

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

Eugene Goldberg, Matthias Gdemann, 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, \dots, 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, \dots, 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, \dots, 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 $P := P_1 \wedge \dots \wedge P_k$ 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

* This is an extended version of the paper published at DATE-2018.

Supported by ERC project 280053 “CPROVER”, the H2020 FET OPEN 712689 SC² and SRC contracts no. 2012-TJ-2269 and 2016-CT-2707.

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 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 quite expensive (e.g., if a counter is involved). Third, we provide implementation details (Section 7) and experimental results showing the viability of JA-verification (Sections 9, 10, 11). In particular, in Section 11, we give evidence that JA-verification facilitates parallel computing.

2. LOCAL AND GLOBAL PROOFS

Separate verification is based on the assumption that, in general, proving P_i is easier than $P_1 \wedge \dots \wedge P_k$ 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 \bar{P} -state) is a *good* (respectively *bad*) state. A property P is called *inductive* with respect to T if $P \wedge T \rightarrow P'$ holds where a primed predicate symbol means that the predicate in question depends on next state variables S' .

We will call a sequence of states (s_j, \dots, s_k) a *trace* if $T(s_i, s_{i+1})$ is true for $i = j, \dots, k-1$. 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, \dots, s_k) where s_i , $i = 0, \dots, k-1$ are P -states and s_k is a \bar{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 \bar{P} -state. Thus, one does not consider traces where two \bar{P} -states occur. Proving Q is reduced to showing that there is no initialized trace of Q -states leading to a \bar{Q} -state. Since $P \rightarrow Q$, a \bar{Q} -state is a \bar{P} -state as well. On the other hand, a Q -state can also be a \bar{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 \bar{P} -states.

We will refer to a regular proof of Q (where one shows that no initialized trace of Q -states leads to a \bar{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 \bar{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 \bar{Q} -states is a subset of that of \bar{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 $P := P_1 \wedge \dots \wedge P_k$, 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 \bar{P} -state and $s \neq s'$.
- $T^P(s, s') = 1$ if s is a \bar{P} -state and $s = s'$.

Informally, T^P is obtained from T by excluding any transitions from a \bar{P} -state other than a transition to itself. Hence, a trace in (I, T^P) -system cannot have two different \bar{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 .

Proof: Assume the contrary. That is Q is not inductive with respect to T^P and hence $Q \wedge T^P \rightarrow Q'$ does not hold. Then there is a transition (s, s') such that

- $T^P(s, s') = 1$ and
- $Q(s) = 1$ and $Q(s') = 0$ (and hence $s \neq s'$)

Since $P \rightarrow Q$, then $P(s') = 0$ as well. Since $s \neq s'$ and $T^P(s, s') = 1$, from definition of T^P it follows that $P(s) = 1$ and $T(s, s') = 1$. So (s, s') is a transition from a P -state to a \bar{P} -state allowed by T . Then P is not inductive with respect to T . We have a contradiction. QED

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 $P \rightarrow Q$.

- A) If property Q holds with respect to transition relation T (i.e., globally), it also holds with respect to T^P (i.e., locally).

The opposite is not true.

- B) 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 \bar{P} -states s_i and s_j where $s_i \neq s_j$.

Proof: A) Assume that Q does not hold with respect to T^P . Then there is an initialized trace of P -states leading to a \bar{Q} -state. Since T^P inherits all transitions of T from P -states, this trace is valid with respect to T . Since $P \rightarrow Q$, every P -state is also a Q -state. So there is an initialized trace of Q -states leading to a \bar{Q} -state that is valid with respect to T . Hence Q does not hold with respect to T and we have a contradiction.

B) Assume the contrary, i.e. Q holds with respect to T^P and there is a CEX with respect to T containing only one \bar{P} -state. (So neither Q holds globally nor every CEX contains at least two \bar{P} -states). The CEX above is also a CEX with respect to T^P and hence Q fails locally. So we have a contradiction. QED

Informally, Proposition 2 means that proving Q locally is “as good as” proving globally modulo CEXs that do not make sense from the viewpoint of proving P . These CEXs have at least two \bar{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 \bar{Q} -state. Since $P \rightarrow Q$, a \bar{Q} -state is a \bar{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 \bar{Q} -state. From Proposition 2, it follows that this CEX has at least two \bar{P} -states. One of these states is s_m (because $P \rightarrow Q$). Another \bar{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_j , $j = 1, \dots, k$ holds with respect to T .

