Pushing to the top

Alexander Ivrii
IBM Research
alexi@il.ibm.com

Arie Gurfinkel
Software Engineering Institute
http://arieg.bitbucket.org

***Abstract*—IC3 is undoubtedly one of the most successful and important recent techniques for unbounded model checking. Understanding and improving IC3 has been a subject of a lot of recent research. In this regard, the most fundamental questions are how to choose Counterexamples to Induction (CTIs) and how to generalize them into (blocking) lemmas. Answers to both questions influence performance of the algorithm by directly affecting the quality of the lemmas learned. In this paper, we present a new IC3-based algorithm, called QUIP[1], that is designed to more aggressively propagate (or push) learned lemmas to obtain a safe inductive invariant faster. QUIP modifies the recursive blocking procedure of IC3 to prioritize pushing already discovered lemmas over learning of new ones. However, a naive implementation of this strategy floods the algorithm with too many useless lemmas. In QUIP, we solve this by extending IC3 with may-proof-obligations (corresponding to the negations of learned lemmas), and by using an under-approximation of reachable states (i.e., states that witness why a may-proof-obligation is satisfiable) to prune non-inductive lemmas. We have implemented QUIP on top of an industrial-strength implementation of IC3. The experimental evaluation on HWMCC benchmarks shows that the QUIP is a significant improvement (at least 2x in runtime and more properties solved) over IC3. Furthermore, the new reasoning capabilities of QUIP naturally lead to additional optimizations and new techniques that can lead to further improvements in the future.**

I. Introduction

IC3 [1] (also known as PDR [2]) is one of the most powerful algorithms for unbounded model checking of hardware. It is highly customizable [3], [4], and was successfully extended to more general domains [5]–[7].

In a nutshell, IC3 aims at constructing an inductive invariant proving the property. IC3 works by iteratively detecting states that lead to a property violation (in IC3-literature these states are also identified with counterexamples-to-induction and are called CTIs) and by learning lemmas that demonstrate why these CTIs cannot be reached from the initial states within a bounded number of steps. In this way, IC3 incrementally refines over-approximations F_k of states that are reachable in up to k steps, and terminates when one of the sets F_k represents a safe inductive invariant, or when a counterexample is found. The general scope of this paper is to further improve on the invariant generation capabilities of IC3. In what follows, we first analyze and discuss some of the choices made by IC3, and then present our approach.

One of the most important decisions made by IC3 pertains to the process of generalization of new lemmas at the time when they are discovered. Ideally, given a CTI, we would like to generate the strongest possible lemma that excludes this CTI and holds on all reachable states. However, obviously the set of all reachable states is not available. IC3 solves this problem by attempting to find the strongest lemma φ that is relatively inductive with respect to the appropriate over-approximation F_k. However, as F_k is neither an over-approximation nor an under-approximation of the set of all reachable states, φ can be either too strong or too weak. Being too strong means that φ excludes some of the reachable states and hence has no chance to be in the final inductive invariant, while being too weak means that φ prunes less unreachable states which degrades convergence. Another deficiency of IC3 is that once a lemma is added, it remains in the system, and there is no mechanism to detect and prune non-inductive lemmas, which translates to the wasted effort spent to propagate them.

An important optimization that already exists in IC3 consists of blocking the same CTI at many different levels. In our experience, IC3 often discovers many different lemmas to block the same CTI. On the one hand, different lemmas are in general of different quality and so having a variety of lemmas to choose from is beneficial. On the other hand, keeping several lemmas for the same CTI leads to a wasted effort of storing and pushing multiple lemmas when one would be enough. IC3 partially addresses this concern by pushing each lemma as far as possible when it is created (which implicitly blocks the corresponding CTIs at higher levels); however, it often happens that a lemma φ cannot be pushed forward because the appropriate over-approximation F_k is not strong enough. An alternative solution is to derive additional supporting lemmas that enable pushing φ forward, thus prioritizing the usage of a lemma already in the system, at the expense of finding additional lemmas required to support it. We believe that the new strategy is superior, as it should lead to an inductive invariant faster. Unfortunately, a naïve implementation forces the algorithm to start discovering new lemmas to support lemmas already in the system, and then new lemmas to support these supporting lemmas, and so on – flooding the algorithm with a huge number lemmas. To some extent the problem again boils down to lack of control on the usefulness of lemmas in the system, and the need to detect and prune the less useful ones.

In this paper, we present an improvement to the core of the IC3 algorithm. Motivated by the considerations above, we present an algorithm, called Quip, that combines the following innovations:

1) In Quip, we periodically detect the maximal inductive subset of all lemmas discovered so far. These lemmas are stored separately (in F_∞ in the terminology of PDR) and

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002441.

[1]QUIP is an acronym for "a QUest for an Inductive Proof".

represent *good* lemmas – lemmas that should always remain in the system.

2) In Quip, we turn existing lemmas into additional proof obligations (and prioritize considering these proof obligations over regular proof obligations). Given $\varphi \in F_k \setminus F_{k+1}$, we add $\neg\varphi$ at level $k+1$ as a *may-proof-obligation*. In this way, we either succeed to push φ further (if $\neg\varphi$ is blocked), or find a witness trace that explains why φ cannot be pushed. Since $\neg\varphi$ does not necessarily represent a CTI, the witness trace does not necessarily lead to a property violation; however, it produces a concrete forward reachable state that is excluded by φ and hence which explains why φ is not inductive. In particular, φ is a *bad* lemma – lemma that has no chance to be in the inductive invariant.

3) In Quip we dynamically discover reachable states. These reachable states are used in several ways. First, each time that a new reachable state is discovered, it is used to mark as bad all lemmas in the system that exclude this state. Second, reachable states are used to automatically invalidate other may-proof-obligations or to discover a real counterexample. Finally, they are used to effectively enlarge the set of initial states and take the enlarged initial states into account when generalizing lemmas in the future.

