

Learning Lemma Support Graphs in Quip and IC3

Ryan Berryhill¹, Neil Veira¹, Andreas Veneris^{1,2}, Zissis Poulos¹

Abstract— Formal verification is one of the fastest growing fields in verification. The Boolean satisfiability-based unbounded model checking algorithm of IC3 has become widely applied in industry and is frequently used as a subroutine in other formal verification algorithms, such as FAIR and IICTL. Any improvement to IC3 can therefore yield substantial benefits in many areas of formal verification. Towards that end, this paper introduces the notion of a support graph, which is applied in IC3. Techniques are presented to compute the support graph by modifying the satisfiability queries used in IC3 at the cost of a modest increase in runtime. It is used to increase the re-use of information across runs of the model checker, thereby improving runtime performance in incremental model checking. It can also be applied within a single run of the model checker to avoid unnecessary queries to the satisfiability solver and accelerate the discovery of a proof. Experiments are presented on HWMCC’15 circuits demonstrating the benefits of the presented approaches.

I. INTRODUCTION

Verification is the primary bottleneck in hardware design, consuming an average of 57% of the total project time [1]. Formal verification is one of the fastest growing segments in the field [1]. Of particular importance in formal verification is the problem of unbounded model checking, which asks if particular states are reachable in a circuit. IC3 [2] (also known as Property-Directed Reachability (PDR) [3]), has established itself as one of the state-of-the-art model checking techniques, and is now widely applied in industry [4], [5]. IC3 has been generalized to other domains such as software model checking [6], [7] and is frequently used as a subroutine in other algorithms [8], [9]. Any improvements to IC3 can therefore have wide-reaching impact in many areas of formal verification.

The core functionality of IC3 is as follows. It accepts as input a formula representing a safety property. A Boolean Satisfiability (SAT)-based procedure identifies states that can reach a property violation. Subsequently, the algorithm tries to learn a lemma that explains why such states are not reachable in a specific number of clock cycles, called the lemma’s *level*. The lemmas form a sequence of over-approximations of the reachable state space at each level. A procedure of *pushing* promotes lemmas from one level to the next, strengthening the approximations. The runtime of these algorithms is dependent on learning and pushing relevant lemmas. Additionally, in an incremental setting where the algorithm is called more than once, re-using lemmas can improve runtime. Learning high quality lemmas is mostly within the scope of generalization, which has been studied extensively [8], [10], [11]. Towards the other goals stated, this paper presents a framework to learn the relationships between lemmas. It can be used in the incremental setting to increase the re-use of lemmas from previous runs, and in a non-incremental setting to identify relevant lemmas earlier and exploit that knowledge.

Our approach works by learning which lemmas are needed to *support* other lemmas. In pushing, a lemma at a particular level can only be promoted to a higher level (*i.e.*, proven to over-approximate reachable states for a greater number of cycles) when the approximation at its current level is strong enough. In practice however, only a small number of specific lemmas are necessary to support pushing in most cases. Quip and IC3 learn that a lemma can

be pushed using a single query to a SAT solver. Our approach uses the results of a similar query to both determine whether or not the lemma can be pushed and learn which other lemmas were necessary to support this fact. A variety of approaches are presented that use either unsatisfiable cores or the solver’s conflict trail to compute the necessary lemmas. The results are stored in a *support graph*, in which each lemma is a vertex and directed edges indicate that the source of the edge supports its destination.

Two applications of the support graph are considered. The first is accelerating incremental runs of the model checker. When the model checker is called multiple times with different properties, almost all of the internal state can be re-used. However, when the initial state changes between calls, as is the case in IICTL [8], most of the lemmas have to be discarded. Using the support graph, it is possible to identify which bits of the initial state were necessary to support particular lemmas, and therefore save those that are still accurate over-approximations. Our experiments demonstrate that in practice, few of the initial state bits support any lemmas, and therefore most of them are saved in the incremental setting. The second application is accelerating the model checker itself in the standard non-incremental setting. In particular, it is our expectation that the support graph can be used to develop heuristics that identify high-quality lemmas. Using the additional capabilities that Quip [12] (a recent extension of IC3 with additional reasoning capabilities) offers over IC3, these lemmas can be targeted for aggressive pushing. This essentially provides additional guidance to the algorithm in its search for a proof.

Experiments on HWMCC’15 circuits demonstrate the potential of this approach. We find that vertices in the support graph have an average in-degree of roughly 12. However, most lemmas have a much lower in-degree, with a mode of 2. Additionally, we modified an implementation of IICTL to leverage the incrementality application mentioned above. It is found that over 60% of lemmas can be saved using this method, though it is highly dependent on the particular problem. Heuristics based on the support graph are also developed and examined experimentally. Re-using lemmas and using simple heuristics is found to speed up model checking somewhat. Computing the support graph introduces an average slowdown of 22% that is partially mitigated by the improvements noted above.