Proof: *If part.* Let P_j , $j = 1, \dots, k$ hold with respect to T . Assume the contrary i.e. P does not hold. Then there is a CEX (s_0, \dots, s_m) where s_i , $i = 0, \dots, m-1$ are P -states and s_m is a \bar{P} -state. This means that s_m falsifies some property P_j . Since a P -state is also a P_j -state, the CEX above is an initialized trace of P_j -states leading to a \bar{P}_j state. Hence P_j does not hold and we have contradiction.

Only if part. Let P hold with respect to T . Assume the contrary i.e. a property P_j does not hold. Then there is a CEX (s_0, \dots, s_m) where s_i , $i = 0, \dots, m-1$ are P_j -states and s_m is a \bar{P}_j -state. Since $P \rightarrow P_j$ holds, s_m is a \bar{P} -state as well. As far as states s_i , $i = 0, \dots, m-1$ are concerned one can have the following two situations.

- All these states are P -states as well. Then the CEX above breaks P and we have a contradiction.
- At least one state s_i , $0 \leq i \leq m-1$ is a \bar{P} -state. In this case, trace (s_0, \dots, s_i) is a CEX breaking P and we have a contradiction. QED

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

Proof: Any trace of P -states leading to a \bar{P} -state is valid both with respect to T and T^P . So if P fails with respect to T it also does with respect to T^P and vice versa. QED

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

Proof: From Proposition 4 it follows that P holds with respect to T iff it holds with respect to T^P . After replacing T with T^P in Proposition 3, one concludes that P holds with respect to T^P iff every P_i , $i = 1, \dots, k$ holds with respect to T^P . QED

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 $P_i \in D$.

Proof: Assume the contrary, i.e., there is a CEX for P such that s_m falsify only properties of $\{P_1, \dots, P_k\}$ that are not in D . Let P_i be a property falsified by s_m . Since this CEX consists of P -states leading to a \bar{P} -state, then P_i fails locally and so it is in D . Thus, we have a contradiction. QED

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

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

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 P_0 and P_1 globally by ABC, a mature tool developed at UC Berkeley [5]. 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 \bar{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 , $j \neq i$. Before describing clause re-using, we give a high-level view of IC3.

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 $G(S)$ such that $Q \wedge G$ is inductive with respect to T . Then $(Q \wedge G \wedge T) \rightarrow (Q' \wedge G')$. (Recall that a primed predicate symbol means that the predicate in question depends on next state variables S' .) The fact that $Q \wedge G$ is inductive implies that $Q \wedge G$ 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, $Q(s) \wedge G(s) = 1$.

Let F denote $Q \wedge G$ i.e property Q strengthened by G . 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 is strengthened by adding so called *inductive clauses*. The objective of this strengthening is to exclude the F_j -states from which a bad state is reachable in one transition. The exclusion of an F_j -state may require excluding F_i -states, $i < j$ (by adding inductive clauses to F_i) from which a bad state can be reached in $j - i + 1$ transitions. 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 G_Q is a set of clauses which makes Q inductive in the (I, T) -system. This means that $Q \wedge G_Q$ 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 G_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 $R \wedge G_Q$, rather than with R .

C. State lifting in IC3

In this subsection, we briefly describe an important technique of IC3 called *state lifting* [4], [6]. This description is used in Section 7 when explaining how local proofs are implemented. Let s be a state from which a bad state is reachable. IC3 tries to exclude s by generating an inductive clause falsified by s . If such a clause cannot be built immediately, IC3 tries to exclude every state q of the previous time frame from which there is a transition to state s . The number of those states q can be very large. So, IC3 tries to “lift” q to a cube C_q that not only contains the state q itself, but many other states that are one transition away from state s . Subsequently, IC3 tries to exclude all states of C_q in one shot. Informally, the larger C_q , the greater the performance boost by lifting.

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.

