number of false and true properties that ABC or *Ic3-db* managed to solve within the time limit. The second column of the pair reports the amount of time taken by ABC or *Ic3-db*. The last three columns report data about JA-verification: the time limit per property, the number of false and true properties solved within the time limit, and the total time taken by *Ic3-db*. In all tables, the run times that do not exceed one hour are given in seconds.

For all examples but *6s258*, JA-verification solved all properties *locally*. On the other hand, for many examples, in joint verification, only a small fraction of properties were solved by *Ic3-db* and ABC *globally*. Let us consider example *6s207* in more detail. JA-verification solved all properties of *6s207* fairly quickly generating the debugging set of two properties. On the other hand, joint verification by *Ic3-db* proved that ten properties failed globally within 10 hours. Since JA-verification showed that only two properties failed locally, eight out of those ten failed properties were true locally. Let P_i be one of those eight properties. The CEX found for P_i by joint verification first falsifies a property of the debugging set. Thus, we do not know if *there is* a CEX where P_i fails before other properties. JA-verification does not determine whether P_i fails but guarantees that *every* CEX for P_i (if any) first fails some other property.

B. Designs where all properties hold

In the second experiment, we used designs where all properties were true. Recall that if all properties hold locally, they all hold globally as well. Ideally, we would like to use designs with as many properties as possible. However, for such designs, joint verification is usually outperformed by JA-verification since the presence of even a few hard properties P_i cripples its performance. This problem can be addressed by partitioning $P_1, \ldots, P_k$ into smaller clusters of properties [9], which is beyond the scope of this paper. Thus, we picked eight designs that have less than a thousand properties and that can be solved by joint verification without partitioning into clusters. The point we are making is that separate verification is competitive with joint verification even for the benchmarks that favor the latter.

The results are given in Table III. The first three columns are the same as in Table II. The next two columns give run

TABLE III
All properties are true

name	#latch	#pro-pert.	Joint verification		JA-verification by *Ic3-db* with clause re-use		
			ABC	*Ic3-db*	time limit	#un-solved	*Ic3-db*
6s124	6,748	630	>10 h	2.9 h	0.8 h	0	**1.9 h**
6s135	2,307	340	123 s	**335 s**	0.8 h	0	746 s
6s139	16,230	120	4.7 h	**1.7 h**	2.8 h	2	6.5 h
6s256	3,141	5	>10 h	**602 s**	2.8 h	1	2.9 h
bob12m09	285	85	1,692 s	930 s	0.8 h	0	**784 s**
6s407	11,379	371	1.3 h	3.4 h	0.8 h	0	**2,077 s**
6s273	15,544	42	1.8 s	325 s	0.8 h	0	**290 s**
6s275	3,196	673	334 s	**1,154 s**	0.8 h	0	1,611 s

[1]A rule of thumb here is to verify easier properties first to accumulate strengthening clauses and use them later for harder properties.

TABLE IV
Re-using strengthening clauses in JA*-verification. Time limit per property is the same as in Table III. Verification of 6s124, 6s139, and 6s407 (without clause re-use) was aborted after 10 hours*

name	#properties	without clause re-use		with clause re-use	
		#unsolved	time	#unsolved	time
6s124	630	505	10 h	0	**1.9 h**
6s135	340	0	2.7 h	0	**746 s**
6s139	120	116	10 h	2	**6.5 h**
6s256	5	0	**892 s**	1	2.9 h
bob12m09	85	0	1.1 h	0	**784 s**
6s407	317	270	10 h	0	**2,077 s**
6s273	42	0	1,445 s	0	**290 s**
6s275	673	0	3,273 s	0	**1,611 s**

times of ABC and *Ic3-db* in joint verification. The last three columns provide information about JA-verification: time limit per property, number of unsolved properties and total run time. The best of the run times obtained in joint verification and JA-verification based on *Ic3-db* is given in bold. In three cases, joint verification based on ABC was the fastest but we needed a comparison that uses a uniform setup.

Table III shows that joint verification performed slightly better. In particular, for benchmarks *6s139* and *6s256*, JA-verification failed to solve some properties with the time limit of 2.8 hours. However, when we verified properties in an order different from the one of design description, both benchmarks were solved in time comparable with joint verification.

