Efficient Modular SAT Solving for IC3

Sam Bayless*, Celina G. Val*, Thomas Ball†, Holger H. Hoos*, Alan J. Hu*

*University of British Columbia, {sbayless, vcelina, hoos, ajh}@cs.ubc.ca
†Microsoft Research, tball@microsoft.com

***Abstract*—We describe an efficient way to compose SAT solvers into chains, while still allowing unit propagation between those solvers. We show how such a "SAT Modulo SAT" solver naturally produces sequence interpolants as a side effect — there is no need to generate a resolution proof and post-process it to extract an interpolant. We have implemented a version of IC3 using this SAT Modulo SAT solver, which solves both more SAT instances and more UNSAT instances than PDR and IC3 on each of the 2008, 2010, and 2012 Hardware Model Checking Competition benchmarks.**

***Index Terms*—SAT, IC3, PDR, Interpolants**

I. Introduction

SAT solvers play a central role in many hardware and software model-checking techniques. In this paper, we introduce three inter-dependent contributions, culminating in an improved state-of-the-art model-checker. First, we describe a way to compose multiple SAT solvers into chains and trees, in order to efficiently solve problems that have an underlying "modular" structure (for example, instances produced by unrolling a transition function). We show that this technique can be thought of as a nested SAT Modulo Theory (SMT) solver, and that we can apply techniques from lazy SMT solvers to improve the performance of this "SAT Modulo SAT" solver. Our nested SAT solver provides a general-purpose way to take advantage of locality while solving a CNF with (known) structure.

Secondly, we show that our SAT Modulo SAT solver produces sequence interpolants [5], [21], by extending previous work by Chockler et al. [6]. These sequence interpolants are produced without requiring explicit proof-traces.

Our third contribution is to demonstrate that our SAT Modulo SAT solver can be useful in practice, by implementing a variant of IC3 [4] using it (and, implicitly, the sequence interpolants we produce).[1] We show that the resulting model checker outperforms both IC3 and PDR [11] on the 2008, 2010, and 2012 Hardware Model Checking Competition benchmarks.

II. Modular SAT Solvers

Given a *partitioned CNF* formula $\phi_0, \phi_1, \dots \phi_n$, where each ϕ_i is a set of clauses, the *partitioned Boolean satisfiability problem* consists of determining the satisfiability of $\bigcup_{i=1}^{n} \phi_i$. Here, we will consider only cases where the partitioning into clause sets is explicitly specified or can be observed directly from the underlying problem. We will refer to the clause sets $\phi_0, \phi_1, \dots$ as *modules*, and to any SAT solver that is designed to solve such a partitioned CNF, as a *modular SAT solver*.

Obviously, to solve a partitioned CNF one could simply merge all the partitions and solve the resulting CNF using a standard SAT algorithm, but doing so loses any structural information that might have been present in the partitioned CNF. Real-world problems often possess a high degree of modular structure (e.g., formulas derived from real-world circuits or software), so this structural information may be useful. An approach that has been investigated widely in the literature is to find the variables that are shared between modules (we will refer to these as *interface variables*), and to assign them first. Because the partitions ϕ_i are independent of each other under any complete assignment to these variables, each module can then be solved independently [17] (and in parallel [15]). Unfortunately, this method requires a potentially exponential number of assignments to the interface variables to be tested. Alternatively, the interface variables can simply be used to inform a static decision heuristic. Many strategies for partitioning a CNF have been investigated for this latter approach (e.g., [1], [9], [13]).

In this paper, we describe a new modular SAT algorithm. This algorithm relies upon three existing capabilities of typical incremental SAT solvers (such as MiniSat [10] and PicoSat [2]), namely:

1) Incremental SAT solvers allow for a CNF to be solved repeatedly as new clauses are added (maintaining heuristic values and learned clauses between runs).
2) They allow for the *temporary* addition (and subsequent removal) of unit clauses in the CNF. Equivalently, they allow for the CNF to be solved repeatedly under the temporary assumption of different partial assignments.
3) When the CNF is not satisfiable under such a partial assignment, they can return a concise clause over just the assumed unit clauses that 'explains' why those units cannot be mutually true in the CNF. This clause will include only variables that are common to both the assumed unit clauses and to the CNF being solved under the assumption.