A. Using properties as constraints

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 \bar{P} -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 to T affects the lifting procedure of IC3 (see Subsection 6-C). The reason is that one needs to guarantee that all states of the cube C_q obtained by lifting a state q satisfy the constraints in question. In our case, all states of C_q must be P -states. Unfortunately, this can drastically decrease the size of C_q and therefore reduce the effectiveness of lifting. For that reason, *Ic3-db* has an option to make the lifting procedure ignore the constraints forcing $P_j, j \neq i$ to be equal 1.

The relaxation of lifting above can lead to appearance of “spurious” CEXs that contain transitions from \bar{P} -states to other states. (This can occur only if P does not hold). If a spurious CEX is generated when proving property P_i , then *Ic3-db* is invoked again. This time the lifting procedure is forced to respect the constraints specified by $P_j, j \neq i$.

B. Implementation of clause re-using

The correctness of re-using strengthening clauses is discussed in Subsection 6-B. Assume that properties P_1, \dots, P_k are processed in the order they are numbered. Let G_{P_i} denote the strengthening of property P_i , i.e., $P_i \wedge G_{P_i}$ is inductive with respect to T . *Ja-ver* maintains an external file *clauseDB* that collects strengthening clauses. Therefore, after making P_1 inductive, the clauses of G_{P_1} are written to *clauseDB*.

When *Ic3-db* is invoked to prove P_2 , the clauses of G_{P_1} are extracted from *clauseDB*. When proving P_2 , the formula over-approximating $Rch(I, T, j)$ is initialized with $P_2 \wedge G_{P_1}$ (rather

than with P_2). Let G_{P_2} denote the strengthening clauses added to $P_2 \wedge G_{P_1}$ to make the latter inductive with respect to T . These clauses are written to *clauseDB* (that already contains G_{P_1}). In general, when *Ic3-db* is invoked to prove P_j , all clauses $\bigcup_{i < j} G_{P_i}$ are extracted from *clauseDB* to be used in the proof.

8. COMPARING LOCAL AND GLOBAL PROOFS

In Sections 9, 10 and 11, we experimentally compare methods based on local and global proofs. In this section, we summarize the information we provided earlier to help better understand these experimental results. We will refer to methods proving properties locally (respectively globally) as a local (respectively global) approach.

Let \mathbb{P} denote the set $\{P_1, \dots, P_k\}$ of properties to verify. There are two cases where local and global approaches provide the same information. Assume one managed to prove every property of \mathbb{P} locally. In this case, every property of \mathbb{P} holds globally as well. Thus global and local approaches provide the same information here. Now assume that one proved $P_i \in \mathbb{P}$ false locally (and hence globally). Assume also that in a global approach one proved P_i false and in the generated CEX property P_i is *the first* to fail. Then, P_i is “inadvertently” proved false not only globally but locally too. In this case, both local and global approaches provide the same information for P_i .

Now consider the cases where local and global approaches provide different information for a failed property. Assume that one proved P_i false globally and the generated CEX falsifies at least one property $P_j \in \mathbb{P}$, $i \neq j$ before P_i . Consider the following two cases. The first case is that one proves P_i false locally. Then the local approach provides more information than its global counterpart because in addition to proving P_i false it shows that P_i is in the debugging set. The second case is that one proves P_i true locally. In this situation, local and global approaches do not have a clear winner. On the one hand, the global approach gives more information by finding a CEX for P_i . On the other hand, the local approach provides more information by showing that *every* CEX falsifying P_i (if any) first falsifies some other property of \mathbb{P} . Note that from the viewpoint of debugging, the information provided by the local approach in the second case is more useful since P_i is shown not to be in the debugging set.

Finally, let us consider the following situation. Let $\mathbb{P}' \subset \mathbb{P}$ be the subset of properties proved true locally and $\mathbb{P}'' = \mathbb{P} \setminus \mathbb{P}'$. A property is in \mathbb{P}'' if it is proved false locally or it is too hard to solve or it has not been tried yet. Since $\mathbb{P}'' \neq \emptyset$, the fact that $P_i \in \mathbb{P}'$ holds locally does not mean that it holds globally as well. Thus, proving P_i true globally provides more information than in the local approach. However, from the debugging point of view, proving P_i true locally is almost as good as globally. This proof means that P_i cannot fail “on its own”. So one first needs to focus on properties of \mathbb{P}'' , in particular, to fix the design behaviors causing property failures.