Note that the ideas above are highly interdependent: without considering may-proof-obligations there is no way to produce interesting reachable states, while without considering reachable states there is no way to prune lemmas in the system. We also claim that Quip partially addresses the problems described in the beginning. By prioritizing may-proof-obligations over regular-proof-obligations, we try to reuse lemmas that already exist. In addition, as may-proof-obligations usually consist of significantly fewer literals than regular proof obligations, we effectively try to avoid detecting lemmas that are too weak, while by computing and using the set of reachable states for generalization, we also try to avoid detecting lemmas that are too strong. Finally, we can now classify lemmas as *good*, *bad* and *unknown*, and thus gain some control on which lemmas we want to propagate and keep, and which lemmas we do not. In what follows, we show how to integrate the presented ideas into an efficient algorithm and experimentally demonstrate that this represents a significant performance improvement over classical IC3.

We believe that our work extends the IC3 framework with additional reasoning capabilities: computing maximal inductive invariants, considering may-proof-obligations and forward reachable states. These naturally lead to other optimizations and new techniques that can lead to further improvement in the future. Last but not least, the new framework can be used with all other known IC3 optimizations and can be adapted to more general domains.

The rest of the paper is structured as follows. In Section II, we review the necessary background about IC3. We present the Quip algorithm at high-level in Section III, and the details of our implementation in Section IV. Our empirical evaluation is reported in Section VI. Finally, we conclude the paper with an overview of related work in Section VII, and conclusion in Section VIII.

II. Background

Let $\mathcal{V}$ be a set of variables. A *literal* is either a variable $b \in \mathcal{V}$ or its negation $\neg b$. A *clause* is a disjunction of literals. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A *cube* is a conjunction of literals. A Boolean formula in *Disjunctive Normal Form (DNF)* is a disjunction of cubes. It is often convenient to treat a clause or a cube as a set of literals, a CNF as a set of clauses, and DNF as a set of cubes. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

Let $\mathcal{V}$ be a set of variables and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. A safety verification problem is a tuple $P = (Init, Tr, Bad)$, where $Init(\mathcal{V})$ and $Bad(\mathcal{V})$ are formulas with free variables in $\mathcal{V}$ denoting initial and bad states, respectively, and $Tr(\mathcal{V}, \mathcal{V}')$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ denoting the transition relation. Without loss of generality, we assume that $Init$ and Tr are in CNF.

The verification problem P is SAT (or UNSAFE) iff there exists a natural number N such that the following formula is SAT:

$$Init(\vec{v}_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(\vec{v}_i, \vec{v}_{i+1}) \right) \wedge Bad(\vec{v}_N) \tag{1}$$

P is UNSAT (or SAFE) iff there exists a formula $Inv(\mathcal{V})$, called *a safe invariant*, that satisfies the following conditions:

$$Init(\vec{v}) \rightarrow Inv(\vec{v}) \qquad Inv(\vec{v}) \wedge Tr(\vec{v}, \vec{v}') \rightarrow Inv(\vec{v}') \tag{2}$$

$$Inv(\vec{v}) \rightarrow \neg Bad(\vec{v}) \tag{3}$$

A formula Inv that satisfies (2) is called an *invariant*, while a formula Inv that satisfies (3) is called *safe*.

We give a brief description of IC3 that highlights some steps, but omits many crucial optimizations. We refer the reader to [8] for an overview of available optimizations and their possible implementations.

IC3 maintains a sequence of sets of clauses $F_0, F_1, \ldots$ called a *trace*. Each set of clauses F_i in a trace is called a *frame*, each clause $c \in F_i$ is called a *lemma*, and the index of a frame is called a *level*. We assume that F_0 is initialized to $Init$ and that $Init \rightarrow \neg Bad$. IC3 maintains the following invariant:

$$F_i \rightarrow \neg Bad \qquad F_{i+1} \subseteq F_i \qquad F_i \wedge Tr \rightarrow F'_{i+1}$$

That is, each element of the trace is safe, the trace is syntactically monotone, and each F_{i+1} is inductive relative to F_i.

Additionally, IC3 maintains a queue of *proof obligations* (or *CTI's*) of the form $\langle m, i \rangle$ where m is a cube over state variables and i is a *level*. At each point of the execution, it considers a proof obligation $\langle m, i \rangle$ with the smallest level i, and attempts to prove that m is reachable in i steps. If $i = 0$ then there is a real counterexample. Otherwise, it makes a *predecessor* query $SAT?(\neg m \wedge F_{i-1} \wedge Tr \wedge m')$ that checks whether a state in m can be reached from a state in F_{i-1}. If the result is satisfiable, it adds a predecessor of m as a new proof obligation at level $i-1$. If the result is unsatisfiable, it learns a new lemma φ, such that $Init \rightarrow \varphi$, $\varphi \rightarrow \neg m$ and $\varphi \wedge F_{i-1} \wedge Tr \rightarrow \varphi'$, and adds φ to all F_j,

