Learning Support Sets in IC3 and Quip: the Good, the Bad, and the Ugly

Ryan Berryhill
Dept. of ECE
University of Toronto
Toronto, Canada
ryan@eecg.toronto.edu

Alexander Ivrii
IBM Research
Haifa, Israel
alexi@il.ibm.com

Neil Veira
Dept. of ECE
University of Toronto
Toronto, Canada
neil.veira@mail.utoronto.ca

Andreas Veneris
Depts. of ECE and CS
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

***Abstract*—In recent years, IC3 has enjoyed wide adoption by academia and industry as an unbounded model checking engine. The core algorithm works by learning lemmas that, given a safe property, eventually converge to an inductive proof. As such, its runtime performance is heavily dependent upon "pushing" (or "promoting") important lemmas, possibly by discovering additional supporting lemmas. More recently, Quip has emerged to be a complementary extension behind the reasoning capabilities of IC3 as it allows it to target particular lemmas for pushing. This also raises the following question: which lemmas should be promoted? To that end, this paper extends the reasoning capabilities of IC3 and Quip using special SAT queries to find *support sets* that represent fine-grained information on which lemmas are required to push other lemmas. Further, this paper presents an IC3-based algorithm called Truss (Testing Reachability Using Support Sets) that uses support sets to identify sets of lemmas that may be close to forming an inductive proof. The set is targeted for promotion as a cohesive unit. If any of the lemmas cannot be promoted, the entire set is abandoned and a new set excluding that lemma is found. In the presented framework, there are two reasons why a lemma cannot be promoted: either because it blocks a known reachable state (in which case, the lemma is permanently marked as *bad*), or because lemma promotion exceeds a specified amount of effort (in which case the lemma is temporarily marked as *ugly*). Intuitively, the proposed approach allows the algorithm to construct a proof more quickly by focusing on the important yet easily-pushed lemmas. Experiments on the HWMCC'15 benchmark set show a significant improvement against existing practices. Compared to Quip, our algorithm solves 17 more problem instances and it offers an impressive 1.77x speedup.**

I. Introduction

Formal verification remains one of the fastest growing segments in verification [1]. Unbounded model checking, which determines if particular states are reachable in a circuit, is a problem of fundamental importance in this area. `IC3` [2], [3] has established itself as a state-of-the-art unbounded model checker and has seen wide adoption in industry [4]. More recently, the closely related technique of `Quip` [5] has emerged with better run time performance and greater reasoning capabilities. Any improvements to `IC3` or `Quip` can therefore have wide-reaching impact in formal verification.

Both `Quip` and `IC3` aim to construct an inductive invariant proving a given safety property. They share similar core functionality. A Boolean Satisfiability (SAT)-based procedure is used to detect states that can reach a property violation, which are referred to as counter-examples-to-induction (CTIs). When a CTI is detected, the algorithm tries to learn a lemma that explains why the state is not reachable in a bounded number of steps called the lemma's *level*. Through this procedure, the algorithm learns and refines a sequence of over-approximations of the states reachable at each level. This sequence is known as the *inductive trace*, and the approximation at each level is called a *frame*. An additional pushing step promotes lemmas from one level to the next if their current frame is strong enough. The run time performance of these algorithms is dependent on their ability to learn and promote relevant lemmas.

This paper presents an algorithm called `Truss` (Testing Reachability Using Support Sets) that leverages the features of modern incremental SAT solvers to compute lemma supports and the reasoning capabilities of `Quip` to encourage the promotion of highly relevant lemmas that are easy to promote. In a nutshell, special SAT queries are used to identify a set of lemmas that participate in a bounded proof of the property, and thus may potentially appear in a safe inductive invariant. This set is targeted for promotion as a cohesive unit. If any lemma in the set fails to promote, the entire set is abandoned and one or more lemmas are temporarily or permanently blacklisted from appearing in future sets. A new set is found and targeted for promotion. This process repeats until it can be determined that no such set is available, at which point the algorithm falls back to the usual recursive blocking approach used in `Quip`.

More specifically, a special SAT query identifies a set of lemmas sufficient to support the promotion of a particular lemma or the property itself, called its *support set* [6]. The query is similar to the one normally used to check for relative induction, but requires the addition of a unique activation literal for each lemma. When the property is being promoted to a new level, a support set is computed from the frame one level below the property. If one is found, the algorithm attempts to promote the supporting lemmas first. This is accomplished by recursively computing their support sets, and attempting to promote those lemmas, and so on. Lemmas at the bottom level do not have support sets; they are promoted using the usual recursive blocking method.

When supporting lemmas cannot be promoted, the algorithm is allowed to expend a configurable amount of effort to promote the lemma with recursive blocking. If a lemma still cannot be promoted, it is marked as ugly and is temporarily blacklisted from appearing in support sets. As is the case in `Quip` this process can reveal reachable states. When a lemma blocks a known reachable state, it is marked as bad. Bad lemmas are permanently blacklisted, as they can never appear in an inductive invariant. Similarly to Quip, absolute invariant lemmas are classified as good. Not all lemmas are classified, some are left in the category of unknown. Ugly lemmas also fit into the unknown class as it isn't known whether they are good or bad. Hence the algorithm partitions lemmas into classes containing the unknown, the good, the bad, and the ugly lemmas. This contrasts with the work of [6], where support sets are used to compute heuristic functions for each lemma. A special step after pushing attempts to learn new lemmas in order to promote those that have high heuristic values. Effort limits are not used and the heuristics require support sets for every lemma, making the approach presented in that work computationally expensive.

Experiments on HWMCC'15 circuits demonstrate the effectiveness of Truss. Compared to Quip, it offers a 1.77x speedup on the HWMCC'15 benchmark set and solves 17 more problem instances. Further, while Truss processes a similar number of must-proof obligations, it processes only one third as many may-proof obligations as Quip. This comes with the cost of additional overhead to compute support sets, which is found to represent less than 5% of the total runtime.