The rest of this paper is organized as follows. Section II presents background information on IC3 and Quip. Section III presents the techniques used to compute support graphs. Section IV presents the applications of the support graph. Section V presents experimental results and section VI concludes the paper.

II. PRELIMINARIES

A. Notation

The following notation and terminology is used throughout this paper. For a sequential circuit C , let $S = \{s_1, \dots, s_{|S|}\}$ denote the set of state elements (registers) of C , and let $S' = \{s' | s \in S\}$ denote the set of next-state elements (inputs to registers) of C . For a formula P over the state elements S , the primed formula P' represents the same formula over the next-state elements (*i.e.*, with each $s_i \in S$ replaced by s'_i). A *state* is an assignment to all of the state elements, and can be represented by a cube over S . For instance, if $S = \{s_1, s_2, s_3\}$, the state in which $s_1 = 1$, $s_2 = 1$, and $s_3 = 0$ is represented by the cube $(s_1 \wedge s_2 \wedge \neg s_3)$. Applying the same priming notation above to a state t , the primed version t' represents the same cube over S' .

¹University of Toronto, ECE Department, Toronto, ON M5S 3G4
({ryan@eecg, neil@mail, veneris@eecg, zpoulos@eecg}.toronto.edu)

²University of Toronto, CS Department, Toronto, ON M5S 3G4

The transition relation of C is $T(S, S')$ and is assumed to be represented in conjunctive normal form (CNF). Given a pair of states (t_0, t_1) , the formula $t_0 \wedge T \wedge t_1'$ is satisfiable (SAT) if and only if there exists an assignment to the primary input of C that causes a state transition from t_0 to t_1 . If the formula is SAT, then there is an assignment to the primary input such that t_0 is assigned to the state elements and t_1 to the next-state elements, because t_1 is primed in the formula. The set of initial states of C is also represented in CNF as $I(S)$ where a state t is an initial state if and only if $t \wedge I$ is satisfiable. Any predicate $P(S)$ can similarly be represented in CNF, where $t \in P$ if and only if $t \wedge P$ is satisfiable. For such a predicate, any state $t \in P$ is referred to in this paper as a P -state.

States are called i -step reachable if they can be reached in i or fewer steps from an initial state. Initial states are zero-step reachable. States that are i -step reachable for some value of i are *reachable*.

B. Problem Definition

An unbounded model checking problem is a tuple (S, I, T, P) where S is the set of state elements, $I(S)$ is a predicate representing the set of initial states, $T(S, S')$ represents a transition relation, and $P(S)$ is a predicate representing the set of safe states. This is similar to the definition used in [12]. The goal is to prove that P is invariant, which requires all reachable states are P -states (*i.e.*, safe states), or to find a counter-example trace showing that a $\neg P$ -state (*i.e.*, an unsafe state) is reachable. A proof takes the form of a *safe inductive invariant* $V(S)$, which is a formula satisfying the following three properties.

$$I(S) \implies V(S) \quad (1)$$

$$V(S) \wedge T(S, S') \implies V(S') \quad (2)$$

$$V(S) \implies P(S) \quad (3)$$

A formula satisfying Eq. 1 is said to satisfy *initiation*, meaning that when interpreted as a predicate, it contains all initial states. A formula satisfying Eq. 2 is *inductive*. An inductive formula represents a set of states where no V -state can reach a $\neg V$ -state. In other words, once the circuit is in a V -state, it will never transition to a state that is not a V -state. Note that the combination of initiation and induction implies that V over-approximates all reachable states. A formula satisfying both Eq. 1 and Eq. 2 is called an inductive invariant. Finally, a formula satisfying Eq. 3 is *safe*. A safe formula represents a subset of the safe states. It can be seen that a safe inductive invariant is both a superset of all reachable states and a subset of all safe states. Such a formula is a proof that the safety property P holds. Note that each of the above properties can be verified using a single query to a SAT solver.

C. Overview of IC3

This section presents a simplified explanation of IC3 [2], [3]. Given an unbounded model checking problem, IC3 works by maintaining a sequence of formulas F_0, F_1, \dots, F_k over S called the *inductive trace*. Each F_i is a CNF formula called a *frame*. A frame F_i over-approximates the set of i -step reachable states and satisfies initiation. The frame F_0 is identical to I . Each clause of F_i is called a *lemma*. The set of F_i -states is a subset of the set of F_{i+1} -states, and the lemmas in F_{i+1} are all present in F_i .

