Incremental Formal Verification of Hardware

Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, Ziv Nevo
IBM Research – Haifa
E-mail:{hanac,alexi,ariem,shirim,nevo}@il.ibm.com

***Abstract*—Formal verification is a reliable and fully automatic technique for proving correctness of hardware designs. Its main drawback is the high complexity of verification, and this problem is especially acute in regression verification, where a new version of the design, differing from the previous version very slightly, is verified with respect to the same or a very similar property. In this paper, we present an efficient algorithm for *incremental verification*, based on the ic3 algorithm, that uses stored information from the previous verification runs in order to improve the complexity of re-verifying similar designs on similar properties. Our algorithm applies both to the positive and to the negative results of verification (that is, both when there is a proof of correctness and when there is a counterexample). The algorithm is implemented and experimental results show improvement of up to two orders of magnitude in running time, compared to full verification.**

I. Introduction

Today's rapid development of complex hardware designs requires reliable verification methods. In *formal verification*, we verify the correctness of a design with respect to a desired behavior by checking whether a labeled state-transition graph that models the design satisfies a specification of this behavior, expressed in terms of a temporal logic formula or a finite automaton [CGP99]. The main advantages of formal verification tools are their reliability (if a design passes verification, then it is 100% correct with respect the specification), full automation of the verification process, and the ability of the tools to accompany a negative answer to the correctness query by a counterexample to the satisfaction of the specification in the design [CGMZ95].

The main drawback of the formal verification technology, and the one that prevents it from being even more widely used in the hardware industry, is that it requires significant computational effort, even for moderately sized designs. Moreover, when small changes are introduced into the design or the specification, for example due to a bug fix or an upgrade, the whole design needs to be re-verified, generally requiring the same amount of resources as for the initial verification. The problem is especially acute in *regression verification*, where a new version of a hardware design is re-verified with respect to the same (or very similar) specification. Since regression verification is a only a preliminary (albeit necessary) stage in functional verification of a new version, the time and effort allocated to it are usually much lower than for the initial verification; in reality, since the amount of effort is the same as for the initial verification, our experience is that regression verification is often not performed thoroughly enough, thus possibly leading to lower quality designs.[1] A better option would be to verify the changes *incrementally*, that is, to reuse the results from the previous execution and only verify the change.

Another area in which *incremental verification* techniques are in dire need is *coverage computation* in formal verification. Most of the existing work on coverage in formal verification is based on the notion of *mutation coverage*, where small mutations are introduced to the design and the mutant designs are checked with respect to the original specification [HKHZ99], [CKV06b], [CKV06a], [KLS08], [CKP10], with the goal of checking thoroughness of formal specifications. Efficient incremental verification techniques can reduce the cost of computing mutation coverage, where a large number of slightly modified designs need to be checked with respect to the same property.

Several papers view the problem of incremental verification as an instance of *dynamic graph algorithms*. In this setting, a design is represented as a graph and incremental verification checks the influence of small changes in the graph (edge insertion and removal) on the properties that were previously satisfied in this graph, thus reducing the problem of incremental verification to a dynamic graph problem. However, dynamic graph connectivity, one of the main problems in dynamic graph algorithms, and the one that is most relevant to verification, is an open problem, hence this reduction is of limited value in practice [SBS95], [CK03]. A somewhat related direction is using the reduction to dynamic graph

This work is partially supported by the European Community under the call FP7-ICT-2009-5 – project PINCETTE 257647.

[1] Our experience is based on participating in the formal verification of IBM hardware designs, as well as on discussions with formal verification engineers in other companies.

problems in order to prove complexity results for LTL model checking of evolving designs with non-changing properties [KW03].

The idea of saving the result of model checking in order to use it for subsequent model-checking queries is extensively used in counter-example-guided abstraction refinement (CEGAR) approach [CGJ+03], where the state-explosion problem is addressed by iterative verification and refinement of an abstract design. Abstract counterexamples are analyzed and, if spurious, are used in order to guide the refinement process of the abstraction for the next iteration of the verification process (see also [LBBO01]).

In this paper, we present an approach for re-using the result of model-checking a design (either a proof or a counterexample) for verification of the same or a slightly different property on the same or a slightly modified design. Our method applies both for the case where the result of the model-checking query is negative, in which case we re-use a counterexample, and where it is positive, and we re-use the correctness proof. In fact, the later scenario is very common in regression verification (since the previous version of the design is assumed to pass the verification successfully).

The basis of our work is the novel ic3 ("Incremental Construction of Inductive Clauses for Indubitable Correctness") model checking algorithm, recently proposed by Aaron Bradley [Bra11]. In addition to being one of the the fastest bit-level verification methods [BEM11], [BBC+11], the ic3 algorithm lends itself very naturally to incremental verification.