The rest of this paper is organized as follows. Section II presents notation and background material on unbounded model checking. Section III presents the technique used to compute support sets. Section IV presents the main algorithm. Section V presents an empirical evaluation of the approach. Section VI examines alternative approaches and related work. Finally, section VII concludes the paper.

II. Preliminaries

A. Notation

The following terminology and notation is used throughout this paper. A literal is either a variable or its negation. A cube is a conjunction of literals. A clause is a disjunction of literals. A Boolean formula in Conjunctive Normal Form (CNF) is a conjunction of clauses. A clause or a cube can be treated as a set of literals, and a CNF formula as a set of clauses. For a CNF formula F, $c \in F$ means that the clause c appears in F. Similarly $l \in c$ means that the literal l occurs in c.

Consider a finite transition system, and let $\mathcal{V}$ be the state variables of the system. The primed versions $\mathcal{V}' = \{v' | v \in \mathcal{V}\}$ represent the next-state functions. That is, for each $v \in \mathcal{V}$, v' is a binary function of the current state and input defining the next state for v. For any formula F over $\mathcal{V}$, the primed version F' represents the same formula with each free variable $v \in \mathcal{V}$ replaced by v'. A model checking problem is a tuple $P = (Init, \mathcal{T}, Bad)$ where $Init(\mathcal{V})$ and $Bad(\mathcal{V})$ are CNF formulas over $\mathcal{V}$ representing the initial states and the unsafe states, respectively. States that are not unsafe are called safe states. The transition relation $\mathcal{T}(\mathcal{V}, \mathcal{V}')$ is a CNF formula over $\mathcal{V} \cup \mathcal{V}'$. It is encoded such that $\mathcal{T}(\vec{v}, \vec{v}\,')$ is satisfiable iff state $\vec{v}$ can transition to state $\vec{v}\,'$. States are called *i-step reachable* if they can be reached in i or fewer steps from an initial state under $\mathcal{T}$. States that are i-step reachable for some value of i are *reachable*.

For any formula F over $\mathcal{V}$, a state $\vec{v}$ that satisfies F (*i.e.,* $F(\vec{v}) = 1$) is called an F-state. Given two formulas $F(\mathcal{V})$ and $G(\mathcal{V})$, F is inductive relative to G if:

$$G(\vec{v}) \wedge F(\vec{v}) \wedge \mathcal{T}(\vec{v}, \vec{v}\,') \Rightarrow F(\vec{v}\,')$$

If F is inductive relative to itself, it is simply *inductive*.

A problem instance P is UNSAFE iff there exists a natural number N such that the following formula is SAT:

$$Init(\vec{v_0}) \wedge \Big(\bigwedge_{i=0}^{N-1} \mathcal{T}(\vec{v_i}, \vec{v_{i+1}}) \Big) \wedge Bad(\vec{v_N}) \tag{1}$$

A problem instance P is SAFE iff there exists an inductive formula $Inv(\mathcal{V})$ that also meets the following conditions:

$$Init(\vec{v}) \Rightarrow Inv(\vec{v}\,') \tag{2}$$

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

A formula satisfying Eq. 2 satisfies *initiation*, meaning that it contains all initial states. An inductive formula that satisfies initiation contains all reachable states and is called an *inductive invariant*. A formula satisfying Eq. 3 is *safe*, meaning that it represents a superset of the safe states. A safe inductive invariant represents a superset of the reachable states and a subset of the safe states. Each of these properties can be checked using a single query to a SAT solver, so a safe inductive invariant is a proof that P is SAFE.

B. Overview of Quip

This section gives an overview of Quip [5], which is itself based on IC3 [2], [3]. Given an unbounded model checking problem, it either returns an inductive invariant proving the property or a counter-example trace that reaches an unsafe state. It works by maintaining a sequence of CNF formulas $F_0, F_1, ...$ called the *inductive trace*. Each F_i is a *frame*, and its index i is called its *level*. Each clause $c \in F_i$ is called a *lemma*. The frame F_0 is identical to $Init$. The algorithm maintains two invariants for all $i \geq 0$:

$$F_i \wedge \mathcal{T} \Rightarrow F'_{i+1}$$

$$F_{i+1} \subseteq F_i$$

That is, each frame is inductive relative to the frame below it and each frame contains a subset of the lemmas in the frame below it. These invariants imply that F_i is an over-approximation of the i-step reachable states. The algorithm also maintains a special frame F_∞ containing lemmas that over-approximate all reachable states. In addition, Quip maintains a set R of known reachable states, which is used in various optimizations.

Algorithm 1 presents pseudocode for the top-level procedure of Quip. Certain bookkeeping details are omitted for succinctness, including the details needed to construct counter-example traces. Line 2 checks if any initial states are unsafe, as the main loop does not handle this case. Lines 3 through 7 contain the main loop. The loop calls the recursive blocking procedure on line 4, which strengthens the inductive trace such that $\neg Bad$ is inductive relative to F_{k-1}. After handling all proof obligations, line 5 performs the pushing procedure. For each non-bad lemma φ in each non-empty F_i, the algorithm checks if $F_i \wedge \mathcal{T} \Rightarrow \varphi'$. If so, the lemma is promoted to level $i + 1$. If at any point $F_i = F_{i+1}$ for some value of i then F_i is inductive and can be added to F_∞. Finally, line 6 checks if $F_\infty \Rightarrow \neg Bad$. If this holds, F_∞ is a safe inductive

Algorithm 1 `Quip` $(Init, \mathcal{T}, Bad)$

```
1: $R = \emptyset,\ F_0 = Init,\ k = 1$
2: if SAT? $(F_0 \wedge Bad)$ then return UNSAFE
3: loop
4:     if $\neg$Quip_Block$(k)$ then return UNSAFE
5:     Quip_Push()
6:     if $\neg$SAT?$(F_\infty \wedge Bad)$ then return SAFE
7:     $k = k + 1$
```