Data: A cex queue Q, where $c \in Q$ is a pair $\langle m, i\rangle$, m is a cube over state variables, and $i \in \mathbb{N}$. A level N. A trace $F_0, F_1, \ldots$
Initially: $Q = \emptyset$, $N = 0$, $F_0 = \mathit{Init}$, $\forall i > 0 \cdot F_i = \top$.
repeat
- **Unreachable** If there is an $i < N$ s.t. $F_i \subseteq F_{i+1}$ **return** *Unreachable*.
- **Reachable** If there is an m s.t. $\langle m, 0\rangle \in Q$ **return** *Reachable*.
- **Unfold** If $F_N \rightarrow \neg \mathit{Bad}$, then set $N \leftarrow N + 1$, and $Q \leftarrow \emptyset$.
- **Candidate** If for some m, $m \rightarrow F_N \wedge \mathit{Bad}$, then add $\langle m, N\rangle$ to Q.
- **Predecessor** If $\langle m, i+1\rangle \in Q$ and there are m_0 and m_1 s.t. $m_1 \rightarrow m$, $m_0 \wedge m_1'$ is satisfiable, and $m_0 \wedge m_1' \rightarrow F_i \wedge \mathit{Tr} \wedge m'$, then add $\langle m_0, i\rangle$ to Q.
- **NewLemma** For $0 \leq i < N$: given $\langle m, i+1\rangle \in Q$ and a clause φ, such that $\varphi \rightarrow \neg m$, if $\mathit{Init} \rightarrow \varphi$, and $\varphi \wedge F_i \wedge \mathit{Tr} \rightarrow \varphi'$, then add φ to F_j, for $j \leq i+1$.
- **ReQueue** If $\langle m, i\rangle \in Q$, $0 < i < N$ and $F_{i-1} \wedge \mathit{Tr} \wedge m'$ is unsatisfiable, then add $\langle m, i+1\rangle$ to Q.
- **Push** For $0 \leq i < N$ and a clause $(\varphi \vee \psi) \in F_i$, if $\varphi \notin F_{i+1}$, $\mathit{Init} \rightarrow \varphi$ and $\varphi \wedge F_i \wedge \mathit{Tr} \rightarrow \varphi'$, then add φ to F_j, for each $j \leq i+1$.

until ∞;

Fig. 1. Rule-based description of IC3/PDR.

for $j \leq i$. In other words, the lemma φ represents a new over-approximation, and in particular demonstrates why the state m cannot be reached in up to i steps from the initial states. An important optimization is to re-enqueue $\langle m, i+1\rangle$ as a new proof obligation. If at any point of the execution $F_{i-1} = F_i$ and $F_i \rightarrow \neg \mathit{Bad}$, then F_i represents an inductive invariant establishing the correctness of the property.

Fig. 1 shows a rule-based overview of `IC3` (adapted from [9]). Initially, Q is empty, $N = 0$ and $F_0 = \mathit{Init}$. Then, the rules in Fig. 1 are applied (possibly in a non-deterministic order) until either **Unreachable** or **Reachable** rule is applicable. **Unfold** extends the current trace and increases the level at which counterexample is searched. **Candidate** picks a bad state. **Predecessor** extends a counterexample from the queue by one step. **NewLemma** blocks a counterexample and adds a new lemma. **ReQueue** moves the counterexample to the next level. Finally, **Push** pushes a lemma to the next level, optionally generalizing it inductively. A typical schedule of the rules is to first apply all applicable rules except for **Push** and **Unfold**, followed by **Push** at all levels, then **Unfold**, and then repeating the cycle.

III. QUIP: THE ALGORITHM

In this section, we give a high-level description of `Quip` as a set of rules. This description shows various reasoning capabilities of `Quip` and establishes its correctness. A practical implementation of these rules is described in Section IV.

The main data structures and rules for `Quip` are shown in Fig. 2. Similarly to `IC3`, `Quip` manages proof obligations using a priority queue Q. However each proof obligation is a triple $\langle m, i, t\rangle$, where m and i are as in `IC3`, and t is

Data: A cex queue Q, where $c \in Q$ is a triple $\langle m, i, t\rangle$, m is a cube over state variables, $i \in \mathbb{N}$, and $t \in \{\mathit{may}, \mathit{must}\}$. A level N. A trace $F_0, F_1, \ldots$ An invariant F_∞. A set of reachable states REACH.
Initially: $Q = \emptyset$, $N = 0$, $\text{REACH} = F_0 = \mathit{Init}$, $\forall i \geq 1 \cdot F_i = \top$, $F_\infty = \top$.
Require: $\mathit{Init} \rightarrow \neg \mathit{Bad}$
repeat
- **Unreachable** If $F_\infty \rightarrow \neg \mathit{Bad}$ **return** *Unreachable*.
- **Reachable** If $\langle m, i, \mathit{must}\rangle \in Q$, $m \cap (\vee\text{REACH}) \neq \emptyset$ **return** *Reachable*.
- **Unfold** If $F_N \rightarrow \neg \mathit{Bad}$, then set $N \leftarrow N + 1$.
- **Candidate** If for some m, $m \rightarrow F_N \wedge \mathit{Bad}$, then add $\langle m, N, \mathit{must}\rangle$ to Q.
- **Predecessor** If $\langle m, i+1, t\rangle \in Q$ and there are m_0 and m_1 s.t. $m_1 \rightarrow m$, $m_0 \wedge m_1'$ is satisfiable, and $m_0 \wedge m_1' \rightarrow F_i \wedge \mathit{Tr} \wedge m'$, then add $\langle m_0, i, t\rangle$ to Q.
- **NewLemma** For $0 \leq i < N$: given $\langle m, i+1\rangle \in Q$ and a clause φ, such that $\varphi \rightarrow \neg m$, if $(\vee\text{REACH}) \rightarrow \varphi$, and $\varphi \wedge F_i \wedge \mathit{Tr} \rightarrow \varphi'$, then add φ to F_j, for $j \leq i+1$.
- **ReQueue** If $\langle m, i, \mathit{must}\rangle \in Q$, and $F_{i-1} \wedge \mathit{Tr} \wedge m'$ is unsatisfiable, then add $\langle m, i+1, \mathit{must}\rangle$ to Q.
- **Push** For $1 \leq i$ and a clause $(\varphi \vee \psi) \in F_i \setminus F_{i+1}$, if $(\vee\text{REACH}) \rightarrow \varphi$ and $\varphi \wedge F_i \wedge \mathit{Tr} \rightarrow \varphi'$, then add φ to F_j, for each $j \leq i+1$.
- **MaxIndSubset** If there is $i > N$ s.t. $F_{i+1} \subseteq F_i$, then $F_\infty \leftarrow F_i$, and $\forall j \geq i \cdot F_j \leftarrow F_\infty$.
- **Successor** If $\langle m, i+1, t\rangle \in Q$ and exist m_0, m_1 s.t. $m_0 \wedge m_1'$ are satisfiable and $m_0 \wedge m_1' \rightarrow (\vee\text{REACH}) \wedge \mathit{Tr} \wedge m'$, then add m_1 to REACH.
- **MayEnqueue** For $i \geq 1$ and a clause $\varphi \in F_i \setminus F_{i+1}$, if $(\vee\text{REACH}) \rightarrow \varphi$, add $\langle \neg\varphi, i+1, \mathit{may}\rangle \in Q$.
- **ResetQ** $Q \leftarrow \emptyset$.
- **ResetReach** $\text{REACH} \leftarrow \mathit{Init}$.