We describe an algorithm for saving the relevant parts of the proof obtained by ic3, and using them to reproduce the proof (or counterexample) on a new, possibly modified version of the model. This requires very little computational effort in case the same proof works for the new version as well; in case the same proof does not apply, our method attempts to "patch" it by extracting the maximum valid part of the previously saved information and using it as a basis for proving the new version. The latter case, where the original proof does not apply "as is" is the main strength of our method. Roughly speaking, the main idea of our algorithm is as follows. First we observe that producing (and saving) re-usable information does not incur any significant overhead on top of the standard execution of ic3; in addition, the saved part is usually very small. We also describe a query-efficient SAT-based algorithm that we call an *invariant finder* for extracting the maximum valid part of the previously saved information. Then we describe how the valid parts extracted by the invariant finder can be used as a starting point of subsequent ic3's execution, with minor modifications to the algorithm.

Similarly, our algorithm allows to save a small part of a counterexample produced by ic3 that is, nevertheless, sufficient in order to easily reproduce a concrete counterexample and then use this part in order to compute counterexamples for a modified design (or a modified property), by re-using the saved part "as is" or "patching" it to produce a valid counterexample for a modified version. To improve the performance of the algorithm for reusing counterexamples, we propose a simple technique that reduces the size of the partial assignments produced by ic3 by $\approx 30\%$.

The algorithm is implemented in the model-checking engine IVE (*incremental verification engine*), which is a part of the formal verification platform of IBM [Rul], [Six] and is checked on the Hardware Model Checking Competition (HWMCC'10) [HWM10] benchmarks as well as on a real IBM hardware design. Our results show a significant speed-up (up to three orders of magnitude) compared to the re-run of the same model-checking procedure. We note that the performance of IVE is on par with the state-of-the art model checking tools, which makes the speed-up achieved by our incremental verification algorithm even more significant.

The rest of this paper is organized as follows. The necessary definitions and an overview of the ic3 algorithm are provided in Section II. We describe the main contribution of this paper – the invariant finder and the incremental verification algorithms – in Section III, and present the experimental results of executing our implementation on known benchmarks and on an IBM design in Section IV. In Section V we summarize our contributions and discuss possible directions for future work. The complete table of running times and speed-ups achieved by our algorithm on all benchmarks from the HWMCC'10 competition appears in the full version of this paper [Rul].

II. Preliminaries

In this section, we give the necessary definitions for our algorithm and provide overviews of a SAT solver with incremental capabilities and of the ic3 algorithm.

A. Definitions

Throughout this paper we consider verification of safety properties on finite state machines (FSMs). An FSM M is a tuple $\langle X, I, T\rangle$, where X is a set of Boolean state variables, such that each assignment $s \in \{0,1\}^X$ corresponds to a state of M, and the predicates $I \subseteq \{0,1\}^X$ and $T \subseteq \{0,1\}^X \times \{0,1\}^X$ define the initial states and the transition relation of M, respectively. A

predicate $P \subseteq \{0,1\}^X$ defines a property to be verified on M.

State variables and their negations are called *literals*, and disjunctions of literals are called *clauses*. A CNF formula is a conjunction of clauses. (We sometimes refer to CNF formulas as sets of clauses as well.)

We follow the standard notation of $X' = \{x' : x \in X\}$ representing the state variables in the next step, and we assume that the FSMs are given in a representation that allows encoding pairs of states $\langle s, s' \rangle$ into a CNF formula ψ on variables $X \cup X'$, so that $\langle s, s' \rangle \in T$ if and only if ψ is satisfiable (containment in I and P should be similarly expressible with a CNF formula on variables in X).

A sequence π of states $t_0, \ldots, t_n$ is a *path* in M if for each $0 \leq i < n$, $\langle t_i, t_{i+1} \rangle \in T$, that is, there is a transition between each subsequent pair of states in π. A path that starts from an initial state is called an *initialized path*. A state $t \in \{0,1\}^X$ is *reachable* if there is an initialized path that ends in t. Let R denote the set of all reachable states, and for $k \geq 0$, let R_k denote the set of states reachable by initialized paths of length at most k. In particular, $R_0 = I$ and $R_{2^{|X|}} = R$. The goal of a (formal) verification algorithm is to prove $R \subseteq P$, that is, to prove that the property P holds in all reachable states of M.

In all that follows we make definitions and claims with respect to some FSM M on state variables X, with its initial and transition relations I, T, and some property P, without explicitly mentioning them.

Definition 2.1 (invariants and inductive invariants):

- A CNF formula φ is an *invariant* if $s \in R \implies s \models \varphi$. Furthermore, φ is a *k-step invariant* if $s \in R_k \implies s \models \varphi$.
- A CNF formula φ is an *inductive invariant* if $I \implies \varphi$ and $(s \models \varphi \wedge \langle s, s' \rangle \in T) \implies s' \models \varphi$. Similarly, φ is a *k-step inductive invariant* if $I \implies \varphi$ and $(s \in R_{k-1} \wedge s \models \varphi \wedge \langle s, s' \rangle \in T) \implies s' \models \varphi$.