Algorithm 2 `Quip_Block` (k)

```
1: $Q = \emptyset$
2: Add$(Q, \langle Bad, k, must\rangle)$
3: while $\neg$Empty$(Q)$ do
4:     $\langle m, i, p\rangle$ = Pop$(Q)$
5:     if $(i = 0) \vee$ Match$(R, m)$ then
6:         if $p = must$ then return false
7:         else AddReachable$(R)$
8:     else if SAT?$(F_{i-1} \wedge \mathcal{T} \wedge m')$ then
9:         $u$ = Predecessor$(m)$
10:        Add$(Q, \langle$Lift$(u), i-1, p\rangle)$
11:        Add$(Q, \langle m, i, p\rangle)$
12:    else
13:        $(\varphi, g)$ = Generalize$(\neg m, i)$
14:        AddLemma$(\varphi, g)$
15:        if $g < k - 1$ then Add$(Q, \langle \neg\varphi, g+1, may\rangle)$
16: return true
```

invariant and the algorithm terminates, otherwise the algorithm continues to the next iteration.

The recursive blocking procedure is described in detail in Algorithm 2. `Quip` maintains a queue of proof obligations of the form $\langle m, i, p\rangle$, where m is a cube over $\mathcal{V}$ or $m = Bad$, i is a level, and $p \in \{must, may\}$ is the type of obligation. A must-proof obligation represents a cube that must be blocked in order for the problem to be SAFE. A may-proof obligation represents a cube that may be useful to block, but its failure does not necessarily imply the problem is UNSAFE. Line 2 initializes the queue to contain the obligation to block all unsafe states at level k.

At each step, the algorithm attempts to discharge an obligation $\langle m, i, p\rangle$ with the lowest level by proving that no m-state is i-step reachable (*i.e.,* blocking m at level i). This process has three potential outcomes.

The first (lines 5–7) occurs when either $i = 0$ or R contains an m-state. In this case, the obligation cannot be discharged and a counter-example is found (if $p = must$) or new reachable states are discovered. The `AddReachable` procedure called on line 7 adds any newly-discovered reachable states to R by traversing the chain of obligations that includes $\langle m, i, p\rangle$. All lemmas in the inductive trace are checked against R at this point, and any lemma that blocks a reachable state is marked bad.

A second possibility (lines 8–11) occurs when $F_{i-1} \wedge \mathcal{T} \not\Rightarrow \neg m'$, meaning that m has a predecessor state u in F_{i-1}. The obligation $\langle m, i, p\rangle$ is returned to the queue and $\langle u, i-1, p\rangle$ is added. The `Predecessor` procedure called on line 9 extracts a predecessor state u of m from the SAT solver. A proof

Algorithm 3 `Support` $(F, \mathcal{T}, \varphi)$

```
1: $F_{en} = \emptyset$, $assumps = \emptyset$, $\Gamma(\varphi) = \emptyset$
2: for all $c_i \in F$ do
3:     $l_i$ = ActivationLit$(c_i)$
4:     $assumps = assumps \cup \{\neg l_i\}$
5:     $c_{en} = c_i \cup \{l_i\}$
6:     $F_{en} = F_{en} \cup \{c_{en}\}$
7: $\Phi = F_{en} \wedge \mathcal{T} \wedge \neg\varphi'$
8: if SAT?$(\Phi, assumps)$ then return NULL
9: $conflicts$ = ConflictAssumptions$(\Phi)$
10: for all $(\neg l_i) \in conflicts$ do
11:     $\Gamma(\varphi) = \Gamma(\varphi) \cup \{c_i\}$
12: return $\Gamma(\varphi)$
```

obligation for u is added on line 10. However, in practice, u is lifted to a smaller cube that represents more states, all of which are predecessors of m. This is handled by the `Lift` procedure.

The final possibility (lines 13–15) occurs when $F_{i-1} \wedge \mathcal{T} \Rightarrow \neg m'$. In this case, the obligation is successfully discharged and the algorithm learns a new lemma φ such that $Init \Rightarrow \varphi$, $\varphi \Rightarrow \neg m$, and $\varphi \wedge F_{i-1} \wedge \mathcal{T} \Rightarrow \varphi'$. The lemma over-approximates the i-step reachable states and demonstrates why m is not i-step reachable. It is added to all frames F_j for $j \leq i$. When blocking m, the lemma $\varphi = \neg m$ is sufficient. However, key to the performance of `Quip` and `IC3` is the `Generalize` procedure on line 13, which may find a stronger clause that is inductive relative to F_{g-1} for some $g \geq i$. The generalized lemma is added at level g on line 14. In addition, on line 15 a new obligation $\langle \neg\varphi, g+1, may\rangle$ can be added to the queue. This forces the algorithm to push φ forward, thereby blocking m at higher levels.

III. Computing Support Sets

Computing support sets, first introduced in [6], is an integral aspect of this work. This section defines the concept and describes a method to compute them as a matter of practical interest.

A support set for a lemma φ is a set of lemmas relative to which φ is inductive. `IC3` and `Quip` use this concept implicitly when executing SAT queries of the form SAT?$(F_i \wedge \mathcal{T} \wedge \neg\varphi')$ relative to various frames F_i. This query asks if φ is inductive relative to F_i, or equivalently, if F_i is a support set for φ. In practice, it is often the case that only a small subset of F_i is actually needed to support φ [6]. Various methods can be used to compute small support sets, but in this work only one method is considered. It takes as input a CNF formula F, a transition relation $\mathcal{T}$ also given in CNF, and a clause φ. It returns a subset of F relative to which φ is inductive. In other words, it returns a support set $\Gamma(\varphi) \subseteq F$. Note that the support of φ is not necessarily unique.