9. JA-VERIFICATION VERSUS JOINT VERIFICATION

In this section, we experimentally compare JA-verification and joint verification to show the viability of our approach to multi-property verification. We use benchmarks from the multi-property track of the HWMCC-12 and 13 competitions. 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 $P := P_1 \wedge \dots \wedge P_k$. 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.

Ideally, we would like to use designs with as many properties as possible. However, for a design with a very large number of properties, JA-verification usually outperforms joint verification (see Subsection 9-A). This problem can be addressed by partitioning P_1, \dots, P_k into smaller clusters of properties [8], which is beyond the scope of this paper. To make joint verification more competitive, in Subsections 9-B and 9-C, we picked sixteen designs (eight designs per subsection) that have less than a thousand properties. For these designs, we cross-checked the results of *Ic3-db* in joint verification with those reported by the latest version of ABC [5].²

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 the tables of results. 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. Unfortunately, the HWMCC competitions do not identify properties of multi-property benchmarks that are expected to fail, if any (see Section 5). So, in the experiments we just assumed that every property was expected to hold.

A. A few designs with a large number of properties

In this subsection, we compare JA-verification with joint verification on a few benchmarks that have a very large number of properties. The point we are making here is that for such benchmarks, JA-verification is typically more robust than joint verification. One of the reasons for that is as follows. When one conjoins a set of very different properties joint verification may fail to prove the aggregate property even if all properties are simple individually. For instance, it may be the case that each property of this set depends on a small subset of state variables and hence can be easily proved separately. However, if different properties depend on different subsets of variables, the aggregate property depends on a large subset of state

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

²Joint verification is the natural mode of operation for ABC. 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.

variables and becomes very hard to prove. Another reason for the poor performance of joint verification is that the presence of a few too-hard-to-solve properties P_i can blow up the complexity of the aggregate property P .

TABLE II
A few designs with a large number of properties

name	#all props	#props tried	Joint verification		JA-verification		
			#un-solved	time	time limit	#un-solved	time
6s400	13,784	100	100	10h	0.3h	3	3,167 s
6s355	13,356	100	100	10h	0.3h	2	2,175 s
		100	0	2,817s	0.3h	0	1,974 s
		200	0	1,095 s	0.3h	2	1.1h
6s289	10,789	500	100	10h	0.3h	3	2.1 h
		100	0	919 s	0.3h	0	1,126 s
		200	0	828 s	0.3h	1	2,329 s
6s403	2,382	500	0	717 s	0.3h	1	3,265 s

In this experiment, we used JA-verification and joint verification to verify the first k properties of a benchmark. (Joint verification was performed by *Ic3-db*). The results of the experiment are given in Table II. The value of k is given in the third column. The first and second columns of this table provide the name of a benchmark and the total number of properties. For benchmarks 6s400 and 6s355, JA-verification clearly outperformed joint verification. For benchmark 6s289 both JA-verification and joint verification performed well for $k = 100$ and $k = 200$. However, for $k = 500$ JA-verification outperformed joint verification. Benchmark 6s403 was the only one out of four where joint verification outperformed JA-verification.

B. Designs with failing properties

In this section, we describe an experiment where we verified designs with failing properties. Our objective was to show that solving properties locally can be much more efficient than globally. The results are given in Table III. 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

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

TABLE IV
All properties are true

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

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 of experimental sections, 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.

C. Designs where all properties hold

In this subsection, we describe an experiment with eight designs where all properties were true. The results are given in Table IV. The first three columns are the same as in Table III. The next two columns give run 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 IV 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.