The algorithm proceeds through a series of iterations $0, 1, \dots, k$ in which iteration i seeks to prove that no unsafe state is i -step reachable. Iterations 0 and 1 are special cases, each consisting of a single SAT query to detect zero-step and one-step counter-examples, respectively. In iterations 2 and later, the algorithm uses a priority queue of *proof obligations*. A proof obligation is a pair (t, i) , where t is a cube over S and i is a natural number referred to as the *level* of the obligation. Obligation (t, i) indicates a requirement to prove that no t -state is i -step reachable.

Algorithm 1 IC3(S, I, T, P)

```

1:  $F_0 = I$ 
2: if SAT( $F_0 \wedge \neg P$ ) or SAT( $F_0 \wedge T \wedge \neg P'$ ) then return UNSAFE
3:  $k = 2$ 
4: loop
5:   Enqueue( $Q, (\neg P, k)$ )
6:   while !Empty( $Q$ ) do
7:      $(t, i) =$  Dequeue( $Q$ )
8:     if SAT( $F_{i-1} \wedge T \wedge t'$ ) then
9:        $u =$  predecessor of  $t$ 
10:      if  $i = 1$  then return UNSAFE
11:      Enqueue( $Q, (\text{LIFT}(u), i - 1)$ )
12:      Enqueue( $Q, (t, i)$ )
13:    else
14:       $(c, g) =$  GENERALIZE( $t, i$ )
15:      ADDLEMMA( $c, g$ )
16:      if  $g < k - 1$  then Enqueue( $Q, (t, g + 1)$ )
17:    if PUSHCLAUSES() = PROOF then return SAFE
18:     $k = k + 1$ 

```

An iteration k begins by enqueueing a proof obligation for $(\neg P, k)$ and ends when the queue is empty. At each step, the algorithm takes the obligation (t, i) with the smallest level from the queue. This will have one of three possible effects: finding a counter-example, adding a new obligation at level $i - 1$, or learning a lemma to block all t -states from F_i .

A SAT query for $F_{i-1} \wedge T \wedge t'$ determines which of those possibilities occurs. If the formula is satisfiable, then the satisfying assignment contains an F_{i-1} -state u that is a predecessor of a t -state. If $i = 1$, then u is an initial state, meaning t is reachable and a counter-example has been found. Otherwise, meeting the obligation requires the removal of u from F_{i-1} . A new obligation $(u, i - 1)$ is enqueued and (t, i) is returned to the queue, as t may have other predecessors in F_{i-1} . Alternatively, the formula is unsatisfiable, which implies that no t -states are i -step reachable. To record this fact, a new lemma is derived from t by applying a generalization procedure on the clause $\neg t$. An explanation of generalization is beyond the scope of this work. It returns a lemma that contains a subset of the literals from $\neg t$ and over-approximates the set of i -step reachable states. The generalized lemma is conjoined to F_i .

When the queue is empty F_{k-1} contains no predecessors of unsafe states, implying that no k -step counter-examples exist. A pushing step is executed next, which strengthens the inductive trace and attempts to find a proof. For each value of i from 1 to $k - 1$, the algorithm tries to push every lemma c in F_i to F_{i+1} . This is done using a SAT query for $F_i \wedge T \wedge \neg c'$. If the formula is satisfiable then c is not pushed. If it is unsatisfiable, then no $\neg c$ -state is $(i + 1)$ -step reachable and c is added to F_{i+1} . If every clause is pushed from some F_i to F_{i+1} , then $F_{i+1} = F_i$. This implies that F_i is a safe inductive invariant proving the property.

Pseudocode for the procedure is shown in Algorithm 1. In the description, the procedure ADDLEMMA(c, g) adds the lemma c to all of the formulas F_i for $i \leq g$. A few points that were not mentioned above are noted here. On line 11, a procedure LIFT is called on u . Lifting converts u from a single state to a cube representing a set of states, all of which can reach a t -state in one step. On line 14, generalization occurs. It returns a lemma c and a level $g \geq i$, and c is added at level g . An important optimization in IC3 allows generalization to push lemmas forward, so the generalized lemma may be added at a higher level than that of the proof obligation. Another optimization is on line 16. When a proof obligation is met and a lemma is added at level g , the obligation can be re-enqueued at

Algorithm 2 PUSHCLAUSES()

```

1: for  $i = 1$  to  $k - 1$  do
2:   for all clauses  $c$  in  $F_i$  do
3:     if  $\text{!SAT}(F_\infty \wedge T \wedge \neg c')$  then
4:       ADDLEMMA( $c, \infty$ )
5:     else if  $\text{!SAT}(F_i \wedge T \wedge \neg c')$  then
6:       ADDLEMMA( $c, i + 1$ )
7:   if  $F_i = F_{i+1}$  then return PROOF
8: return NO_PROOF

```

level $g + 1$. This is because a proof obligation represents a state that can reach a property violation. Therefore, it eventually needs to be removed from every F_i , even if it may not lead to a k -step counter-example. This can potentially increase the chances of finding a proof or counter-example more quickly.