The basic version of the method is shown in Algorithm 3. In that description, `ActivationLit`(c) is a procedure that returns a new activation literal unique to clause c. A unique activation literal is added to each clause of F to construct a new formula F_{en}. A SAT query is constructed from the following formula:

$$F_{en} \wedge \mathcal{T} \wedge \neg\varphi' \bigwedge_{c_i \in F} \neg l_i$$

where l_i is the activation literal for clause c_i. The clauses forcing the activation literals to `0` are passed to the solver as assumptions. If the formula is satisfiable, then F is not a support set for φ and the algorithm returns NULL. Otherwise, the activation literals in the conflicting assumption set are mapped back to their corresponding clauses, each of which is added to $\Gamma(\varphi)$. Intuitively, this is equivalent to intersecting a clausal UNSAT core of $F_i \wedge \mathcal{T} \wedge \neg\varphi'$ with F_i. However, Algorithm 3 is useful in practice as it may offer better performance and can be applied when using SAT solvers without support for efficient generation of clausal cores.

In the context of `Truss`, Algorithm 3 is used to compute supports of various lemmas in the inductive trace. Given a lemma $\varphi \in F_{i+1}$ for some i, we need to compute its support relative to a subset of lemmas in frame F_i. More precisely, on each invocation we may also have a set $B \subset F_i$ of *blacklisted* lemmas, and a call to `Support` $(F_i \setminus B, \mathcal{T}, \varphi)$ asks if F_i contains a support set for φ consisting only of non-blacklisted lemmas.

Furthermore, the SAT queries in Algorithm 3 can be executed incrementally. To this end, each time `AddLemma`(φ, g) is called, φ is also added to the incremental solver with its unique activation literal. When constructing *assumps* in Algorithm 3, each activation literal corresponding to a lemma in $F_i \setminus B$ is added with negative polarity. Those corresponding to all other lemmas are added with positive polarity, effectively removing the corresponding clause from the resulting formula.

When $\varphi \in F_{i+1}$ has a support $\Gamma(\varphi) \subseteq F_i \setminus B$, what we will really need is the *critical* part of the support set, defined as $\Gamma(\varphi) \cap (F_i \setminus F_{i+1})$. This critical part of the support set represents lemmas that would be sufficient to promote in order to promote φ. In particular, when $\Gamma(\varphi) \cap (F_i \setminus F_{i+1})$ is empty, φ can be immediately added to a higher frame. In practice we do not introduce activation literals for lemmas in F_∞, as in our algorithm these lemmas are never blacklisted and will never be part of a critical support set.

IV. Safety Checking With Support Sets

This section presents the `Truss` algorithm, which solves the safety checking problem using support sets to guide its search for an inductive proof. We first explain the classification of lemmas into categories of good, bad, and the novel classification of ugly. We then present the algorithm itself. Finally, its strategy is contrasted against `Quip` and `IC3`.

A. Classifying Lemmas

A key aspect of the algorithm is its classification of lemmas into the categories of unknown, good, bad, and ugly. This subsection explains the criteria that lead to these classifications. The next subsection explains the algorithm and how it treats lemmas based on their classification.

The first three categories are also present in `Quip`, and we briefly describe them here. Unknown is the default classification, and simply means that the lemma is not known to belong to the other categories. Good lemmas are those that have been promoted to the frame F_∞. They are intuitively good because the algorithm terminates when $\neg Bad$ is inductive relative to F_∞.

On the other hand, bad lemmas are those that are known to be non-inductive. This is detected through the discovery of reachable states. Letting the cube r represent a known reachable state, a lemma φ is marked bad if the formula $\varphi \wedge r$ is unsatisfiable, which indicates that φ does not over-approximate the set of reachable states. These lemmas are undesirable for the algorithm. It is forced to spend time pushing them despite the fact that they cannot appear in a proof. As mentioned earlier, non-inductive lemmas may also make it harder for the algorithm to discover a proof. In `Quip` and in our approach, no attempt is made to push bad lemmas forward.

The novel classification used in `Truss` is *ugly*. Unlike good and bad, which are applied to a lemma permanently, ugly may be a temporary classification. In that sense, ugly is a sub-class of unknown, since ugly lemmas are also not known to be good or bad. Informally, an ugly lemma is one that appears difficult to push to higher levels. This could happen if the lemma is non-inductive, or if the algorithm simply needs to learn more supporting lemmas to push it forward. A lemma φ may be marked as ugly when considering a may-proof obligation of the form $\langle \neg\varphi, i, may \rangle$. If promoting φ to level i requires adding more than one lemma to F_{i-1}, it appears difficult to push and is therefore ugly. Different criteria could be applied instead. This criterion ignores many aspects of the true cost of supporting φ, but is simple to implement and works well in practice.

In addition, an ugly lemma may be reclassified into any of the other classifications. If the lemma is promoted to F_∞, it is marked as good. If it is found to exclude a reachable state, it becomes bad. Finally, if it is pushed forward during `Quip_Push`, it ceases to be ugly and becomes unknown. Intuitively, this is because the lemma was marked ugly as a result of insufficient support at its current level. The fact that it was successfully pushed indicates that the algorithm has learned enough supporting lemmas.

B. The Algorithm

This subsection describes the `Truss` algorithm. It uses the outer loop from `Quip` as described in Algorithm 1. It also uses a similar pushing procedure, the only difference being the re-classification of ugly lemmas as noted in the previous subsection. However, it uses a novel blocking procedure `Truss_Block` that discharges proof obligations using support sets where possible. This section describes the new blocking procedure.

As is the case for Algorithm 2, `Truss_Block` takes as input a natural number k representing the level at which to block all unsafe states. As mentioned earlier, it identifies a set of lemmas that could be close to forming a safe inductive invariant. This is accomplished by computing a critical support set for the property, and then computing critical support sets for the supporting lemmas, and so on. This continues until reaching lemmas for which no suitable support set can be found. Those lemmas are promoted using an approach similar to that used by `Quip_Block`. When a lemma cannot be promoted, the set is abandoned and a new one identified.