until ∞;

Fig. 2. Rule-based description of Quip.

the type of the proof-obligation: either *may* or *must*. Must proof-obligations represent cubes that must be blocked for the problem to be SAFE. May-proof-obligations represent cubes that we would like to block, but the problem might be SAFE even if they are not blocked. As in `IC3`, `Quip` maintains a trace of clauses $F_0, F_1, \ldots$. However, the number of the non-empty frames in the trace can be larger than the current depth N. Intuitively, a non-empty frame F_i with $i > N$ contains clauses that are inductive up to a yet-to-be-explored level i. Additionally, as in `PDR`, `Quip` maintains a set F_∞ of absolute invariants. The unique feature of `Quip` is that it also maintains a set REACH of states reachable from Init. In practice, we keep REACH as a set of cubes. We say that a lemma $\varphi \in F_i$ is *good* if it is also in F_∞, *bad* if it excludes a state in REACH, and *unknown* otherwise. Note that the categories above are exclusive – a lemma cannot be both *good* and *bad* at the same time.

We now describe the rules.

a) Termination: The rule **Unreachable** in `Quip` is even simpler than the corresponding one in `IC3`: the verification problem is deduced to be SAFE as soon as $F_\infty \Rightarrow \neg Bad$. Note that this formulation makes it extremely easy to handle designs with multiple properties. The rule **Reachable** in `Quip` states that the problem is UNSAFE if a must-proof-obligation includes a reachable state; that is, either an initial state or a new reachable state explicitly found by the algorithm.

b) Generating proof obligations: The rules **Candidate**, **Predecessor**, and **ReQueue** are similar to the corresponding rules of `IC3`. The rule **MayEnqueue** is new. **Candidate** picks a bad state and adds it as a must-proof-obligation. **Predecessor** adds a CTI m_0 for an already existing proof obligation m_1 as a new proof obligation, at the level one lower than that of m_1. The type of m_0 is the same as that of m_1, and so in particular m_0 is a must-proof-obligation whenever m_1 is. **ReQueue** moves a blocked must-proof-obligation to the next level. We explicitly limit this rule to must-proof-obligations only, as may-proof-obligations are handled by **MayEnqueue**. **MayEnqueue** picks a lemma $\varphi \in F_i \setminus F_{i+1}$ that is not yet established at level $i+1$ and adds its negation $\neg\varphi$ as a may-proof-obligation at level $i+1$. The rule is only applicable if the status of φ is *unknown*. Note that it is actually sound to take *any* clause ψ such that $Init \Rightarrow \psi$ and *any* level k, and add $\neg\psi$ at level k as a may-proof-obligation. However, we do not currently use this level of generality.

c) Managing lemmas: **Unfold** increases the level at which a counterexample is searched. **NewLemma** adds a new lemma that blocks a proof obligation. We explicitly disallow learning *bad* lemmas. For correctness, it is possible to take any clause ψ such that $\psi \wedge F_i \wedge Tr \rightarrow \psi'$ and add ψ to all F_j for $j \leq i+1$. **Push** pushes a lemma to the next level, optionally generalizing it inductively. As before, we limit pushing and generalization to *unknown* lemmas only. An important distinction from `IC3` is that in `Quip` **Push** is not limited to the current working depth N of the algorithm.

d) Inductive invariant: **MaxIndSubset** checks whether for some i there is $F_i = F_{i+1}$. In this case, F_i is an inductive invariant which is used to enlarge F_∞. In the case $i < N$, F_∞ is a safe inductive invariant and an immediate application of **Unreachable** finishes verification. Otherwise, it discovers new *good* lemmas. Correctness follows from the fact that $F_i = F_{i+1}$ indirectly implies that $\forall j \geq i \cdot F_j \cap \text{REACH} = \emptyset$. That is, there are no *bad* lemmas in any F_j for $j \geq i$. Note that a maximal inductive subset of current lemmas is computed by applying **Push** as much as possible, followed by **MaxIndSubset**.

e) Reachability: **Successor** adds new reachable states. Given a proof obligation m that can be reached in one transition from an already known reachable state (either an initial state or an explicitly found reachable state), it computes a new reachable state m_1 that is included in m and adds it to REACH.

f) Restarts: The final set of rules deals with various reset mechanisms. The rule **ResetQ** allows to empty the proof obligation queue. This rule can be though of as a "local reset" that may guide `Quip` in a different search place by examining different predecessors and learning new lemmas. Note that in `IC3`, **ResetQ** is implicitly included in **Unfold**. That is, `IC3` resets its queue every time a new depth is explored. On the other hand, in `Quip` this choice is flexible. The rule **ResetReach** resets the reachable states. In practice, we may remove only some (less useful) reachable states when their number becomes too large.

$F_0 = Init$
if $F_0 \wedge Bad$ **then**
  return CEX
$N \leftarrow 0$; $F_\infty \leftarrow \top$; REACH $= F_0$
while *(true)* **do**
  $N \leftarrow N + 1$
  if `Quip_RecBlockCube`$(Bad, N) =$ CEX **then**
    return CEX
  if `Quip_Push`$() =$ PROOF **then**
    return PROOF

Fig. 3. Main Procedure (`Quip_Main`).

IV. QUIP: IMPLEMENTATION

In this section, we describe our implementation of the `Quip` rules.