Pseudocode for PUSHCLAUSES is shown in Algorithm 2. An important aspect that was not noted above is on line 3. A special frame F_∞ contains lemmas that are absolute invariants, *i.e.*, they over-approximate the set of all reachable states. Intuitively, F_∞ behaves like any other frame in the inductive trace. In particular, all lemmas in F_∞ are present in all frames. It is also possible for GENERALIZE to return (c, ∞) , meaning that the generalized lemma c is an absolute invariant and can be added to F_∞ .

D. Quip

Quip [12] extends IC3 with additional reasoning capabilities and performance enhancements. A key difference is that Quip has a more flexible form of proof obligation, where obligations are triples (t, i, m) , and $m \in \{\text{may}, \text{must}\}$ is the type of obligation. A must-proof obligation is similar to a proof obligation in IC3. A may-proof obligation is handled the same way as a must-proof obligation, but its failure does not imply the existence of a counter-example. This provides a means to aggressively push forward particular lemmas if desired. The algorithm can enqueue the negation of a lemma as a may-proof obligation, forcing it to try to push the lemma forward even if that requires learning new lemmas.

Given a may-proof obligation (t, i, may) , Quip may fail to meet such an obligation without finding a counter-example. The algorithm can therefore dynamically discover traces containing reachable states. Quip stores the states and uses them in various performance optimizations. In particular, when pushing clauses forward, if any lemma is found to exclude a known reachable state, it is marked as bad and no further attempts are made to push it forward. This is because such a lemma can never appear in an inductive invariant, and therefore little is gained by pushing it forward. Other differences are present in Quip, but they are not important to this work.

III. SUPPORT GRAPHS

This section defines the support graph, which is informally described as a directed graph that indicates which supporting lemmas are needed to push each lemma. Techniques to compute support graphs are also introduced in this section. The tradeoffs between the techniques are examined experimentally in section V.

A. Support Sets and Support Graphs

The support graph is a directed graph in which vertices correspond to lemmas. The presence of a directed edge (c_1, c) indicates that lemma c_1 is in the current *support set* of lemma c , denoted $Su(c)$. A support set for lemma c is a set of lemmas $Su(c)$ such that the following formula:

$$\left(\bigwedge_{c_i \in Su(c)} c_i \right) \wedge T \wedge \neg c' \quad (4)$$

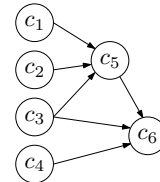


Fig. 1. Example support graph

is unsatisfiable. The conjunction of all lemmas in $Su(c)$ is sufficient to support pushing lemma c . In other words, if every lemma from $Su(c)$ is present in F_i then c can be added to F_{i+1} .

The support set of a lemma is not unique. For instance, any superset of a support set is also a support set. For the purposes of this paper, the support graph only contains a single support set of each lemma. It is possible to extend it to allow multiple support sets for each lemma, as discussed in section III-C. Figure 1 depicts an example support graph where $Su(c_5) = \{c_1, c_2, c_3\}$ and $Su(c_6) = \{c_3, c_4, c_5\}$.

B. Computing Support Sets

This subsection discusses how to integrate support set computation into Quip and IC3, towards the goal of learning a support graph. Three techniques are presented, and the tradeoffs between them are examined experimentally in section V. The techniques may vary in terms of the runtime expense they incur and the usefulness of the support graph they compute. In each of the presented approaches, a support set is computed as a result of an unsatisfiable consecution query of the form used in PUSHCLAUSES and shown below in Eq. 5.

$$F_k \wedge T \wedge \neg c' \quad (5)$$

When the formula in Eq. 5 is unsatisfiable, it means that a subset of the clauses of F_k form a support set for c . Indeed, this query effectively asks the question “is F_k a support set of c ?” As demonstrated in section V, in practice only a small number of lemmas from F_k are needed in the support set. The rest of this subsection presents various approaches to find smaller support sets using certificates of unsatisfiability for Eq. 5. They differ in the kind of certificate they use and how they integrate into Quip and IC3. The first type of certificate used is defined below.

Unsatisfiable core: A subset of the clauses in a SAT problem that are not mutually satisfiable.

The first approach uses an unsatisfiable core of Eq. 5. Unsatisfiable cores can be computed using well-known techniques [13], [14], and in practice are expected to be much smaller than the original problem. Every time the query is executed, a core is computed and clauses from F_k that are in the core are added to $Su(c)$. In practice, this tends to yield a much smaller support set, but can add significant runtime expense.

The next approach uses a different kind of certificate provided by an incremental SAT solver. Modern SAT solvers provide an incremental interface that supports solving with a set of assumptions in the form of unit literal clauses (*i.e.*, clauses with one literal). The next type of certificate comes from an incremental SAT solver and is defined below.