A simple, recursive algorithm is shown in Alg. 1. To our knowledge, we are the first to propose solving a general partitioned CNF in this way; however, this algorithm is very closely related to several other approaches, as we will discuss below. Subsequently, we will build on this algorithm and arrive

[1]Just to forestall a potential point of confusion: Though we apply some techniques from lazy SMT solvers, we are not extending IC3 to handle theories other than SAT. This has been done (see, e.g., [7]), but is orthogonal to our contribution here. Our use of SMT techniques is instead to directly speed up the core Boolean satisfiability reasoning of IC3.

Algorithm 1 (Unoptimized) Modular SAT Solver

Input: Partial assignment α_i, set of clauses ϕ_i.
Output: TRUE if $\bigcup_{j=0}^{i} \phi_j$ is SAT under α_i, else a conflict-clause which is inconsistent with α_i and contains only variables common to α_i and $\bigcup_{j=0}^{i} \phi_j$.
//The (initially empty) sets of conflict-clauses L_{ϕ_i} maintain
//the invariant $\bigcup_{j=0}^{i-1} \phi_j \Rightarrow L_{\phi_i}$.
function MODULARSOLVE(α_i, ϕ_i)
  loop
    if $\phi_i \cup L_{\phi_i} \cup \alpha_i$ is SAT **then**
      $\alpha_{i-1} \leftarrow$ satisfying assignment to $\phi_i \cup L_{\phi_i} \cup \alpha_i$
      if $i = 0$ **then**
        return TRUE
      else
        $c \leftarrow$ MODULARSOLVE(α_{i-1}, ϕ_{i-1})
        if c = TRUE **then**
          return TRUE
        else
          $L_{\phi_i} \leftarrow L_{\phi_i} \cup \{c\}$
    else
      $c \leftarrow$ conflict-clause for $\phi_i \cup L_{\phi_i} \cup \alpha_i$
      return c
  end loop

at a new, improved method: our SAT Modulo SAT solver, described in Alg. 3.

Alg. 1 operates progressively over the modules, first attempting to solve module ϕ_n, which will either be unsatisfiable, or will provide us with an assignment α_{n-1} (Figure 1). It then recursively solves ϕ_{n-1} under assignment α_{n-1}. Note that while α_{i-1} is a complete assignment to ϕ_i, it may be a partial assignment to ϕ_{n-1}.

If a module ϕ_i cannot be satisfied under α_i, the incremental SAT interface produces a learned clause c over the variables in α_i. We will refer to such a clause as an *interface clause*. We add this interface clause c into the set L_{ϕ_i}, which will be conjoined with ϕ_i when solving it in all subsequent iterations. With the addition of c, the conjunction $\phi_i \wedge L_{\phi_i}$ will now either be unsatisfiable (in which case we are done), or force the solver into a new solution with a different assignment to the interface variables.

Correctness of Alg. 1 follows from straightforward induction on i; termination is guaranteed, because L_{ϕ_i} is strengthened at every call (unless a satisfying assignment is found, in which case the algorithm terminates and returns TRUE).

One feature of Alg. 1 is that it expects an ordering over the modules. The specific ordering chosen is effectively a heuristic, and in some cases a good choice may be obvious from the problem context (*e.g.*, bounded model checking); the algorithm is correct even if the order is chosen arbitrarily (though this would impact the meaning of the interpolants produced by the algorithm, examined in the next section), and is also correct if the ordering is changed dynamically at runtime. Though we do not explore it here, one could consider randomly permuting the order, or applying a dynamic heuristic to adjust the order as the algorithm proceeds, rather

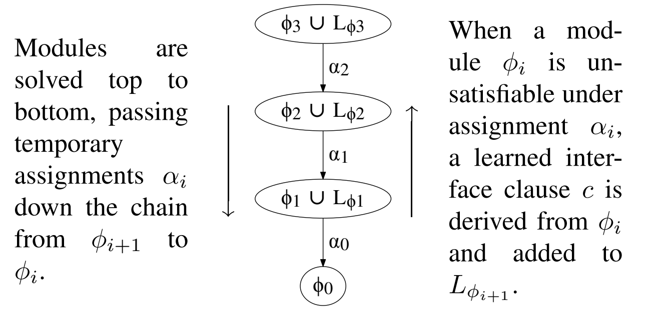

Fig. 1: A chain of four SAT Modules.