The set of all reachable states handled by `Quip` is of the form REACH $= Init \cup R$, where $Init$ are the initial states and R are the reachable states dynamically discovered by the algorithm. In our current implementation, R consists only of *concrete* states. That is, each element of R is a complete assignment to *all* state variables. Each state in R is stored as a Boolean array. The main functionality required from R is checking whether a given cube s intersects (or equivalently subsumes) one of the states r in R. In the pseudocode below, the function `Intersect`(R, s) returns NULL if $R \cap s = \emptyset$, and returns a state $r \in R$ with $r \cap s \neq \emptyset$ otherwise.

In what follows, we require an additional bookkeeping mechanism. If a proof obligation $\langle s, f, p\rangle$ is added as a predecessor of another proof obligation $\langle \tilde{s}, \tilde{f}, \tilde{p}\rangle$ using the **Predecessor** rule, then we say that $\tilde{s}$ is a *parent* of s. On the other hand, if $\langle s, f, p\rangle$ is added using either **Candidate** or **MayEnqueue**, then we say that s has no parent. Finally, the rule **ReQueue** keeps the parent information. In the pseudocode, we let `Parent`(s) be the parent of s or NULL if none. To some extent this bookkeeping is already supported by most `IC3` implementations as it is required for reconstructing counterexamples.

A. The Main Loop

Our implementation of `Quip` is structured similarly to `PDR` [2]. For completeness, the main loop is shown in Fig. 3. The algorithm first checks for a counterexamples at level 0 ($N = 0$), and then incrementally increases the working level N until either a counterexample or a safe inductive invariant is found.

B. Recursive Block Cube

The central procedure, `Quip_RecBlockCube`, that recursively blocks a bad state, is shown in Fig. 4. On the surface, it looks similar to `Pdr_RecursiveBlockCube` from [2], but there are many important differences.

```
   Input: (Cube s_0, Frame f_0)
   Data: Priority queue Q of triples ⟨c, f, t⟩, where c is a
         cube, f is a level and t ∈ {may, must}
   Data: Map Parent from a proof obligation to its
         parent proof obligation (NULL if none)
   Data: Array R containing concrete reachable states
 1 Add(Q, ⟨s_0, f_0, must⟩)
 2 Parent(s_0) ← NULL
 3 while ¬Empty(Q) do
 4     ⟨s, f, p⟩ ← Pop(Q)
 5     if f = 0 then
 6         if p = must then
               // Found Real Counterexample
 7             return CEX
 8         else
               // New reachable state
 9             Find r such that Init ∧ Tr → r' and
               r ∩ Parent(s) ≠ ∅; Add r to R
10             continue
11     if (r_0 ← Intersect(R, s)) ≠ NULL then
12         if p = must then
               // Found Real Counterexample
13             return CEX
14         else
15             if Parent(s) ≠ NULL then
                   // New reachable state
16                 Find r such that r_0 ∧ Tr → r' and
                   r' ∩ Parent(s) ≠ ∅; Add r to R
17             continue
18     ⟨t, g⟩ ← Block(s, f)
19     if g ≠ f − 1 then
           // Cube s is successfully blocked
              by lemma ¬t
           // Lemma ¬t holds until frame g
20         if (g < N) then
21             if t ≠ s then
22                 Add(Q, ⟨t, g + 1, may⟩)
23                 Parent(t) ← NULL
24             else
25                 Add(Q, ⟨t, g + 1, p⟩)
26     else
           // t is a predecessor of s
27         Add(Q, ⟨t, f − 1, p⟩)
28         Add(Q, ⟨s, f, p⟩)
29         Parent(t) ← s
30 return BLOCKED
```

Fig. 4. Recursive Block Cube (`Quip_RecBlockCube`).

`Quip_RecBlockCube` accepts a must-proof-obligation $\langle s_0, f_0, must\rangle$, and either succeeds to strengthen the trace so that s_0 is blocked at level f_0, or finds a concrete reachable state r that intersects s_0 (hence r is a witness that $\neg s_0$ is not an invariant).

`Quip_RecBlockCube` starts by adding the proof-obligation $\langle s_0, f_0, must\rangle$, with no parent, to Q (lines 1–2) and proceeds to the main loop. In each iteration of the loop, it retrieves the proof-obligation from Q with the lowest-level, and in case of a tie, with the smaller number of literals. In particular, the proposed tie-breaking condition means that when Q contains two proof-obligations s_1 and s_2 at the lowest level, with $s_1 \subseteq s_2$, the algorithm will select s_1 first – hence attempting to derive the strongest possible lemma (that would automatically block s_2 as well). Let $\langle s, f, p\rangle$ be this proof obligation (line 4).

Let us first assume that the level f of the proof obligation is 0 (lines 5–10). In particular, $s \cap Init \neq \emptyset$ and $\texttt{Parent}(s) \neq$ NULL (according to our rules, only **Predecessor** can add proof-obligations at level 0). If this is a $must$-proof-obligation (lines 6–7), then our property is deduced to be UNSAFE and `Quip_RecBlockCube` terminates. Moreover, a concrete counterexample can be reconstructed using the parent information. If this is a may-proof-obligation (lines 8–10), then we compute a new reachable state r that is one-step reachable from $Init$ and that intersects $\texttt{Parent}(s)$. Note that such a state r must always exist since s is a CTI for $\texttt{Parent}(s)$. In our implementation, we use a dedicated SAT-solver for all the successor queries, including reconstruction of real counterexamples. However, by also saving for each predecessor the assignment to inputs, this task can be reduced to simulation. The new state r is then added to R. In particular, when on some future iteration the algorithm returns to examining the proof-obligation corresponding to $\texttt{Parent}(s)$, $\texttt{Parent}(s)$ already intersects R.