Note that inductive invariance implies invariance (but not vice versa).

Observation 2.1: If φ is an inductive invariant and $\varphi \implies P$, then P holds in all reachable states.

B. SAT solver with incremental capabilities

Our algorithm invokes SAT-based procedures for verifying P on M. In order to allow efficient incremental verification, we use an extended version of *mage*, an IBM SAT solver with incremental capabilities (see the homepage of the formal verification platform of IBM [Rul], [Six], for the description of mage). Incremental capabilities of the extended version of mage are similar to those of MiniSAT [ES03], [EMA10], specifically, the following query is supported (φ is a CNF formula and $\mathcal{A}$ is a set of clauses (assumptions)): $Sat(\varphi, \mathcal{A})$?

- if $\varphi \wedge \mathcal{A}$ is unsatisfiable return UNSAT and a minimal subset $\mathcal{B} \subseteq \mathcal{A}$ so that $\varphi \wedge \mathcal{B}$ is still unsatisfiable[2];
- if $\varphi \wedge \mathcal{A}$ is satisfiable return SAT and a satisfying assignment $\alpha \in \{0,1\}^X$;

For example, the following queries (referring to the given FSM) can be translated to a single query to the SAT solver:

- $s \in I$? (is s an initial state?)
- $\varphi \wedge T \wedge \neg\varphi'$? (Is there a pair of states, $\langle s, s' \rangle \in T$, so that s satisfies φ but s' does not? If the answer is yes we can also extract a witness pair of states $\langle s, s' \rangle$ from the assignment returned by the solver.)

C. Overview of ic3

In this section we provide a brief overview of the ic3 algorithm and highlight the features of ic3 that are relevant to incremental verification. For a more in-depth description of ic3 the reader is refered to the original paper by Bradley [Bra11] and to the paper of Brayton et al. [BEM11], who provide an overview of ic3 and present *pdr* – an improved version of ic3.

The main advantage of ic3 is its ability to perform unbounded SAT-based model checking without unfolding the transition relation. Given a model checking instance consisting of an FSM M and a property P as defined in Section II-A, the ic3 algorithm decides whether P is an invariant in M, producing an inductive strengthening if so, and a counterexample trace if not. The algorithm proceeds by incrementally refining and extending a sequence $\mathcal{F}_1, \ldots, \mathcal{F}_k$ of sets of clauses, each $\mathcal{F}_i$ forming an i-step inductive invariant CNF formula. Initially $k = 1$, and it gradually grows until termination. Furthermore, if M satisfies P (P holds in all reachable states) then on termination of ic3, for some $i \leq k$, the set $\mathcal{F}_i$ [3] forms an inductive invariant CNF formula that implies P:

- $I \implies c$ for every clause $c \in \mathcal{F}_i$;
- $\mathcal{F}_i \wedge T \implies \mathcal{F}_i'$;
- $\mathcal{F}_i \implies P$;

If M does not satisfy P then ic3 produces a *set of counter-examples* in form of a sequence $\alpha_0, \ldots, \alpha_k$ of partial assignments to X. In this sequence

[2]Although obtaining the minimal subset is hard, there are efficient ways to compute subsets that tend to be quite small in practice.

[3]$\mathcal{F}_i$ is the set that becomes empty during "clause pushing".

- all α_0 states (states formed by extending α_0 to a full assignment) are in I;
- all α_k states are **not** in P;
- all α_i states lead to some α_{i+1} state;

A concrete counter-example (CEX) may be extracted from such a sequence using $k+1$ calls to a SAT solver.

D. Additions to ic3

a) Shrinking partial assignments: Given a pair (s, α), where s is a full assignment to X (describing a state), α is a partial assignment to X' (describing a set of states) and there is an α-state t such that $\langle s, t\rangle \in T$, "shrinking" is the process of generalizing s into a set of states (represented by a sub-assignment of s) all leading to some α-state in one step.

The optimization from [BEM11] shrinks assignments using ternary simulation, and it is indeed very efficient. Here we propose a different method to further shrink the assignments using a SAT solver. This is described in detail in Section III-C.

b) Injecting invariants: We note that instead of starting "from scratch", ic3 can take, as input, a set $\mathcal{I}$ of invariant clauses, and use it as an absolute invariant: during its entire execution, all clauses from $\mathcal{I}$ can be directly injected into each of the sets $\mathcal{F}_i$.

III. Algorithm

In this section we present the main contribution of this paper – an algorithm for efficient incremental verification. We start with an overview of the algorithm, divided into an overview of the invariant finder and an overview of the incremental verification algorithm, which combines the invariant finder and ic3. Then we describe the algorithm in more detail, including some additional (smaller) contributions that further improve its performance.