than relying upon a static ordering. We also observe that Alg. 1 can be trivially extended to operate over a tree-ordering of the modules: instead of MODULARSOLVE(α_i, ϕ_i) recursively calling just MODULARSOLVE(α_{i-1}, ϕ_{i-1}), it would make a recursive call for each of its children. Alg. 1 is related to several recent algorithms. For example, in the case of exactly two modules, Alg. 1 is equivalent to (the simplest case of) the proofless interpolant computing algorithm introduced in [6], and it also resembles typical all-SAT procedures [12] and some 2QBF solvers [18]. We can also think of this algorithm as forming a nested series of counter-example-guided abstraction-refinement loops: each $\phi_i \cup L_i$ is an abstraction of its conjunction with the modules below it, and the learned interface clauses returned from later modules serve to refine that abstraction by eliminating spurious counter-examples.

If we obtain the modules by unrolling a transition function (with one module per time step), then Alg. 1 is roughly equivalent to a simplified, slightly re-organized version of the recursive cube-blocking procedure at the core of IC3 and PDR; in Section III, we will examine this connection to IC3 in more detail. However, the modular SAT solver we have described here, and the proof above, is general to the case where the partitions ϕ_i are not all copies of the same transition function.

A. SAT Modulo SAT

Alg. 1 has the benefit that it can be implemented directly using the incremental interface exposed by typical SAT solvers without any modifications. Unfortunately, in practice it performs poorly, because unit propagation between modules is delayed until each previous module is completely solved, and many learned clauses must be passed up the chain to eliminate parts of the search space that would normally have been pruned by unit propagation alone (if we were solving the complete conjunction of the modules). In the worst case, this can lead to an exponential slow-down, as an exponential number of solutions from ϕ_1 might need to be produced before finding one that satisfies ϕ_2.

Lazy SAT Modulo Theory (SMT) solvers [19] face many of the same challenges as our naïve modular SAT solver above, and we can adopt the mechanisms they use to address these challenges by observing that Boolean satisfiability is itself an ideal candidate to be a theory in a lazy SMT solver. Instead of first finding a complete satisfying assignment α_i to ϕ_i

and then solving ϕ_{i-1} under it, we modify Alg. 1 to eagerly perform unit propagation on ϕ_{i-1} (and ϕ_{i-2}, etc.) as the partial assignment to ϕ_i is being constructed, returning a conflicting interface clause as soon as the partial assignment of ϕ_i would lead to a conflict in ϕ_{i-1} or a lower module.

In Alg. 2 and 3, we describe in detail the changes needed to convert a typical incremental SAT solver into an efficient modular SAT solver using this eager unit propagation across modules. In the interest of space, we will assume the reader is familiar with MiniSat [10] and describe the necessary modifications in reference to that implementation.

To apply this eager unit propagation across modules efficiently, we introduce a new method, PROPAGATEALL, which first applies unit propagation locally on the current module ϕ_i (by calling PROPAGATE) and then recursively propagates any resulting assignments to the interface variables in the next module, ϕ_{i-1}. If this leads to a conflict, then MiniSat's ANALYZEFINAL method returns a conflict clause over the interface variables.

If unit propagation of the interface assignments is successfully applied to ϕ_{i-1}, then we check if that unit propagation assigned any new literals on the interface between ϕ_i and ϕ_{i-1}. If any such assignments were made, then we again propagate those assignments locally, and continue in this way passing assignments back and forth between adjacent modules until we reach a fixed point or a conflict.

In order to accomplish this eager unit propagation efficiently, we make one additional change. When a literal is propagated, CDCL SAT solvers store a reference to the clause that was unit, so that they can explore it later during conflict analysis. If an assignment is made to the interface by ϕ_{i-1} in Alg. 2, then we would not actually have any such clause in the SAT solver for ϕ_i to use as a reason for this assignment. We can create such a clause on the interface variables by calling MiniSat's ANALYZEFINAL, but rather than call this eagerly after each unit propagation from ϕ_i, we instead set the literal's "reason" to a temporary placeholder value.

MiniSat accesses these reason references only during conflict analysis, using a helper method, REASON. We modify REASON to check for that placeholder value, and replace it with a clause produced by calling ANALYZEFINAL on ϕ_{i-1}. In this way we create the reason clauses for units propagated from ϕ_{i-1} to ϕ_i only if needed by conflict analysis (efficient SMT solvers take a similar approach).