To illustrate the benefit of re-using strengthening clauses, Table IV compares JA-verification with and without re-using strengthening clauses on the examples of Table III. Table IV shows that JA-verification with re-using strengthening clauses significantly outperforms its counterpart. The only exception is *6s256* which has only five properties to check.

9. Related Work

We found only a few references to research on multi-property verification. In [10], some modifications of ABC are presented that let it handle multi-property designs. In [9], [11], the idea of grouping similar properties and solving them together is introduced. The similarity of properties is decided based on design structure (e.g., properties with similar cones of influence are considered similar). The main difference of this approach from ours is that the latter is purely *semantic*. Thus, the optimizations of separate verification we consider (local proofs and re-using strengthening clauses) can be incorporated in any structure-aware approach. One further difference is that the idea of grouping favors *correct* designs. Grouping may not work well for designs with broken properties that fail for different reasons and thus have vastly different CEXs.

Assume-guarantee reasoning is an important method for compositional verification [12], [13]. It reduces verification of the whole system to checking properties of its components under some assumptions. To guarantee the correctness of verification one needs to prove these assumptions true. As we mentioned earlier, JA-verification uses yet-unproven properties as assumptions without subsequent justification. This is achieved by our particular formulation of multi-property verification. Instead of proving or refuting every property, JA-verification builds a subset of failed properties that are the first to break or proves that this subset is empty.

10. Conclusions

We consider the problem of verifying multiple properties $P_1, \dots, P_k$ of the same design. We make a case for separate verification where properties are proved one by one as opposed to joint verification where the aggregate property $P_1 \land \ldots \land P_k$ is used. Our approach is purely semantic, i.e., we do not rely on any structural features a design may or may not have.

We introduce a novel variant of separate verification called JA-verification. JA-verification checks if P_i holds locally, i.e., under the assumption that all other properties are true. We show that if all properties hold locally, they also hold globally, i.e., without any assumptions. Instead of finding the set of all failed properties, JA-verification identifies a "debugging" subset. The properties in the debugging subset highlight design behaviors that need to be fixed first, which can yield substantial time savings in the design-verification cycle.

We experimentally compare conventional joint verification and JA-verification. We give examples of designs with failed properties where JA-verification dramatically outperforms its counterpart, especially for designs where a small debugging set D exists. For these designs, one needs to find only $|D|$ CEXs which are typically shallow. Computation of deeper CEXs for false properties that are not in D is replaced with proving them true locally. Re-using inductive invariants generated for individual properties that are locally true significantly speeds up JA-verification. In particular, for correct designs, it makes JA-verification competitive with joint verification even for benchmarks that favor the latter.

References

[1] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *DAC*, 1999, pp. 317–320.
[2] K. L. Mcmillan, "Interpolation and SAT-based model checking," in *CAV*. Springer, 2003, pp. 1–13.
[3] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.
[4] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011.
[5] E. Goldberg, M. Gudemann, D. Kroening, and R. Mukherjee, "Efficient verification of multi-property designs (the benefit of wrong assumptions) (extended version)," Tech. Rep. arXiv:1711.05698 [cs.LO], 2017.
[6] B. L. Synthesis and V. Group, "ABC: A system for sequential synthesis and verification," 2017, http://www.eecs.berkeley.edu/∼alanmi/abc.
[7] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 7–12.
[8] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *FMCAD*, 2011.
[9] G. Cabodi and S. Nocco, "Optimized model checking of multiple properties," in *DATE*, 2011.
[10] Http://people.eecs.berkeley.edu/∼alanmi/presentations/updating_engines00.ppt.
[11] P. Camurati, D. P. C. Loiacono, P. Pasini, and S. Quer, "To split or to group: from divide-and-conquer to sub-task sharing in verifying multiple properties," in *DIFTS*, 2014.
[12] C. Jones, "Specification and design of (parallel) programs," in *IFIP 9th World Congress*, 1983, pp. 321–332.
[13] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logic and Models of Conc. Sys.*, vol. 13, 1984, pp. 123–144.