A. Overview

We start with an overview of the *invariant finder*. Note that in addition to playing a crucial role in our algorithm for incremental verification, the invariant finder might be of independent interest. Invariant finder takes a model M and an arbitrary set $\mathcal{C}$ of (candidate invariant) clauses as input, and finds the *maximum* subset $\mathcal{I} \subseteq \mathcal{C}$ that is an inductive invariant with respect to M, i.e.:

- $I \implies c$ for every clause $c \in \mathcal{I}$;
- $\mathcal{I} \wedge T \implies \mathcal{I}'$;

The general idea of generating and exploiting inductive invariants in formal verification is not new (see e.g. [CCG+09], [CNQ09], [CMB07], [BMC+09]). The novelty of our algorithm is in the way it extracts the maximum set of inductive invariants (from an arbitrary set of candidates) using a SAT solver with incremental capabilities described in Section II-B; in particular, as the experimental results show, our algorithm is very efficient in practice (see Section IV).

The *incremental verification* algorithm combines ic3 and the invariant finder for storing and re-using information from previous verification runs in order to speed up subsequent verification on modified models and properties as follows.

If the result of the verification run of P on M is positive, the ic3 algorithm produces an invariant set $\mathcal{I}$ of clauses that implies the property P on M. We use the invariant finder to extract from $\mathcal{I}$ the largest invariant subset that holds on a modified model and provide this subset as a starting point to ic3 (see Section II-D).

If the verification of P fails on M, the set of counterexamples generated by ic3 is saved in a form of partial assignments, together with the clauses $\mathcal{C} \triangleq \bigcup_{i=1}^{k} \mathcal{F}_i$. Then, in subsequent runs we check whether the saved partial assignments can be extended to full assignments that produce a counterexample that is valid in a modified model with a modified property. If so, we have found a valid counterexample; otherwise, we extract the maximal inductive invariant from $\mathcal{C}$, and provide this subset as a starting point to ic3.

B. Detailed description of the algorithm

Invariant finder: Recall that our task is: Given a candidate set $\mathcal{C}$ of clauses, find its maximal subset $\mathcal{I}$ that is inductive invariant with respect to a given model M. To this end, we first check if the whole set $\mathcal{C}$ is inductive invariant by making the query $\mathcal{C} \wedge T \implies \mathcal{C}'$. If so, we are done; otherwise, there are clauses $c \in \mathcal{C}$ not implied by $\mathcal{C} \wedge T$. We then update $\mathcal{C}$ by removing (possibly a subset of) such clauses, and repeat.

To make this straightforward process more practical, we encode the SAT queries using auxiliary variables so that 1) the learnt information in the solver can be re-used from iteration to iteration; 2) $\mathcal{C}$ gets updated quickly – by detecting many non-implied clauses c simultaneously.

Specifically, we propose the following algorithm:

1) Remove from $\mathcal{C}$ all clauses not in I;
2) For each clause $c_i \in \mathcal{C}$ (whose literals refer to current cycle)
   - introduce two auxiliary variables x_i and y_i;
   - introduce the "shifted" copy c'_i of c_i (whose variables refer to the next cycle);
   - add the clause $(\neg x_i \vee c_i)$ to the solver (this is equivalent to $x_i \implies c_i$);
   - for each literal $a'_{i,j}$ of c'_i, add the binary clause $(y_i, \neg a'_{i,j})$ to the solver (these clauses are equivalent to $c'_i \implies y_i$)

3) Initialize $it \leftarrow 0$, $\mathcal{I}_{it} \leftarrow \mathcal{C}$;
4) while $\mathcal{I}_{it} \neq \emptyset$ do:
   a) If $Sat(\{(x_1), ..., (x_{|\mathcal{I}_{it}|}), (\neg y_1 \vee \ldots \vee \neg y_{|\mathcal{I}_{it}|})\})$ is UNSAT report "$\mathcal{I} \triangleq \mathcal{I}_{it}$ is invariant";
   b) Else, let α denote the satisfying assignment that respects the assumptions $(x_1), ..., (x_{|\mathcal{I}_{it}|}), (\neg y_1 \vee \ldots \vee \neg y_{|\mathcal{I}_{it}|})$. Form $\mathcal{I}_{it+1}$ by removing from $\mathcal{I}_{it}$ all clauses with indices corresponding to each y_i assigned to 0 in α (see Remark 3.1), update x_i's and y_i's accordingly, and proceed with $it \leftarrow it + 1$;
5) Report "no invariant";

Remark 3.1:

- Observe that in Step 4b there must be at least one such y_i assigned to 0, thus in the worst case the number of iterations and SAT calls until termination is bounded by $|\mathcal{C}| - |\mathcal{I}|$. In practice, however, many y_i's may be assigned to 0 at once, making the loop terminate faster.
- Note that all SAT queries involve only a single copy of transition relation.
- Note that the invariant finder can take any set of clauses as the candidate set $\mathcal{C}$, thus there is no need to worry about validity of the previously saved information.