Critical assumptions: A subset of the assumptions in a SAT problem that are not mutually satisfiable, derived from the solver’s conflict trail [15].

The approach uses the critical assumptions from a modified version of Eq. 5 to compute smaller support sets. Each lemma c_i is granted a unique *activation literal* l_i , which is disjoined to c_i . As such, the lemma c_i is transformed to $c_i \vee l_i$. When performing a consecution query of the form $F_i \wedge T \wedge \neg c'$, an assumption $\neg l_i$ is added for each activation literal l_i . The resulting formula simplifies to that of

Eq. 5, but the critical assumptions indicate a support set. For each assumption $\neg l_i$ in the critical assumptions, the corresponding clause c_i is part of the support set. This approach is similar to a strategy used in practical IC3 implementations to answer consecution queries from different levels incrementally by having one activation literal per level [3]. In practice the extra assumptions may introduce slowdowns in the SAT solver, so using this approach on every consecution query may be computationally expensive.

The third approach avoids slowing down every consecution query. Numerous queries may occur in which the lemma c does not end up in the inductive trace and the added runtime to find a support set is wasted. To combat this, a dedicated query is used to compute the support set of a lemma immediately upon learning it, which occurs on line 15 of Algorithm 1. In addition, the consecution queries in Algorithm 2 are used to compute support sets as in the first two approaches. This avoids slowing down every consecution query, but adds extra queries. Effectively, it computes support sets on an as-needed basis: when a lemma is learned or when it is promoted to a higher level.

C. Multiple Support Sets

Over time, multiple support sets can be learned for a lemma. Since the support graph only holds one for each lemma, a mechanism to replace existing support sets may be needed. In our implementation, a support set is replaced by a newly-learned support set if and only if the size of the new set is less than or equal to that of the old one. This favors smaller support sets, as it is expected that they are more useful. When both sets are the same size, the newer support set is chosen. This is because new lemmas are learned over time, so an older support set may have lemmas that are no longer relevant.

One can imagine more advanced techniques that consider *e.g.*, the number of literals in the supporting lemmas or some heuristic measure of their value. Alternatively, the support graph could be modified to record all known support sets of each lemma. A topic of future work is to examine different replacement criteria and support graphs containing multiple support sets for each lemma.

IV. APPLICATIONS

This section presents two applications of the support graph. The first uses the support graph in incremental model checking to re-use parts of the inductive trace across calls to IC3 or Quip. This is applied in the context of IICTL [8]. It would also be applicable to other algorithms that use IC3 as a subroutine, or in a direct application of IC3 to solving multiple problem instances. The second application is within IC3 and Quip itself. The support graph is used to determine when lemmas can be pushed without a SAT query. Additionally, we propose using the support graph to develop heuristics to target important lemmas for aggressive pushing with may-proof obligations.

A. Incremental Model Checking

Unbounded model checking can be used in an incremental context. The model checker is called multiple times to prove different properties, possibly with different initial states. Results from earlier runs are re-used where possible to speed up later runs. However, without any further assumptions, it is not possible to re-use the inductive trace. This section presents a technique to increase re-use of the inductive trace in a commonly-used incremental model checking context.

Similar approaches exist, but they require additional processing or assumptions. The work of [16] presents an algorithm to find what portions of an inductive invariant hold after a change to the transition relation and the initial states. This can require several queries to a SAT solver. In the context of debugging, the work of [17] uses incremental model checking extensively while re-using the entire inductive trace. However, it assumes the initial states change in a specific way that is only applicable to the particular debugging problem it addresses.

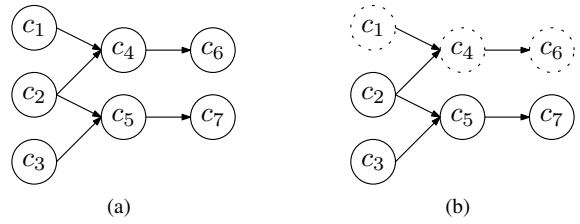


Fig. 2. (a) Original support graph (b) Labeled support graph

We propose using the support graph to speed up incremental model checking in the context of IICTL [8]. IICTL is an algorithm to check properties specified in computation tree logic (CTL). Other CTL-checking algorithms exist [18], but our technique is only applicable to IC3-based approaches. A full description of IICTL is beyond the scope of this paper. It accepts as input a tuple (S, I, T, P_{ctl}) where $S, I,$ and T are similar to the corresponding inputs to IC3. The CTL property P_{ctl} is a property specified in CTL. It may make calls to IC3 of the form $IC3(S, I_i, T, P_i)$ where I_i is a reachable state and P_i is a predicate specified in CNF. Lemmas are only promoted to F_∞ if they satisfy initiation with respect to the true initial states. In this mode of execution, lemmas in F_∞ can be saved across different runs of IC3 but the rest of the inductive trace must be discarded.