Next, let us assume that s intersects a state $r_0 \in R$ (lines 11–17). If this is a must-proof-obligation, then our property is deduced to be UNSAFE and the procedure terminates. By additionally storing for each state in R its predecessor (not explicitly shown in the pseudocode), we can again reconstruct a real counterexample. If this is a may-proof-obligation and $Parent(s) \neq$ NULL, then as before we compute a reachable state r that is one-step reachable from r_0 and that intersects $\texttt{Parent}(s)$ – and so when the algorithm returns to examining $\texttt{Parent}(s)$ the condition $\texttt{Intersect}(R, \texttt{Parent}(s)) \neq \emptyset$ is activated and the reachable state is further propagated. In other words, as soon as a recursive predecessor of a may-proof-obligation intersects an initial or an already existing reachable state in R, a sequence of additional reachable states is discovered, including a reachable state that intersects a given proof-obligation.

The helper procedure `Block` (line 18), adapted from `PDR` [2], hides some less relevant details. In our implementation, $Block(s, f)$ first *syntactically* checks whether s is already blocked in the frame f – i.e., whether there exists a lemma $\neg t \in F_g$ with $t \subseteq s$ and $f \leq g$ (the case $g = \infty$ is also allowed). If so, then (t, g) is returned. Otherwise, $Block(s, f)$ checks whether the formula $F_{f-1} \wedge Tr \wedge s'$ is satisfiable. If it is, a predecessor t of s is extracted and suitably generalized. In this case, $(t, f-1)$ is returned. If the formula is unsatisfiable, then using an inductive generalization procedure, we obtain a lemma $\neg t$ which holds at least up to the frame f (and possibly up to a larger frame g, including ∞). In this case, $Block$ adds the lemma $\neg t$ to F_g and returns (t, g). Note that lemma generalization takes the reachable states R into account, and ensures that new lemmas always include all of R.

Let us first consider the case that the cube s was successfully blocked (lines 20–25), i.e., `Block` returns a lemma $\neg t \in F_g$ with $t \subseteq s$ and $f \leq g$. An important optimization in `IC3` consists of reinserting the proof-obligation s at the level $g+1$, forcing the algorithm to block s in all higher frames as well. The unique feature of `Quip` is that $\neg t$ is inserted into

```
   for k = 1, ... do
   |  for all lemmas c ∈ F_k \ F_{k+1} do
   |  |  // Rule Push
1  |  |  if ¬bad(c) then
2  |  |  |  if F_k ∧ c ∧ Tr ⇒ c' then
3  |  |  |  |  F_{k+1} ← F_{k+1} ∪ {c}
   |  if F_k \ F_{k+1} = ∅ then
   |  |  // Rule MaxIndSubset
4  |  |  F_∞ ← F_k
5  |  |  for j = k + 1, ... do
6  |  |  |  F_j ← F_∞
7  |  |  break;
   if F_∞ ⇒ ¬Bad then
   |  // Found Safe Inductive Invariant
8  |  return PROOF
   return UNKNOWN
```

Fig. 5. Pushing lemmas (`Quip_Push`).

Q at the level $g+1$ instead of s. This forces the algorithm to concentrate on further pushing existing lemma t rather than discovering new lemmas to block s at a higher level. However, $\neg t$ can be only added as a may-obligation (with the only exception being that $s = t$ and s is a must-obligation). Finally, note that when $t \neq s$, the cube t has no parent, otherwise we keep the previous parent of s.

In the case that a predecessor t of s is found (lines 16–29), just as in `IC3`, `Quip` returns $\langle s, f, p\rangle$ to Q, as well as inserts a new proof obligation $\langle t, f-1, p\rangle$ with the same type of a proof obligation as that of s. The parent of t is set to s.

C. Pushing

Fig. 5 describes our pushing procedure `Quip_Push`. For each lemma c, we keep a Boolean flag $bad(c)$ that represents whether c is known to be *bad* (that is, whether c excludes some states in REACH). We say that a lemma is *unknown* if $bad(c) =$ FALSE and $c \notin F_\infty$. Each time that a new reachable state r is added in `Quip_RecBlockCube`, we check it against every *unknown* lemma in the system and mark as bad those lemmas that exclude r. Just as in `IC3`, in practice the sets F_i are delta-encoded: for any i, j, $F_i \cap F_j = \emptyset$. However, for this presentation, we are using the full sets F_i as defined in the introduction. The pushing stage proceeds as in `IC3`, with the following exceptions. First, *bad* lemmas are not pushed. This has two positive effects. The primary effect is conserving resources by not propagating lemmas that have no chance to be in the final invariant. A secondary effect is that as the new lemmas are learned, they are less dependent on the currently known bad lemmas. Second, the lemmas are pushed arbitrarily far past the current depth N. In particular, in the last iteration of the outer *for*-loop, all lemmas at level k are pushed to the next frame. In this case, the *if*-condition on line 3 is true, and all lemmas of F_{k+1} are added to F_∞. It is easy to see that after `Quip_Push`, F_∞ contains the maximal inductive subset of all lemmas in the system. If F_∞ implies $\neg Bad$, i.e., F_∞ represents a safe inductive invariant, then `Quip_Push` returns PROOF.

D. Managing reachable states

Efficiently handling reachable states poses additional challenges. Currently we represent reachable states explicitly, and as their number grows large, the time taken by `Intersect` and the memory required for their storage become significant. However, our experience shows that many of the reachable states can be removed without much sacrificing the number of may-proof-obligations pruned or the quality of lemmas discovered, and that the newly discovered states are more likely to be useful in the immediate future. Thus our solution mimics the clause deletion strategy as used in a SAT solver: for each reachable state we keep its *activity* representing how many times the state was a witness for intersection, and we periodically decay this activity and aggressively delete the less active states. Furthermore, as in our current implementation most of the time on managing lemmas is spent during the inductive generalization (making sure that a learned lemma includes all the states in REACH), we have found it further beneficial to consider even fewer reachable states during the generalization.

It may also be possible to compute partial states directly from the **Successor** query, or to represent reachable states symbolically by computing minimal DNF representation of R. An alternative way to take reachable states into account is to include them directly in F_0. Another optimization is to check whether a given may-obligation is one-step reachable from R. However, we have found both of these difficult to implement efficiently. Finally, it might also be useful to push reachable states forward more aggressively, for example, by running a simulation from already known reachable states.