This procedure is integrated with the proof obligation processing scheme. Before explaining the algorithm, we first

Algorithm 4 `Truss_Block` (k)

```
1:  $Q = \emptyset$
2:  $E[\varphi] = 0\ \forall$ lemmas $\varphi$
3:  $\texttt{Add}(Q, \langle Bad, k, must\rangle)$
4:  while $\neg\texttt{Empty}(Q)$ do
5:      $\langle m, i, p\rangle = \texttt{Pop}(Q)$
6:      if $(i = 0) \vee \texttt{Match}(R, m)$ then
7:          if $p = must$ then return false
8:          else
9:              $\texttt{AddReachable}(R)$
10:             $Q = \{\langle Bad, k, must\rangle\}$; continue
11:     $Ind = \neg\texttt{SAT?}(F_{i-1} \wedge \mathcal{T} \wedge m')$
12:     if $(i > 2) \wedge \neg Ind \wedge \texttt{IsEligible}(\langle m, i, p\rangle)$ then
13:         $\Gamma(\neg m) = \texttt{Support}(F_{i-2} \setminus (B \cup U), \mathcal{T}, \neg m)$
14:         if $\Gamma(\neg m) \neq$ NULL then
15:             for all $\varphi \in (\Gamma(\neg m) \setminus F_{i-1})$ do
16:                 $\texttt{Add}(Q, \langle\neg\varphi, i-1, may\rangle)$
17:             $\texttt{Add}(Q, \langle m, i, p\rangle)$; continue
18:         else
19:             if $p = may \wedge E[\neg m] \geq 1$ then
20:                 $U = U \cup \{\neg m\}$
21:                 $Q = \{\langle Bad, k, must\rangle\}$; continue
22:             $E[\neg m] = E[\neg m] + 1$
23:     if $\neg Ind$ then
24:         $u = \texttt{Predecessor}(m)$
25:         $\texttt{Add}(Q, \langle\texttt{Lift}(u), i-1, p\rangle)$
26:         $\texttt{Add}(Q, \langle m, i, p\rangle)$
27:     else
28:         $(\varphi, g) = \texttt{Generalize}(\neg m, i)$
29:         $\texttt{AddLemma}(\varphi, g)$
30: return true
```

introduce two different methods by which proof obligations are processed in `Truss`. The first is the *fallback behavior*. When processing an obligation using the fallback behavior, the algorithm behaves identically to Algorithm 2, except that it does not add may-proof obligations *i.e.,* line 15 is not executed. Alternatively, an obligation can be processed using support sets. In this case, the obligation $\langle m, i, p\rangle$ is processed by computing a support set $\Gamma(\neg m)$ and enqueuing may-proof obligations $\langle\varphi, i-1, may\rangle$ for all $\varphi \in \Gamma(\neg m)$. If a suitable support set cannot be found, then the obligation is processed using the fallback behavior.

For some obligations, the algorithm skips the support set computation and proceeds directly to the fallback behavior. Obligations that are processed using support sets are called *eligible*. Different eligibility criteria could be considered, and one alternative is discussed in Section VI-C. In this section, eligible obligations are $\langle Bad, k, must\rangle$ and may-proof obligations $\langle\neg\varphi, i, may\rangle$ where φ is a lemma already present in the inductive trace.

Pseudocode for the procedure is shown in Algorithm 4. `Truss_Block` is only called at level k when $\neg Bad$ is inductive relative to F_{k-2}. The algorithm's goal is to strengthen F_{k-1} until $\neg Bad$ is inductive relative to that frame. It begins by processing the obligation $\langle Bad, k, must\rangle$, which is added on line 3. We discuss the handling of this obligation first, and later move on to general obligations. The first step is to check if $\neg Bad$ is already inductive relative to F_{k-1} (line 11). If so, lines 28–29 are executed, adding a lemma $\neg Bad$ at a level $g \geq k$, and the algorithm terminates. Otherwise, it tries to compute a critical support set $\Gamma(\neg Bad)$ from F_{k-2}, excluding any blacklisted (*i.e.,* bad or ugly) lemmas (line 13). This may not succeed, as the property may be supported by those lemmas. In this case, the algorithm uses the fallback behavior, which results in adding a new must-proof obligation at level $i-1$ (lines 24–26).

Now, assume a support set is found. The algorithm adds obligations $\langle\neg\varphi, k-1, may\rangle$ for each $\varphi \in \Gamma(\neg Bad)$ (lines 15–16). The original obligation is also returned to the queue. Note that, since the lemmas in $\Gamma(\neg Bad)$ are in F_{k-2}, they are inductive relative to F_{k-3}. Since the obligations are enqueued at level $i = k-1$, each φ is inductive relative to F_{i-2}. The added may-proof obligations at level i are handled similarly, by computing a support set from F_{i-2} and enqueuing obligations at level $i-1$. This is the reasoning behind Lemma 1 below.

Lemma 1 *In `Truss`, for every proof obligation $\langle\neg\varphi, i, p\rangle$ with $i \geq 2$, φ is inductive relative to F_{i-2}.*

Proof: For $\langle Bad, k, must\rangle$, the proof is trivial. For obligations added on line 25, the proof follows from the properties of `Quip` and `IC3`. For obligations added on line 16, the obligation $\langle\neg\varphi, j, may\rangle$ is added at level $j = i-1$. We have $\varphi \in F_{i-2}$ by the behavior of `Support`. Since $\varphi \in F_{i-2}$ it is inductive relative to F_{i-3} *i.e.,* F_{j-2}. The lemma follows immediately. ■

We now describe the processing of an eligible obligation $\langle\neg\varphi, i, may\rangle$. Note that must-proofs other than $\langle Bad, k, must\rangle$ are not eligible, so the obligation is assumed to be a may-proof. The first step on lines 6–10 is to check if the obligation fails. This step is the same in `Quip_Block` and is also part of the fallback behavior. The next step is to check if φ is inductive relative to F_{i-1} (line 11). If so, the obligation is successfully discharged. Otherwise, by Lemma 1, φ is inductive relative to F_{i-2}. The algorithms tries to compute a support set for φ of the form:

$$\Gamma(\varphi) \subseteq F_{i-2} \setminus (B \cup U) \tag{4}$$

where B and U represent the set of bad and ugly lemmas, respectively. This occurs on line 13.