The support graph can be applied in this context to re-use other parts of the inductive trace. Between calls to the model checker, the support graph is saved. In a subsequent run, any lemmas in the support graph that have in-degree 0 and are not part of the given initial states are labeled as unsupported. These lemmas represent those that were part of I in a previous run of the model checker but are not in this run. Subsequently, any lemma with an unsupported lemma in its support set is also marked unsupported. This continues until the set of unsupported lemmas converges.

A new inductive trace is built from the lemmas that were not labeled as unsupported. An example is shown in Fig. 2. Fig. 2(a) shows the support graph constructed during a run where $I = c_1 \wedge c_2 \wedge c_3$. In Fig. 2(b), the algorithm is executed again with $I = c_1 \wedge c_2$. The lemmas that were supported by c_3 (with dotted outlines in the figure) are marked as unsupported. The lemmas c_5 and c_7 are placed in the inductive trace at their previous levels, but c_4 and c_6 are not. As demonstrated experimentally in section V, in practice this tends to save a large proportion of the lemmas.

B. Lemma Pushing

This subsection presents applications of the support graph to unbounded model checking in a non-incremental setting. Two changes to PUSHCLAUSES are proposed. Though Quip uses a slightly different pushing algorithm, for simplicity we present the changes as a modification to Algorithm 2. Similar changes apply in Quip. The first change uses the support graph to avoid calling the SAT solver. The second identifies lemmas to push using may-proofs. Both changes can be applied in Quip, but IC3 can only use the former.

The proposed changes to PUSHCLAUSES are shown in Algorithm 3. On line 3, the support graph is used to push the lemma. This avoids a SAT query if the support graph indicates that the lemma is supported. Note that if the support graph indicates that c is supported, the checks on lines 5 or 7 would also indicate that fact. Using the support graph is simply an optimization to avoid using the SAT solver. This can be implemented in both Quip and IC3.

The second change is on line 9. Given a heuristic function $h(c)$ which returns a Boolean, a may-proof obligation is enqueued for the lemma if $h(c)$ is true. Intuitively, $h(c)$ should indicate whether or not the lemma is valuable. In the next iteration, the algorithm will try to push the valuable lemmas forward. One can imagine various

TABLE I
COMPARISON OF SUPPORT GRAPH COMPUTING TECHNIQUES

benchmark	Quip	Activation Literals			UNSAT Core			Hybrid		
	time	time	average	median	time	average	median	time	average	median
pdtpvisbakery2	118	-	-	-	3.0x	8	8	1.1x	8	8
bjrb07amba5	95.4	2.9x	19	12	1.4x	22	13	1.0x	17	9
bob3	69.7	5.7x	17	10	3.7x	17	8	0.6x	23	5
6s164	62.8	-	-	-	3.2x	21	12	2.4x	17	8
pdtpmscoherence	44.9	8.9x	136	57	5.4x	80	39	1.9x	63	21
mentorbm1p05	32.6	3.4x	2	2	1.1x	1	1	0.6x	2	2
6s306rb03	19.7	2.5x	1	2	1.0x	1	1	1.2x	1	2
pdtpmsam2901	8.86	1.1x	6	3	1.5x	6	2	2.0x	12	3
eijkbs4863	5.78	2.5x	7	3	1.9x	8	3	1.6x	31	2
pj2003	4.76	1.4x	1	1	1.1x	1	1	0.9x	1	1
MEAN		2.34x	14		1.54x	11.7		1.22x	11.8	

Algorithm 3 SGPUSHCLAUSES()

```

1: for  $i = 1$  to  $k - 1$  do
2:   for all clauses  $c$  in  $F_i$  do
3:     if  $c$  is supported at a level  $j > i$  then
4:       ADDLEMMA( $c, j$ )
5:     else if  $\neg \text{SAT}(F_\infty \wedge T \wedge \neg c')$  then
6:       ADDLEMMA( $c, \infty$ )
7:     else if  $\neg \text{SAT}(F_i \wedge T \wedge \neg c')$  then
8:       ADDLEMMA( $c, i + 1$ )
9:     else if  $h(c)$  then
10:      Enqueue( $Q, (\neg c, i + 1, \text{may})$ )
11:   if  $F_i = F_{i+1}$  then return PROOF
12: return NO_PROOF

```

heuristic criteria. The next section presents experiments considering two different heuristics. The first heuristic function returns true when the out-degree is greater than some constant threshold. This represents an attempt to prioritize lemmas that support many other lemmas. The second $h(c)$ returns true if and only if the out-degree of c is at least double its in-degree. The rationale behind this heuristic is that a lemma with low in-degree will be easier to push forward.