Claim 3.1: Invariant finder always outputs the maximum inductive invariant subset $\mathcal{I} \subseteq \mathcal{C}$ (which may be an empty set).

Proof: First, observe that there is a unique maximum inductive invariant subset; indeed, it is easy to verify that if $\mathcal{A}$ and $\mathcal{B}$ are inductive invariant subsets of $\mathcal{C}$, then so is $\mathcal{A} \cup \mathcal{B}$.

Let $\mathcal{I}$ be the output of the invariant finder upon termination. By definition, since in every iteration we check if $\mathcal{I}_{it} \wedge T \implies \mathcal{I}'_{it}$, the set $\mathcal{I}$ with which the loop terminates is clearly inductive invariant, thus we only need to argue that it is maximal. Let $\mathcal{I}^*$ denote the maximal invariant subset of $\mathcal{C}$, and assume towards a contradiction that $\mathcal{I}^* \not\subseteq \mathcal{I}$. Let it denote the first iteration in which some clause of $\mathcal{I}^*$ was removed from $\mathcal{I}_{it}$, that is, $\mathcal{I}^* \subseteq \mathcal{I}_{it}$ but $\mathcal{I}^* \not\subseteq \mathcal{I}_{it+1}$ (such it must exist since $\mathcal{I}^*$ was initially contained in $\mathcal{I}_0$). This means that $\mathcal{I}_{it} \wedge T$ did not imply $\mathcal{I}^*$, contradicting the inductiveness of $\mathcal{I}^*$. ■

Incremental verification: First, let us consider the (more challenging) case where the design passes the verification, namely, P holds in all reachable states of M. As explained in Section II-C, ic3 produces an invariant set $\mathcal{I}$ of clauses, and we can save it (in form of a standard CNF file) as the "proof of correctness" (see Observation 2.1).

Now assume that we want to *re-verify* P on the same model; this amounts to verifying the validity of $\mathcal{F}_i$, which is done in just three SAT calls:

1) $I \implies \mathcal{I}$?
2) $\mathcal{I} \wedge T \implies \mathcal{I}'$?
3) $\mathcal{I} \implies P$?

To summarize, if the saved proof is re-used for the same model, the verification is completed almost immediately (see Table I).

In case a model or a property are modified, our algorithm invokes the invariant finder to extract from $\mathcal{I}$ the largest invariant subset $\hat{\mathcal{I}}$ (with respect to the updated model), and injects it into ic3's data-structure as described in Section II-C. In particular, after this step, ic3 does not need to rediscover all the invariant clauses that hold both in the original and modified models. We note here that discovering a single invariant clause in ic3 requires several (often more than its size) SAT calls; hence, quite naturally, the amount of work that is saved by re-using the previous verification results corresponds to the amount of overlap between the two models.

Now let us consider the case where the verification fails and a counterexample is produced. We can store the set of counter-examples generated by ic3, in form of the aforementioned partial assignments $\alpha_0, \ldots, a_k$, and the set of clauses $\mathcal{C} \triangleq \bigcup_{i=1}^{k} \mathcal{F}_i$. To check if a concrete counter-example can be extracted in a future run (on a possibly modified model) we make $k+1$ SAT calls, essentially asking for a sequence $s_0, \ldots, s_k$ of full assignments to X with the following properties:

1) s_i is an extension of α_i for all $i \leq k$, and in particular, $s_0 \in I$;
2) $\langle s_i, s_{i+1} \rangle \in T$ for all $i < k$;
3) $s_k \notin P$.

If a counterexample cannot be extracted, we proceed with the usual execution of ic3, but first we attempt, using the invariant finder, to find an inductive invariant subset in $\mathcal{C}$ to use it as a starting point of ic3.

It is important to make the partial assignments $\alpha_0, \ldots, \alpha_k$ as small as possible (in the initial run), so that extracting a concrete counter-example becomes possible even when the modified model significantly differs from the original model. In Section III-C we discuss the improvements we added to the ic3 algorithm that enable to "shrink" the partial assignments computed by ic3.

C. Shrinking partial assignments with solver

We introduce an additional improvement of the algorithm allowing us to further shrink the partial assignments produced by ic3. The goal of this improvement is

to enlarge the set of valid counter-examples as discussed in Section II-D.

For some $i < k$, all α_i states can reach some α_{i+1} state in one step. Shrinking α_i results in an increase of the number of states that can lead to some α_{i+1} state, and is done using a single SAT call (and in fact this SAT call only involves BCP).