If a support set is found, obligations are added for the supporting lemmas on lines 15–16. Note that due to the subtraction of F_{i-1} on line 15, only the lemmas in the critical support set are added.

Conversely, if a support set is not found, the algorithm tries to learn new lemmas to support φ. The rationale behind this behavior comes from the following corollary of Lemma 1.

Corollary 1 *If no support set of the form from Eq. 4 exists, all support sets $\Gamma(\varphi) \subseteq F_{i-2}$ include some lemma in $B \cup U$.*

Proof: Immediate from Lemma 1. ■

Corollary 1 does not imply that φ is non-inductive. However, promoting φ to level $i-1$ requires promoting blacklisted lemmas or learning new ones. The former is undesirable, so the algorithm uses its fallback behavior to learn new lemmas (lines 23–29). However, an effort limit is applied that restricts the algorithm to learn only one new lemma towards the goal of supporting φ. When a support set cannot be found, line 19 checks if the effort limit for φ has been exceeded. If so, the entire obligation queue is abandoned and φ is marked as ugly (lines 19–21). In other words, the second time $\langle \neg\varphi, i, may \rangle$ is popped from the queue, if φ is not inductive relative to F_{i-1} then φ is marked as ugly and the queue is abandoned.

Throughout this procedure, obligations may not be discharged due to a counter-example or due to effort limits. Consider the case where an obligation intended to push a lemma $\psi \in \Gamma(\varphi)$ forward is abandoned. The algorithm has no reason to continue pushing the other lemmas in $\Gamma(\varphi)$ either, since $\Gamma(\varphi) \setminus \{\psi\}$ is not necessarily a support set for φ. Intuitively, the corresponding obligations should be abandoned. However, it may also be the case that φ has no suitable support set, since ψ is now blacklisted. This means φ should be abandoned. These cascading failures can end up requiring a large number of obligations to be abandoned.

Rather than checking each individual obligation, the algorithm simply abandons all may-proof obligations when any one of them cannot be discharged. This leaves only $\langle Bad, k, must \rangle$ in the queue. The algorithm simply repeats all of the steps of computing support sets for the property, then for the supporting lemmas, and so on. In practice, re-computing support sets would be costly, so the most recently-computed support set for each lemma is cached. When the computations are repeated, the cached result is used unless it contains a blacklisted lemma. Therefore, only the support sets that have been invalidated are re-computed.

A corner case occurs when $i < 2$, as F_{i-2} does not exist. The algorithm resorts to the fallback behavior in this case. Additionally, for performance reasons, the fallback behavior is used when $i = 2$. This is because F_0 represents the initial states, and as such contains lemmas that are given as input rather than learned by the algorithm. These lemmas are expected to be non-inductive, so pushing them forward is undesirable. Therefore, the fallback behavior is used when $i \leq 2$. Note that in this case proof obligations are not subjected to effort limits.

C. Comparison with Quip and IC3

`Quip`, `IC3`, and `Truss` all repeatedly attempt to discharge $\langle Bad, k, must \rangle$ for increasing values of k. This represents an obligation to construct a bounded proof of the property at level k. All three algorithms repeat this process until converging to an inductive proof.

The algorithms differ in the additional reasoning they apply to accelerate convergence. In `IC3`, when an obligation $\langle m, i, must \rangle$ is discharged, $\langle m, i+1, must \rangle$ may be returned to the queue. This ensures that the CTI represented by m is blocked at higher levels, but causes the algorithm to expend effort learning multiple lemmas to block the same CTI. In `Quip`, upon discharging the same obligation by learning a lemma φ, a new obligation $\langle \neg\varphi, i+1, may \rangle$ may be added to the queue. This forces the algorithm to try to promote φ to block m at higher levels, even if it requires learning additional lemmas to support φ. This can be an expensive process and may only result in the algorithm discovering reachable states instead of blocking m at higher levels. The algorithms apply these consistently without regard to the particular lemmas or CTIs being considered.

`Truss` instead uses support sets to guide these decisions. Rather than consistently trying to push forward every lemma, support sets are used to identify a set of lemmas that might be close to forming an inductive invariant. In fact, the set of lemmas identified represents a portion of a bounded proof excluding any bad or ugly lemmas. It may be the case that this set of lemmas is only useful together *i.e.,* if one cannot be promoted, the entire set is not useful. Therefore, when a targeted lemma is not promoted, the algorithm detects and abandons obligations that are only valuable in conjunction with that lemma. In essence, the algorithm tries to identify the portion of the existing bounded proof that is likely to be inductive and support it until it becomes inductive. However, effort limits are used to limit this procedure.

All of these algorithms are forced to construct bounded proofs at higher and higher levels in order to "escape" the non-inductive lemmas they have learned and to learn new lemmas to replace them. In `IC3`, once a lemma is learned, no effort will ever be made to learn new lemmas to support it and push it forward. It will only be pushed forward if such lemmas are learned by chance. In `Quip`, an effort is made immediately upon learning a lemma to learn its supporting lemmas using may-proof obligations. However, after that process finishes, no further effort is made. `Truss` is able to identify important lemmas and then learn new lemmas to support them at any time. We believe this represents a significant extension of the algorithm's reasoning capabilities.