V. EXPERIMENTAL RESULTS

All results presented in this section are run on a single core of an i5-3570K 3.4 GHz workstation with 16GB of RAM. The proposed enhancements are added in our own implementation of Quip, which is built on IImc [19]. The Quip solver is also able to handle reachability queries in IICTL, which is included in IImc. Experiments are timed out after ten minutes. Results are presented for HWMCC'15 circuits.

A. Computing Support Graphs

This section presents results for the computation of support graphs. Table I shows a comparison of the presented techniques for a selection of HWMCC'15 circuits. The first column shows the name of the circuit. The second shows the runtime of Quip without support graphs. The next three columns show the runtime, average number of lemmas in the support set of each lemma, and the median number for the activation literal method. The next six columns show similar data for the UNSAT core method and the hybrid method. The bottom row shows the mean across a larger set of 25 HWMCC'15 circuits that is used throughout this section.

Fig. 3 presents histograms of the support set size for all lemmas for bjrb07amba5 for each methodology. Fig. 3(a) shows the UNSAT core method, Fig. 3(b) shows the activation literal method, and Fig. 3(c) shows the hybrid method. The results are representative of the entire set. The mode of 2 observed in the histogram, and in fact is the mode across the entire set of circuits. One can

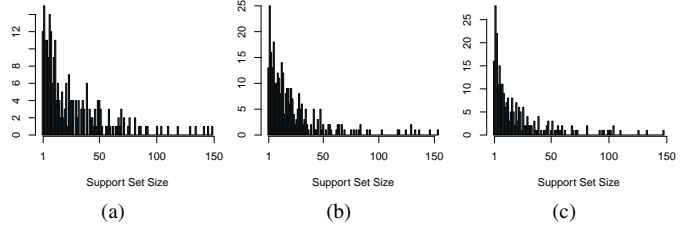


Fig. 3. Histogram of support set size for each methodology bjrb07amba5

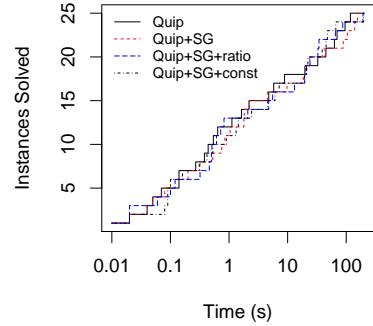


Fig. 4. Instances solved using each method

conclude that most lemmas have very small support sets, but a few outliers exist with very large support sets.

Overall, the UNSAT core method seems to offer better runtime performance than the activation literal method but produces larger support sets. The hybrid approach, which uses specialized queries to compute support sets in addition to the consecution queries in PUSHCLAUSES can use UNSAT cores or activation literals internally. The implementation presented here uses the activation literal method as it appears to find smaller support sets. The hybrid methodology appears to offer the best runtime performance, with comparable performance in terms of support set size. Across all circuits, the hybrid methodology induces a 22% slowdown. All experiments presented later in this section are based on the hybrid methodology.

B. Quip Runtime

This subsection presents results for a modified implementation of Quip that uses the support graph as described in section IV-B. Quip is modified to use Algorithm 3 in place of PUSHCLAUSES. Two versions are implemented, one using the constant threshold heuristic and one using the ratio heuristic. Figure 4 shows the number of problem instances solved over time by each method versus time across 25 HWMCC'15 circuits.

TABLE II
RUNTIME COMPARISON OF PUSHCLAUSES HEURISTICS

Quip	Quip+SG	Quip+SG+Ratio	Quip+SG+Const
1.0x	1.22x	1.14x	1.20x

It can be seen that all of the approaches have fairly similar runtime performance. Overall, the expense of computing the support graph is greater than the gains that are achieved with these simple heuristics. Table II summarizes the results over the 25 circuits. The four columns show the geometric mean runtime relative to Quip for Quip alone, Quip with support graph computation alone, Quip using the ratio heuristics, and Quip using the constant threshold heuristic. It can be seen that the ratio heuristic offers better performance than the constant threshold. Additionally, neither heuristic gives enough benefit to outweigh the cost of computing the support graph. However, the results motivate future research into better heuristics and into targeted support graph computation. The expense of computing the support graph can be reduced by computing it only for certain lemmas. For instance, it may be worthwhile to target the property lemma and its support set instead of every lemma. Additionally, by identifying better heuristics it may be possible to realize substantial speedups. Even with these simple heuristics, it would be possible to achieve speedups if the support graph could be computed with less runtime overhead.