V. ALTERNATIVES

In this section, we present two alternative implementations of `Quip`, which illustrate the variety of possibilities offered by our framework. Unfortunately, for the reasons discussed below, both of these variants do not perform consistently. We sketch how they could be improved in the future.

A. Reset-free approach

Both `IC3` and `Quip` as described previously implicitly reset the queue of proof obligations each time that a new depth is explored. An interesting alternative in `Quip` is as follows. (1) Allow to enqueue proof-obligations at any level (and not only up to N) by removing the *if*-condition on line 20 of `Quip_RecBlockCube`. (2) Check whether $F_k = F_{k+1}$ each time that a lemma is successfully pushed from F_k to a higher-frame (or simply each time that a proof obligation at level $k+1$ is successfully blocked); if $F_k = F_{k+1}$, then grow the set F_∞ to F_k, and check the termination condition $F_\infty \Rightarrow \neg Bad$. (3) Replace `Quip_Main` by a single call to `Quip_RecBlockCube`$(Bad, 1)$. In this way, the negation of *every* unknown lemma in the system is always present as a proof obligation at the corresponding frame and the external pushing stage can be avoided altogether. This alternative procedure takes to the extreme the idea of pushing every lemma in the system as far as possible, and arguably results in an even simpler overall algorithm. However, a preliminary experimental evaluation shows that this scheme performs worse in practice. One possible explanation is that

TABLE I. SUMMARY OF EXPERIMENTAL RESULTS

	UNSAFE solved	UNSAFE time	SAFE solved	SAFE time
IC3	22 (2)	52,302	76 (7)	137,244
Quip	32 (12)	20,302	99 (30)	69,590

Experimental results on the instances solved by either IC3 or Quip separated into unsafe and safe instances. The numbers in parentheses represent the unique solves. The times are in seconds.

periodically resetting the proof obligation queue keeps proof obligations more focused to proving the property, while the procedure above handles the "main" lemmas and the "supporting" lemmas (and the supporting lemmas for the supporting lemmas, and so on) equally. A possible solution would be to define some additional criteria for proof obligations reflecting their expected usefulness, and to take these into account when choosing the next proof obligation.

B. Garbage-collecting bad lemmas

We can use the classification of all lemmas into *good*, *bad* and *unknown* to periodically remove all the bad lemmas from the system. However, as bad lemmas may be supporting other unknown lemmas, we cannot simply remove *bad* lemmas from their corresponding frames. Instead, we can keep all the *good* lemmas in F_∞, put all the *unknown* lemmas into F_1, and use **Push** to push the *unknown* lemmas as far as possible. We have found that during this pushing stage it is important to preserve the set of lemmas as much as possible, which requires to disable both the additional generalizing capability of pushing and the built-in subsumption mechanism for storing lemmas. Note that we might also need to decrease the current bound N at which the property is proved. A preliminary experimental evaluation shows that this variant usually allows Quip to converge at a smaller depth, and in some cases leads to a significant speedup. However, it is also true that applying "garbage collection" too aggressively on average leads to a significant performance degradation, and an ongoing work is to find the a good heuristic for when to apply it and how to properly combine it with the resetting of reachable states described in Section IV.

VI. EXPERIMENTS

In this section, we present our experimental results[2]. We compare Quip with a custom variant of IC3, as implemented in the IBM formal verification tool *Rulebase-Sixthsense* [10]. All experiments were performed on a 2.13Ghz Linux-based machine with Intel Xeon E7-4830 processor, 16GB of RAM, and one hour time limit. We have used 300 single property designs from the HWMCC'13 and HWMCC'14 benchmark sets. These are obtained by removing duplicates and instances solved using standard logic synthesis (similar to the &dc2 command in ABC [11]).

The overall results are shown in Table I. The columns "UNSAFE solved" and "SAFE solved" show that number of unsafe and safe instances, respectively, solved by either IC3 or Quip. The numbers in parentheses represent the number of instances not solved by the other configuration. The columns "UNSAFE time" and "SAFE time" represent the cumulative time in seconds for unsafe and safe properties, respectively.

[2]See http://arieg.bitbucket.org/quip for more details.

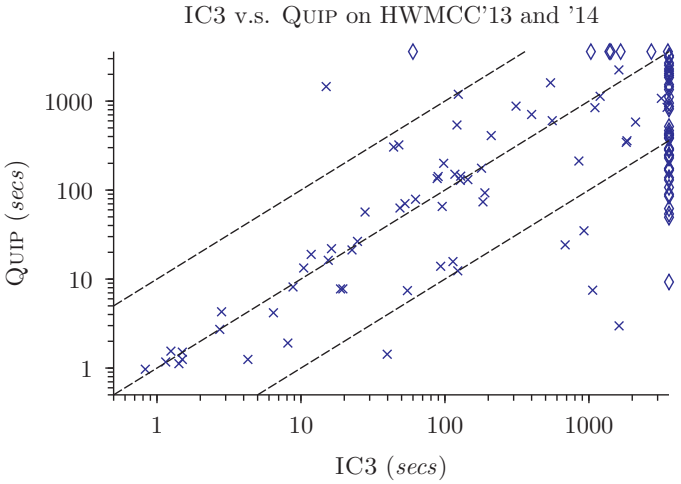

Fig. 6. Run-time comparison between IC3 and Quip. Points below the diagonal are in favor of Quip. The scale is logarithmic. Diagonals mark an order of magnitude. Timeout is 3,600 seconds.

TABLE II. DATA ON REACHABLE STATES DISCOVERED BY QUIP

# reach. states	0–10	11 – 100	101 – 1K	1K – 10K	10K – 50K
# instances	42	19	29	32	9
# unique solved	1	1	10	22	8

According to our experiments, either IC3 or Quip was successful on 34 unsafe instances and 106 safe instances. In the remaining 160 instances both IC3 and Quip timed out. We can see that Quip is clearly superior to IC3 on both safe and unsafe problems, solving more properties and running in roughly half of the time.