V. Experimental Results

All results presented in this section are executed on a single core of a Linux workstation with an i5-3570K 3.4 GHz CPU and 16 GB of RAM. We provide an experimental evaluation of `IC3`, `Quip`, and `Truss`. We have implemented both `Quip` and `Truss`[1] in `IImc` [7], [8]. For all algorithms, the backend SAT solver is `Glucose` [9], [10] as it was found to give a substantial runtime improvement over the other solvers available in `IImc`. Experiments are timed out after one hour.

We present results for problem instances in the HWMCC'15 benchmark set, excluding the proprietary circuits from Intel. The 126 benchmarks that were not solved by any solver in the competition are not considered, leaving 387 circuits. A further 122 circuits that were not solved by any of the evaluated algorithms are excluded. After pruning those circuits, the benchmark set contains 265 circuits.

In order to demonstrate that the "baseline" `Quip` approach is reasonable, it is compared against the `IC3` implementation provided by `IImc`, referred to as `IImc-IC3`. However, we disable several features that are not present in our implementations, including expansion of the initial states using forward Binary Decision Diagrams [11], equivalence propagation, and counter-examples to generalization [12]. All of the disabled features are applicable to `Quip` and `Truss`, but are not present in our implementation. These features are disabled so

[1]Source code and detailed results for each circuit are available at: ryanmb.bitbucket.io/truss

TABLE I
SUMMARY OF RESULTS

	SAFE SOLVED	SAFE TIME	UNSAFE SOLVED	UNSAFE TIME	TOTAL SOLVED	TOTAL TIME
IImc-IC3	175 (6)	77632	66 (5)	30131	241 (11)	107765
Quip	174 (3)	83450	56 (1)	66067	230 (4)	149517
Truss	183 (8)	57394	64 (5)	44147	247 (13)	101541

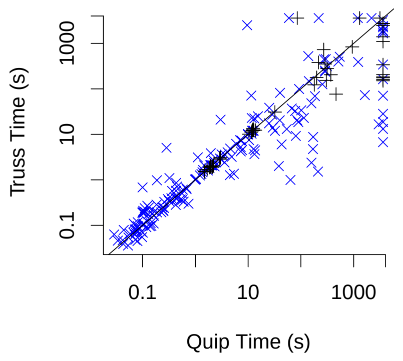

Fig. 1. Runtime comparison of Quip and Truss

as to limit the impact that the unrelated optimizations present in IImc-IC3 have on the results.

A summary of the results is shown in Table I. The columns SAFE SOLVED and UNSAFE SOLVED show the number of safe and unsafe instances solved by each algorithm, respectively. The number in parenthesis shows the number of unique instances solved. The TOTAL SOLVED column shows the total number of instances solved by each algorithm. The SAFE TIME and UNSAFE TIME columns show the total time spent by each algorithm on safe and unsafe instances respectively. The TOTAL TIME column shows the total time spent processing all instances. All times are in seconds.

The experiments show that our Quip implementation, while weaker in comparison to IImc-IC3, is competitive and represents a reasonable baseline for comparison. Since Quip is expected to outperform IC3 in typical cases, we expect this is due to other unrelated optimizations present in IImc-IC3 that could not readily be disabled. The experiments demonstrate that Truss offers a substantial improvement over Quip for both SAFE and UNSAFE instances. It also outperforms the highly-tuned IImc-IC3 implementation, especially on SAFE instances where it achieves a 1.35x speedup. Out of 265 circuits, Truss solves 247 instances compared to the 230 solved by Quip and processes the entire set 1.47x faster. Not counting the 11 instances uniquely solved by IImc-IC3, the speedup increases to an impressive 1.77x.

Figure 1 shows a detailed comparison of the runtime for each approach. It plots the runtime of Truss versus that of Quip for each of the benchmark circuits on a log-log scale. The blue marks indicate SAFE instances while the black marks indicate UNSAFE instances. It includes the 254 circuits that were solved by at least one of Quip or Truss. Points under the solid line indicate that Truss is faster, while points above it indicate that Quip is faster. It can be seen that Truss is faster than Quip in most cases. Indeed, 145 of the 254 points fall below the line. However, this

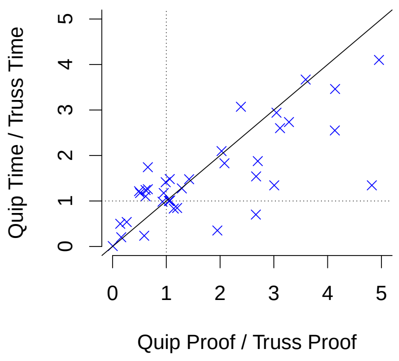

Fig. 2. Speedup versus reduction in proof size for challenging instances

significantly understates the benefits offered by Truss, as it truly shines on the more challenging problem instances. The benchmark sets contains 139 “easy” instances that were solved by both algorithms in under 12 seconds. Excluding the easy instances, Truss outperforms Quip on 81 out of 115 of the remaining “challenging” instances. Apparently, Truss introduces overhead for easy benchmarks, but pays off substantially for challenging ones.

The intuition behind Truss is that by promoting important lemmas, the algorithm is able to more quickly discover a proof. Naturally, it is expected to learn smaller proofs as a result. It may not always do so, as random factors can significantly impact the runtime and change the final proof. For instance, Quip may learn a very important lemma by chance that Truss never learns due to having a different inductive trace. Truss does not address the problem of learning better ones, so this problem is unavoidable.

To examine the relationship between runtime and proof size, Figure 2 plots the speedup for Truss versus the proof size reduction. It includes the 49 SAFE instances in the challenging set that were solved by both algorithms. The solid line indicates a 1:1 correlation between the two axes. Across these 49 instances, Truss finds proofs that are an average of 73.3% as large as those found by Quip. It can be seen that higher speedups tend to occur when Truss computes smaller proofs than Quip. This is unsurprising, as Truss tends to find a smaller proof by processing fewer obligations for more important lemmas.