Specifically, given a pair $\langle s, s' \rangle \in T$ and the corresponding input values β (under which the transition $(s = \alpha \wedge inp = \beta) \rightarrow s'$ holds), we can shrink α to a partial assignment so that all α states lead to s' in one transition. To this end, we query the solver (knowing in advance that the answer is negative) if $(s = \alpha) \wedge (inp = \beta) \wedge \langle s, \hat{s}' \rangle \in T \wedge (\hat{s}' \neq s')$ is satisfiable. The first condition is passed to the solver in form of $|\alpha|$ assumptions, so that it also returns the minimal subset of those assumptions required for the conflict (see Section II-B). Then we can safely remove from α all those indices that are not required for reaching the conflict.

We conjecture that the reason why this additional step is effective is that instead of only looking at the structure of the model (as in ternary simulation) it takes into account all learned information (invariants from ic3 and the learned clauses from solver) to rule out unreachable states that do not lead to any α_{i+1} state.

IV. Experimental results

We implemented our algorithm for incremental verification in IVE (*incremental verification engine*), which is a part of the formal verification platform of IBM [Rul], [Six], and measured its performance on known benchmarks and on a real IBM hardware design.

HWMCC'10 benchmarks

The first set of experiments is based on the benchmarks used in the Hardware Model Checking Competition [HWM10] that was a part of the first Hardware Verification Workshop (HVW'10), affiliated with Computer-Aided Verification (CAV) Conference in 2010. We note that, out of the 758 publicly available benchmarks, IVE successfully verified 713 within 15 minutes, while the winner of the HWMCC'10, an engine *abcsuperprove* [abc] from Berkeley, verified 717 benchmarks within 15 minutes on comparable machines[4]. In other words, the performance of IVE is comparable to the state-of-the-art model-checking tools.

For each benchmark, we measured the running time of the verification procedure (setting 1 hour time limit), and the running time of incremental verification of the *same benchmark* (in other words, re-verification based on the results of the previous verification procedure). This setting emulates the most common scenario in incremental verification, where the changes introduced into the design do not, in fact, affect the verification at all, either because they fall outside of the cone of influence of the verified properties, or because they are "filtered out" during the preliminary reductions.

For each benchmark, we also measured the running time of IVE with incremental verification after small changes were introduced into the design. We simulated introduction of a small change by a *random mutation* in 1% of the assignments in the original instances (represented in AIG form) as follows. An assignment of the form $res = \ell_1 \& \ell_2$ was selected with probability 0.01 and mutated into one of the following assignments: $\{res = 0,\ res = 1,\ res = \neg\ell_1 \& \ell_2,\ res = \ell_1 \& \neg\ell_2,\ res = \neg\ell_1 \& \neg\ell_2,\ res = \ell_1,\ res = \neg\ell_1,\ res = \ell_2,\ res = \neg\ell_2\ \}$ with equal probability. In the absence of a domain-specific knowledge about the structure of the design, random mutations are the best approximation of small changes introduced into the design, and they also ensure that the experimental results are not biased towards any specific type of changes. We then measured the running time of IVE with incremental verification for the original benchmarks after the mutated ones and of the mutated ones after the original ones. In each case, the results of the previous verification were saved and used by the subsequent verification. The detailed results are presented in the full version of this paper [Rul]; Table I contains their summary. The row *"overall"* contains the results summarized for all benchmarks together, thus the speed-up represents a speed-up that is achieved by model-checking all benchmarks one after another. The overall speed-up is by the factor of 76 for re-verification of the same instance, and is by the factor of 3 after a small mutation was introduced. The median speed-up shows only a very slight improvement of our technique compared to re-verification without using the previous results. However, this is mostly due to the fact that median is dominated by very light instances, that constitute the vast majority of HWMCC'10 benchmarks. The most significant improvement in the running time was achieved for heavy instances: indeed, the median speedup (rerun vs. original) computed on instances that take > 60 seconds to solve is larger by two orders of magnitude (277 vs. 1.2).

IBM hardware design

In the second set of experiments, based on a real and up-to-date IBM hardware design, we measured the

[4]HWMCC'10 used Intel Quad Core 2.6 GHz with 8 GB; we used Intel Quad Core 2.6 GHz with 2 GB.

	original	rerun	speedup	original after mutated	speedup	mutated	mutated after original	speedup
Overall:	30597.01	402.89	75.92	10070.84	3.04	50294.46	37348.26	1.35
Average	42.49	0.55	114.06	13.98	1.80	69.85	51.87	2.95
Median	0.175	0.11	1.20	0.13	1.43	0.43	0.15	3.00

TABLE I
SUMMARY OF RUNTIMES ON 721 BENCHMARKS FROM HWMCC'10.