A more detailed comparison between IC3 and Quip is shown on the scatter plot in Fig. 6. Only the 140 instances solved by at least one tool are shown. For instances solved by both, the run time is similar, with an advantage for Quip Sometimes, the advantage is over an order of magnitude. Quip shines on harder instances and is able to solve significantly more of them than IC3.

Finally, we give some intuition on the total number of reachable states typically discovered by Quip and whether these states are useful for verification. Table II contains the data for the 131 instances solved by Quip, including the 42 instances not solved by IC3. In the table, the row "#reach. states" represents a range, the row "#instances" specifies the number of instances solved by Quip with the total number of reachable states in this range, and the row "#unique solves" further specifies the number of instances solved uniquely by Quip. For example, the third column means that 29 instances solved by Quip required between 101 and 10,000 reachable states, and 10 out of 29 are not solved by IC3. We draw two conclusions. First, even though we use concrete reachable states (i.e., complete assignment to all state variables), relatively few states had to be discovered. Second, the advantage of Quip over IC3 is especially pronounced as the number of learned reachable states increases. For example, from the 61 instances where Quip required less than 100 reachable states, only 2 are not solved by IC3. However, from the set of 9 instances where Quip finds more than $10,001$ reachable states, 8 (i.e., all but 1) are not solved by IC3.

VII. Related Work

Computing Maximal Inductive Subset (MIS) is a well-known problem in both hardware and software verification (e.g., [12], [13]). Applying MIS to enlarge F_∞ in IC3/PDR is already suggested in [2], but it was not effective since the cost of computing an MIS out-weighted the gains. In Quip, the MIS computation is amortized by not limiting Quip/**Push** rule to the current bound N (for comparison, see IC3/**Push** in Fig. 1) and by discovering MIS opportunistically using Quip/**MaxIndSubset**. Thus, even if the MIS computation is unsuccessful and no new lemmas are added to F_∞, the trace is strengthened for the future runs of the algorithm. In our experience, extending IC3 in this way is beneficial regardless of the other Quip rules.

Blocking states that are not necessarily backward reachable from an error state and separating proof obligations into *may* and *must* was proposed in the context of IC3-based abstraction refinement [4]. The idea is also implicitly present in computation of minimal inductive clauses [3] and predicate-abstraction-based extensions of IC3 to software [14], [15]. In contrast to the above algorithms, Quip seamlessly integrates *must* and *may* reasoning into one algorithmic procedure without any specialized refinement steps. More significantly, Quip uses the reachable states that witness a failure of a *may*-proof obligation to improve future lemma generalization. Thus, both proving and *disproving* a may-proof-obligation is beneficial to the overall algorithm.

Extracting forward reachable states from spurious counterexamples also appears in NewItp [16] as *states to refinement* in the context of interpolation-based model checking. Similar to Quip, these states are used to guide future interpolants to avoid reachable states. In essence, Quip computes both an over-approximation (lemmas) and under-approximation (Reach) of reachable states. This can be seen as an extension of over- and under-approximations used in Spacer [6] from modular to monolithic-proofs. The key difference is that, Spacer under-approximates summaries of procedures and not states reachable from an initial state.

Interestingly, CTI's of Reverse IC3 [17] – a dual variant of IC3 that recursively enumerates states reachable from $Init$ and that learns an over-approximation of states backwards reachable from Bad – are forward reachable states. Thus, it might be possible to combine IC3 and Reverse IC3 into an algorithm that computes both forward and backward reachable states and their over-approximations, somewhat akin to DAR [18]. Although DAR is restricted only to over-approximations.

VIII. Conclusions

In this paper, we present an improvement to the core of the IC3 algorithm. We propose an approach, called Quip, that is designed to propagate learned lemmas more aggressively, and whose implementation seamlessly integrates must and may proof-obligations and forward reachable states. The experimental results show that a naïve implementation of Quip significantly outperforms a highly-tuned implementation of IC3/PDR. We believe that the new reasoning capabilities introduced in Quip open up many opportunities for further improvements to SAT-based automated verification.

References

[1] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.

[2] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[3] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 157–164.

[4] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and sat-based reachability in hardware model checking," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, 2012, pp. 173–181.

[5] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, pp. 157–171.

[6] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 17–34.

[7] A. Cimatti and A. Griggio, "Software model checking via IC3," in *CAV*, 2012, pp. 277–293.

[8] A. Griggio and M. Roveri, "Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking," in *Proceedings of International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS'14)*, Lausanne, Switzerland, October 2014.

[9] N. Bjørner and A. Gurfinkel, "Property directed polyhedral abstraction," in *VMCAI*, 2015, pp. 263–281.

[10] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, 2004, pp. 159–173.

[11] R. K. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *CAV*, 2010, pp. 24–40.

[12] C. A. J. van Eijk, "Sequential Equivalence Checking without State Space Traversal," in *1998 Design, Automation and Test in Europe (DATE '98), February 23-26, 1998, Le Palais des Congrès de Paris, Paris, France*, 1998, pp. 618–623.

[13] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, 2001, pp. 500–517.

[14] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *TACAS*, 2014, pp. 46–61.

[15] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, "Counterexample to induction-guided abstraction-refinement (CTIGAR)," in *CAV*, 2014, pp. 831–848.

[16] C. Wu, C. Wu, C. Lai, and C. R. Haung, "A counterexample-guided interpolant generation algorithm for sat-based model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1846–1858, 2014.

[17] F. Somenzi and A. R. Bradley, "IC3: where monolithic and incremental meet," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, 2011, pp. 3–8.

[18] Y. Vizel, O. Grumberg, and S. Shoham, "Intertwined forward-backward reachability analysis using interpolants," in *TACAS*, 2013, pp. 308–323.