C. Incremental Quip

This subsection presents results regarding the technique proposed in section IV-A applied in IICTL. HWMCC'15 designs are simple safety checking problems, and therefore do not have CTL properties suitable for IICTL. To obtain the benchmarks, a resetability property of the form $AG\ EF\ I$ is generated for each circuit. This property requires that every reachable state can reach an initial state. The experiment is intended to demonstrate that lemmas can be re-used in an incremental setting. These properties are suitable as they involve reachability queries. The heuristics in PUSHCLAUSES are not used so as to isolate the effects of saving lemmas.

Results are presented for IICTL, IICTL with support graph computation, and IICTL with support graph computation and lemma re-use. Table III shows results for a selection of circuits. The selected circuits are those where IICTL generated multiple queries to Quip that resulted in learning 50 or more lemmas. The first column shows the name of the benchmark. The second shows the runtime of IICTL in seconds, while the third shows the time taken for normal IICTL with support graph computation. The next four show the speedup, sum of lemmas saved, sum of lemmas known, and percentage saved using the support graph. It can be seen that many lemmas are saved, often 97% or more. The results demonstrate the potential of the approach and motivate further research. In particular, saving lemmas seems to improve runtime, as doing so costs a 1.14x slowdown versus 1.18x for computing the support graph alone.

VI. CONCLUSION AND FUTURE WORK

This paper introduces the notion of a support graph in the model checking algorithms of IC3 and Quip. It presents techniques to compute and use support graphs. Two applications are considered. The first is to speed up model checking by using the support graph to avoid satisfiability queries and to identify important lemmas. The simple heuristics presented are effective at identifying valuable lemmas, as they are able to partially mitigate the runtime overhead of computing a support graph. The second application is to increase the re-use of lemmas in incremental model checking. Topics of future work include developing better heuristics, considering support graphs

TABLE III
LEMNAS SAVED IN IICTL

benchmark	No-SG	SG	SG + Save Lemmas		
	time	time	time	sv/tot	%
shift1add256	135	1.01x	0.92x	890/1755	51%
eijks208o	0.86	1.34x	1.47x	677/680	99%
ndista128	1.18	1.02x	1.34x	442/457	97%
eijks4863	0.61	0.98x	1.02x	0/236	0%
nusmvsyncarb10	0.6	1.67x	1.00x	154/158	97%
pdtpmcoherence	2.14	1.42x	1.42x	0/54	0%
eijks820	0.05	1.00x	1.00x	0/52	0%
TOTAL				2163/3392	64%
MEAN		1.18x	1.14x		

that identify multiple support sets of each lemma, and identifying means of targeting lemmas for support graph computation.

REFERENCES

- [1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.
- [2] A. Bradley, "Sat-based model checking without unrolling," in *Intl Conf. on Verification, Model Checking, and Abstract Interpretation*, 2011.
- [3] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, TX: FMCAD Inc, 2011, pp. 125–134.
- [4] A. R. Bradley, *Understanding IC3*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–14.
- [5] —, "Incremental, inductive model checking," in *2013 20th International Symposium on Temporal Representation and Reasoning*, Sept 2013, pp. 5–6.
- [6] A. Cimatti and A. Griggio, *Software Model Checking via IC3*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 277–293.
- [7] T. Lange, M. R. Neuhauber, and T. Noll, "Ic3 software model checking on control flow automata," in *2015 Formal Methods in Computer-Aided Design (FMCAD)*, Sept 2015, pp. 97–104.
- [8] Z. Hassan, A. R. Bradley, and F. Somenzi, "Incremental, inductive CTL model checking," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012, pp. 532–547.
- [9] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, Oct 2011, pp. 144–153.
- [10] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, Nov 2007, pp. 173–180.
- [11] A. R. Bradley, "k-step relative inductive generalization," *CoRR*, vol. abs/1003.3649, 2010.
- [12] A. Ivrii and A. Gurfinkel, "Pushing to the top," in *2015 Formal Methods in Computer-Aided Design (FMCAD)*, Sept 2015, pp. 65–72.
- [13] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '10. Austin, TX: FMCAD Inc, 2010, pp. 221–229.
- [14] J. Huang, "Mup: A minimal unsatisfiability prover," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '05. New York, NY, USA: ACM, 2005, pp. 432–437.
- [15] J. P. Marques-Silva and K. A. Sakallah, "Grasp: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, May 1999.
- [16] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, TX: FMCAD Inc, 2011, pp. 135–143.
- [17] R. Berryhill and A. Veneris, "A complete approach to unreachable state diagnosability via property directed reachability," in *Proceedings of the 2016 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '16, 2016.
- [18] K. L. McMillan, *Applying SAT Methods in Unbounded Symbolic Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 250–264.
- [19] A. R. Bradley and F. Somenzi and Z. Hassan, "Iimc: an Incremental Inductive model checker." [Online]. Available: <https://github.com/mgudemann/iimc>