benefit of our algorithm when the changes between the two verification runs are significant and represent real changes in the design. Namely, we checked our algorithm on two versions of a model composed of a hardware design, augmented with a driver and a set of properties. The design implements logic that responds to requests from several threads. The two versions differ only in the driver: In the first ($1T$: 19,822 state variables and 299,185 gates before reductions; 2,187 state variables and 55,756 gates after reductions) the driver allows requests from a single thread, disabling all others. In the second version ($8T$: 19,831 state variables and 300,316 gates before reductions; 2,249 state variables and 56,458 gates after reductions), the driver enables 8 requesting threads. Therefore, in the first version all interleavings and priority-based decisions between threads are disabled, creating a significant difference in behaviors between the first and the second version. The verification suite for the design consists of 17 temporal logic properties. Table II presents the results of executing IVE with incremental verification implementation for both versions on all properties, including the original running time, the re-verification running time, and the running time of verifying one model after another.

The most important number in Table II is the accumulated speed-up, presented in the row ''overall". This number represents the performance gain of incremental verification in the scenario where the whole verification suite is re-checked in the design, which is a typical scenario after a bug fix and in regression verification.

The accumulated speed-up between the original run of model $1T$ and its re-run, presented in the row ''overall", is by the factor of ≈ 30, compared with re-run of the verification "from scratch"; the accumulated speed-up between the original run of model $8T$ and its re-run is by the factor of ≈ 61. These numbers model a performance gain in a typical scenario of regression verification, when the changes do not affect the properties at all.

The accumulated speed-up between the original run of model $1T$ and the run of model 1T after model $8T$ is by the factor of ≈ 3, and the accumulated speed-up of model $8T$ after model $1T$ is by the factor of ≈ 2. These numbers show that even if a change in the model is wide and applies to all behaviors of the design, using the incremental verification techniques has the potential of reducing the running time quite significantly. The difference in speed-up between verifying the $8T$ model after $1T$ and the $1T$ after the $8T$ is due to the fact that $1T$ has a small fraction of behaviors of $8T$, and hence the proof of $1T$ is only a small step in proving $8T$; on the other hand, the correctness of $1T$ for most part follows from the correctness of $8T$.

It is clear that in some cases, incremental verification techniques will not be beneficial in improving performance; after all, our algorithm incurs an additional computational cost in analyzing stored data. It is easy to see that this situation occurs when the instance is solved almost immediately, thus making the time required for analyzing the stored data very significant in the overall performance estimation – see, for instance, some of the results for properties $\varphi 07$ and $\varphi 08$ in Table II. We also note that the speed-up of this set of experiments is in most cases significantly smaller than the speed-up of the HWMCC'10 benchmarks; this is, again, due to the contribution of the time required to analyze the stored data in this relatively big design.

Overall vs. median and average speed-up: The median and average speed-ups in Tables I and II are affected by the (negligible) speed-up achieved on very light instances, where the analysis of the stored data is the major component in the overall verification. On the other hand, the *overall* speed-up is computed by dividing the sum of the running times of the original verification by the sum of the running times of incremental verification of all instances. Thus, the speed-up achieved on heavy instances, which are also the most important target for the application of incremental verification, is more accurately represented by the overall speed-up column.

The order of verification: Table I presents the results of executing incremental verification of the original designs after the mutant designs and vice versa. It is easy to see that the speed-up achieved by re-verifying the original design after the mutant design is larger (by the factor of 2) than the speed-up achieved by re-verifying the mutant design after the original. It is hard to say whether this difference is meaningful; SAT solvers are based on heuristic techniques, and the correlation of the size of an instance with the time required to solve it is not always clear. We conjecture that the difference may stem from the fact that mutations create SAT instances

property	1T	1T after 1T	speedup	1T after 8T	speedup	8T	8T after 8T	speedup	8T after 1T	speedup
ϕ01	1	0	∞	1	1.00	1	2	0.50	2	0.50
ϕ02	2330	299	7.79	3336	0.70	5603	129	43.43	1742	3.22
ϕ03	95	96	0.99	124	0.77	86	48	1.79	58	1.48
ϕ04	4913	91	53.99	1777	2.76	13458	234	57.51	1231	10.93
ϕ05	204	12	17.00	238	0.86	786	83	9.47	630	1.25
ϕ06	77	19	4.05	863	0.09	112	7	16.00	13	8.62
ϕ07	1	3	0.33	1	1.00	6	2	3.00	2	3.00
ϕ08	0	2	0.00	1	0.00	0	1	0.00	0	∞
ϕ09	13636	332	41.07	3848	3.54	13602	121	112.41	10291	1.32
ϕ10	8	29	0.28	174	0.05	15	1	15.00	0	∞
ϕ11	13823	79	174.97	2	6911.50	17421	55	316.75	9658	1.80
ϕ12	17	96	0.18	37	0.46	9	5	1.80	3	3.00
ϕ13	129	1	129.00	139	0.93	106	21	5.05	2	53.00
ϕ14	177	72	2.46	220	0.80	108	13	8.31	3	36.00
ϕ15	135	8	16.88	170	0.79	250	1	250.00	5	50.00
ϕ16	651	6	108.50	30	21.70	1814	36	50.39	32	56.69
ϕ17	407	93	4.38	749	0.54	779	116	6.72	772	1.01
Overall	36605	1238	29.57	11710	3.13	54160	883	61.34	24447	2.22