To further examine this point, Table II reports the runtime of various operations for Quip and Truss. It considers the 83 challenging instances solved by both algorithms. The first column reports the total runtime. The PUSH TIME, MAY TIME, MUST TIME, and SUPPORT TIME columns report the time spent pushing, processing may-proofs (excluding calls to Support), and processing must-proof obligations, and computing support sets, respectively. The MAY PROOFS and MUST PROOFS columns report the number of may-proof and must-proof obligations processed, respectively. The SUPPORT CALLS column reports the number of calls to Support.

The data explains how Truss achieves better runtime performance. It spends similar amounts of time pushing lemmas and processing must-proofs. However, it processes fewer than one third as many may-proofs as Quip. Additionally, the overhead of computing support sets is small, accounting for

TABLE II
TIME SPENT PER OPERATION

	TOTAL TIME	PUSH TIME	MAY TIME	MAY PROOFS	MUST TIME	MUST PROOFS	SUPPORT TIME	SUPPORT CALLS
`Quip`	14000	2940	7108	1826812	2940	88983	0	0
`Truss`	10878	3317	3372	554233	2935	87616	517	135736

only 4.7% of the total runtime. Indeed, computing a support set is expected to cost substantially less than processing a proof obligation. Processing a proof obligation often results in learning a new lemma, thereby running generalization which may involve numerous SAT queries. Conversely, computing a support set involves only one SAT query. `Truss` avoids a large number of expensive generalization operations by performing a smaller number of less expensive support set computations.

VI. ALTERNATIVES AND FUTURE WORK

This section presents alternative implementations of `Truss` that combine the novel aspects in different ways. We also discuss ways that the implementation could be improved and aspects that may be applicable to other algorithms.

A. No Ugly Lemmas

An alternative version of `Truss` could use different criteria to define ugly lemmas. The simplest alternative is to increase the effort limit from 1. In the most extreme form, infinite effort could be allowed, thereby eliminating ugly lemmas altogether. A preliminary evaluation of this approach found it to perform poorly. The algorithm expends much more effort pushing forward lemmas that ultimately end up being marked as bad. Even increasing the effort limit slightly had a similar but less dramatic effect. Intuitively, it appears as though valuable lemmas tend to be easy to support in most cases, since `Truss` achieves better results when using a low effort limit.

B. Re-Enqueuing Obligations

`Truss` does not re-enqueue obligations in the manner `Quip` does on line 15 of Algorithm 2. The algorithm can be modified to accommodate this operation, though several variations are reasonable. For instance, it's unclear if re-enqueued lemmas should be subject to effort limits or not. A preliminary experimental evaluation found that several variations of this approach performed worse than `Truss`. One possible explanation is that the re-enqueue operation is to ensure important lemmas are available at higher levels. This is exactly the same reasoning behind adding lemmas from support sets. However, support sets contain more fine-grained information about which lemmas are important. Therefore the extra proof obligations from the re-enqueue operation may be less helpful than those added by `Truss`.

C. Using Support Sets for Non-Lemmas

Another alternative implementation could use different eligibility criteria, such as using support sets to discharge every proof obligation. In `Truss`, support sets are only used when an obligation represents a lemma in the inductive trace. However, Lemma 1 holds for every proof obligation, so it would be a reasonable to use support sets for every proof obligation. This procedure takes the goal of re-using of existing lemmas rather than learning new ones to the extreme.

D. Improved Solving under Assumptions

Algorithm 3 for computing lemma supports is based on incremental SAT solving with *many* assumptions, which may significantly slow down SAT-queries, see for example [13]. However, on problems where all assumptions can be cleanly separated into original problem literals and activation literals various solutions are possible [13], [14]. In the future we are planning to investigate the precise effect of assumptions on Algorithm 3 and adjust the back-end SAT solver accordingly. In addition, most state-of-the-art SAT solvers tend to propagate assumptions in the order they are received and it seems a good idea to experiment with different assumption orders. For example, by putting the activation literals in decreasing order of level for the corresponding lemma, the algorithm is likely to find a smaller critical support set.

VII. CONCLUSION

This work presents an `IC3`-based unbounded model checking algorithm called `Truss`. The algorithm detects a subset of lemmas in the inductive trace that participate in a bounded proof of the property and targeting the set for for pushing as a cohesive unit. Experiments on HWMCC'15 designs show a substantial speedup against the state-of-the-art.

REFERENCES

[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *DAC 2015*, June 2015, pp. 1–6.

[2] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, 2011.

[3] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," ser. FMCAD 2011, Austin, TX, 2011, pp. 125–134.

[4] A. R. Bradley, "Incremental, inductive model checking," in *2013 International Symposium on Temporal Representation and Reasoning*, Sept 2013, pp. 5–6.

[5] A. Ivrii and A. Gurfinkel, "Pushing to the top," in *FMCAD 2015*, Sept 2015, pp. 65–72.

[6] R. Berryhill, N. Veira, A. Veneris, and Z. Poulos, "Learning lemma support graphs in quip and IC3," in *Proceedings of the 2017 International Verification and Security Workshop (IVSW) 2017*, 2017.

[7] A. R. Bradley and F. Somenzi and Z. Hassan, "IImc: an Incremental Inductive model checker." [Online]. Available: https://github.com/mgudemann/iimc

[8] Z. Hassan, A. R. Bradley, and F. Somenzi, "Incremental, inductive CTL model checking," in *CAV 2012*, 2012, pp. 532–547.

[9] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the 21st International Joint Conference on Artifical Intelligence (IJCAI)*, San Francisco, CA, USA, 2009, pp. 399–404.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, 2003, pp. 502–518.

[11] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[12] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in ic3," in *FMCAD 2013*, Oct 2013, pp. 157–164.

[13] J. Lagniez and A. Biere, "Factoring out assumptions to speed up MUS extraction," in *SAT 2013*, 2013, pp. 276–292.

[14] G. Audemard, J. Lagniez, and L. Simon, "Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction," in *SAT 2013*, 2013, pp. 309–317.