TABLE V

Separate verification with global and local proofs for examples of Table III. Time limit per property is the same as in Table III. The total time limit per benchmark is 10 hours

name	#properties	global proofs		local proofs	
		#unsolved	time	#unsolved	time
6s104	124	123	10h	0	2.5h
6s260	35	36	10h	0	1,686s
6s258	80	70	10h	7	2.4h
6s175	3	1	1,070s	0	554s
6s207	33	23	7.0h	0	22s
6s254	14	0	237s	0	2s
6s335	61	3	3,243s	0	56s
6s380	897	698	10h	0	550s

10. STUDYING JA-VERIFICATION IN MORE DETAIL

A. Comparison of local and global proofs

In this subsection, we describe an experiment where we compared separate verification with global and local proofs. Both versions of separate verification employed clause re-using. (Thus, separate verification with local proofs is JA-verification). In Table V, we compare global and local proofs on benchmarks with failing properties from Table III. The first two columns provide the name of a benchmark and its number of properties. The next two columns specify the performance of separate verification with global proofs. The first column of the two shows how many properties were solved within a 10-hour time limit for the entire benchmark. The time limit per property was the same as in Table III. The second column gives the overall time for a benchmark. The last two columns provide the same information for separate verification with local proofs. Table V shows that separate verification with local proofs dramatically outperforms the one with global proofs.

In Table VI, we compare global and local proofs on benchmarks from Table IV where all properties are true. The structure of Table VI is the same as that of Table V. Table VI demonstrates that both versions of separate verification show comparable performance. A noticeable difference is observed only on benchmarks 6s256 and 6s407. So one can conclude that it is more likely to see the effect of using local proofs on benchmarks with failed properties. On the other hand, the advantage of using local proofs may become more pronounced even on correct designs if the number of properties is large (see Section 11).

TABLE VI

Separate verification with global and local proofs for examples of Table IV. Time limit per property is the same as in Table IV

name	#properties	global proofs		local proofs	
		#unsolved	time	#unsolved	time
6s124	630	0	2.1h	0	1.9h
6s135	340	0	764s	0	746s
6s139	120	2	8.1h	2	6.5h
6s256	5	2	5.7h	1	2.9h
bob12m09	85	0	809s	0	784s
6s407	317	5	5.4h	0	2,077s
6s273	42	0	278s	0	290s
6s275	673	0	1,542s	0	1,611s

TABLE VII

Re-using strengthening clauses in JA-verification. Time limit per property is the same as in Table IV. 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	10h	0	1.9h
6s135	340	0	2.7h	0	746s
6s139	120	116	10h	2	6.5h
6s256	5	0	892s	1	2.9h
bob12m09	85	0	1.1h	0	784s
6s407	317	270	10h	0	2,077s
6s273	42	0	1,445s	0	290s
6s275	673	0	3,273s	0	1,611s

B. Benefit of clause re-using

To illustrate the benefit of re-using strengthening clauses, Table VII compares JA-verification with and without re-using strengthening clauses on the examples of Table IV. Table VII 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.

TABLE VIII

JA-verification with lifting respecting or ignoring property constraints for examples of Table III. Time limit per property is the same as in Table III

name	#properties	respecting prop. constr.		ignoring prop. constr.	
		#unsolved	time	#unsolved	time
6s104	124	0	2.5h	0	2.5h
6s260	35	2	1h	0	1,686s
6s258	80	0	69s	7	2.4h
6s175	3	0	294s	0	554s
6s207	33	0	33s	0	22s
6s254	14	0	2s	0	2s
6s335	61	0	120s	0	56s
6s380	897	0	878s	0	550s

C. JA-verification and state-lifting

As we said in Subsection 7-A, *Ic3-db* proves property P_i locally by treating all the properties P_j , $j \neq i$ as constraints. We also mentioned that this may affect the state lifting procedure used by *Ic3*. In this subsection, we study this issue experimentally by considering two versions of *Ic3-db*. In the first version, property constraints are respected when lifting a state q whereas in the second version these constraints are ignored. Respecting and ignoring property constraints means the following. Let C_q be the cube obtained by lifting state q . When proving property P_i , the first version of *Ic3-db* guarantees that the states of C_q satisfy P_j , $j \neq i$ whereas this is, in general, not true for the second version.

Table VIII shows the results of both versions of *Ic3-db* on the failed designs of Table III. The first two columns give the name of a benchmark and the total number of properties. The next two columns show the results of the first version of *Ic3-db*. These columns give the number of unsolved properties and the total run time. The last two columns provide the same information for the second version. The results of Table VIII show that both versions of *Ic3-db* have comparable performance.