TABLE II
TWO VERSIONS OF AN IBM DESIGN – ONE THREAD AND EIGHT THREADS. ORIGINAL AND RERUN TIMES IN SECONDS.

that do not always correspond to real designs. Since SAT solvers are fine-tuned to efficiently solve real designs, introducing a small mutation can cause a significant increase in the number of clauses that SAT solver is required to learn in order to solve the instance.

V. CONCLUSIONS

We described a novel algorithm for incremental model-checking of hardware. Our algorithm is partially based on an improved version of the ic3 algorithm and it relies on the results of the previous verification procedure in order to improve the complexity of verification after a small change was introduced in either the design or the property. Our algorithm applies both to the case where the original model-checking procedure failed producing a counter-example, and to the case where the original model-checking was successful producing a proof of correctness. The algorithm requires storing a minimal amount of information from the proof or a counterexample, hence the overhead for the initial verification is negligible. We implemented our algorithm in IVE, an engine which is a part of the formal verification platform of IBM. We measured the performance improvements obtained by our implementation on publicly available benchmarks and on a real IBM design. The performance of IVE engine is on par with the state-of-the-art model checkers on known benchmarks, and we demonstrate that with our implementation we are able to achieve a speed-up of up to two orders of magnitude on "heavy" instances for re-verification after a small change.

To conclude, our technique clearly presents significant performance improvements, even when the difference between the original model and the changed model is significant. In fact, when we compare the original model-checking execution and re-verification of the same model, the speed-up is usually huge. We consider this result to be especially significant, because in the standard re-verification scenario – regression verification – the changes in the design are usually very small, and are often outside the cone of influence of the verification procedure.

REFERENCES

[abc] Abc homepage. http://www.eecs.berkeley.edu/ alanmi/abc/.

[BBC+11] Jason Baumgartner, Robert Brayton, Gianpiero Cabodi, Niklas Eén, Alexander Ivrii, Arie Matsliah, Alan Mishchenko, and Hari Mony. Personal communication. 2011.

[BEM11] Robert Brayton, Niklas Eén, and Alan Mishchenko. Efficient implementation of property directed reachability. In *IWLS*, 2011.

[BMC+09] Jason Baumgartner, Hari Mony, Michael L. Case, Jun Sawada, and Karen Yorav. Scalable conditional equivalence checking: An automated invariant-generation based approach. In *FMCAD*, pages 120–127, 2009.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[CCG+09] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. Speeding up model checking by exploiting explicit and hidden verification constraints. In *DATE*, pages 1686–1691, 2009.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd DAC*, pages 427–432. IEEE Computer Society, 1995.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CK03] G. Cohen and O. Kupferman. Incremental LTL model checking. In *Proc. Workshop on semantics and verification of hardware and software systems*, 2003.

[CKP10] Hana Chockler, Daniel Kroening, and Mitra Purandare. Coverage in interpolation-based model checking. In *DAC*, pages 182–187, 2010.

[CKV06a] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for formal verification. *STTT*, 8(4-5):373–386, 2006.

[CKV06b] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. *Formal Methods in System Design*, 28(3):189–212, 2006.

[CMB07] Michael L. Case, Alan Mishchenko, and Robert K. Brayton. Automated extraction of inductive invariants to aid model checking. In *FMCAD*, pages 165–172, 2007.

[CNQ09] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Strengthening model checking techniques with inductive invariants. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(1):154–158, 2009.

[EMA10] Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction. *CoRR*, abs/1008.2021, 2010.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *ENTCS*, 89(4), 2003.

[HKHZ99] Y. Hoskote, T. Kam, P.-H Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. 36th DAC*, pages 300–305, 1999.

[HWM10] Hardware model checking competition, 2010. http://fmv.jku.at/hwmcc10/.

[KLS08] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD*, pages 1–9, 2008.

[KW03] Detlef Kähler and Thomas Wilke. Program complexity of dynamic LTL model checking. In *Proceedings of CSL*, pages 271–284, 2003.

[LBBO01] Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In *TACAS*, pages 98–112, 2001.

[Rul] Rulebase homepage. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage.

[SBS95] Gitanjali Swamy, Robert K. Brayton, and Vigyan Singhal. Incremental methods for FSM traversal. In *ICCD*, 1995.

[Six] Sixthsense homepage. http://domino.research.ibm.com/comm/research_projects.nsf/pages/sixthsense.index.html.