Table IX shows the results of both versions of *Ic3-db* on the designs of Table IV where all properties are true. The structure

TABLE IX

JA-verification with lifting respecting or ignoring property constraints for examples of Table IV. Time limit per property is the same as in Table IV. The total time limit per benchmark is 10 hours

name	#properties	respecting prop. constr.		ignoring prop. constr.	
		#unsolved	time	#unsolved	time
6s124	630	618	10h	0	1.9h
6s135	340	248	10h	0	746s
6s139	120	98	10h	2	6.5h
6s256	5	0	1,282s	1	2.9h
bob12m09	85	0	1,070s	0	784s
6s407	317	0	1.0h	0	2,077s
6s273	42	0	537s	0	290s
6s275	673	0	6.4h	0	1,611s

TABLE X

Verification of single properties of benchmark 6s289 (10,789 properties) using global and local proofs

prop. index	global proof		local proof	
	#time frames	time	#time frames	time
20	11	213 s	1	4.0 s
137	13	289 s	1	3.0 s
500	12	370 s	1	7.7 s
1,001	10	10 s	1	2.5 s
1,310	9	30 s	1	2.4 s
2,678	9	21 s	1	2.3 s
4,789	14	32 s	1	3.5 s
6,600	14	75 s	1	2.8 s
10,002	10	9.9 s	1	2.7 s
max	14	370 s	1	7.7 s

of this table is the same as that of Table VIII. Table IX shows that the first version is faster than the second version only on one benchmark. On the other benchmarks the second version outperforms the first version, sometimes quite dramatically.

11. JA-VERIFICATION AND PARALLEL COMPUTING

Intuitively, JA-verification can significantly benefit from parallel computing. In this section, we give the results of a simple experiment substantiating this intuition. Our interest in a discussion of JA-verification in the context of parallel computing is based on the following two observations. Let \mathbb{P} denote the set of properties $\{P_1, \dots, P_k\}$ to prove. The first observation is that the larger \mathbb{P} , the easier proving $P_i \in \mathbb{P}$ locally due to growing number of constraints P_j , $j \neq i$. The second observation is that the larger \mathbb{P} , the smaller an inductive invariant for property $P_i \in \mathbb{P}$. Hence, when proving different properties of \mathbb{P} locally, the exchange of information (in the form of strengthening clauses) reduces as \mathbb{P} grows. The observations above suggest that by proving properties in parallel one can significantly decrease verification time.

In Table X, we report the results of a simple experiment where we randomly picked individual properties of benchmark 6s289 and proved them both locally and globally. (The total number of properties of 6s289 is 10,789.) The proofs were generated *independently* of each other i.e. there was no exchange of strengthening clauses. The index of selected property is shown in the first column of Table X. The next two columns provide information about the performance of *Ic3-db* when proving the selected property globally. The first column gives the number of time frames *Ic3-db* had to unfold. The second column of the two shows the run time taken by *Ic3-db*.

The next pair of columns provide the same information when *Ic3-db* proved the selected property locally. Table X shows that proving the properties we tried locally was very easy. Assume that finding a local proof for each of the remaining properties of 6s289 takes a very small amount of time as well. Then, if one had, say, 10,789 processors to prove each property P_i on a separate processor, verification would be finished in a matter of seconds.

12. RELATED WORK

We found only a few references to research on multi-property verification. In [9], some modifications of ABC are presented that let it handle multi-property designs. In [8], [10], 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 [11], [12]. 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.

13. 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 $P_1 \wedge \dots \wedge P_k$ 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] B. L. Synthesis and V. Group, "ABC: A system for sequential synthesis and verification," 2017, <http://www.eecs.berkeley.edu/~alanmi/abc>.
- [6] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *FMCAD*, 2011.
- [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] G. Cabodi and S. Nocco, "Optimized model checking of multiple properties," in *DATE*, 2011.
- [9] [Http://people.eecs.berkeley.edu/~alanmi/presentations/Updating_engines00.ppt](http://people.eecs.berkeley.edu/~alanmi/presentations/Updating_engines00.ppt).
- [10] P. Camurati, C. Loiacono, P. Pasini, D. Patti, and S. Quer, "To split or to group: from divide-and-conquer to sub-task sharing in verifying multiple properties," in *DIFTS*, 2014.
- [11] C. Jones, "Specification and design of (parallel) programs," in *IFIP 9th World Congress*, 1983, pp. 321–332.
- [12] 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.