Finally, we modify the main CDCL loop (Alg. 3) in two ways. First we alter it to call PROPAGATEALL instead of PROPAGATE. Second, once ϕ_i is entirely assigned, we modify it to recurse on ϕ_{i-1}.

Applying unit propagation eagerly allows the SAT solver for each module to prune its search space early, while the lazy reason construction reduces the number of trivial interface clauses that would otherwise have to be learned gradually and passed up through the chain of modules. Taken together, we refer to the modular SAT solver using these SMT-inspired optimizations in Algs. 2 and 3 as a "SAT Modulo SAT" solver.

Algorithm 2 PROPAGATEALL method applies intra-module and inter-module unit propagation. Note that we rely upon a list of assigned literals, $trail_{\phi_i}$, maintained for each module between calls.

```
function PROPAGATEALL(φ_i)
    loop
        // Call unit propagation on φ_i
        c ← PROPAGATE(φ_i)
        if c is a clause then
            return c    // c is a learned clause
        else if i=0 then
            return TRUE
        // Collect all new assignments to interface variables
        if trail_{φ_i} \ trail_{φ_{i-1}} = ∅  then
            return TRUE
        // Propagate new assignments in φ_{i-1}
        for all l ∈ (trail_{φ_i} \ trail_{φ_{i-1}}) do
            ENQUEUE_{i-1}(l)
        if PROPAGATEALL(φ_{i-1}) ≠ TRUE then
            c = ANALYZEFINAL(trail_{φ_i}, φ_{i-1})
            L_i ← L_i ∪ {c}
            ADDCLAUSE(c)
            return c
        else if (trail_{φ_i-1} \ trail_{φ_i}) = ∅ then
            return TRUE // No new interface assignments
        else
            // Propagate new assignments from φ_{i-1} in φ_i
            for all l ∈ (trail_{φ_i-1} \ trail_{φ_i}) do
                ENQUEUE_i(l)
                // Mark reason for lazy construction
                reasons[var] ← 'LazyPlaceholder_{i-1}'
    end loop

function REASON(var)
    if reasons[var] = 'LazyPlaceholder_{i-1}'  then
        c ← ANALYZEFINAL(var,φ_{i-1})
        L_i ← L_i ∪ {c}
        ADDCLAUSE(c)
        reasons[var] ← c
    return reasons[var]
```

B. Interpolants as Side Effects

Interpolants [16] form a core part of many recent SAT-based model checkers, including IC3. Normally, interpolants are constructed by analyzing a resolution proof-trace, which must be generated by a SAT solver as it is solving an instance. This introduces an overhead into the solving process (for this reason, recent work ([6], [20]) has investigated alternative methods that do not require constructing an (explicit) proof trace).

We now show that the sets of learned interface clauses L_{ϕ_i} collected between each module in Alg. 1 form valid interpolants. Taken together, these successive interpolants form a sequence interpolant [21]. An alternative proof for the case of exactly two modules can be found in [6]. For simplicity, we

Algorithm 3 The main CDCL loop of our SAT Modulo SAT solver, using the PROPAGATEALL method. We integrate the recursive call to the next solver directly into the search loop. Other than the changes here and in the PROPAGATEALL method, our implementation follows MiniSat 2.2 [10]. Alg. 3 is a direct replacement for Alg. 1.

```
function MODULARSOLVE(α_i, φ_i)
    loop
        conflict ← PROPAGATEALL(φ_i)
        if conflict is a clause then
            if DECISIONLEVEL( ) = 0 then
                return conflict
            c ← ANALYZE(conflict)
            BACKTRACK( )
            ADDCLAUSE(c)
        else
            if exists an unassigned lit ∈ α_i then
                l ← lit
            else
                l ← PICKBRANCHLIT( )
            if l = NULL then
                // trail_{φ_i} is a satisfying assignment to φ_i
                if i = 0 then
                    c ← TRUE
                else
                    c ← MODULARSOLVE(trail_{φ_i}, φ_{i−1})
                if c = TRUE then
                    return TRUE
                else
                    // Learn clause c from φ_{i−1}
                    L_i ← L_i ∪ {c}
                    BACKTRACK( )
                    ADDCLAUSE(c)
            else
                NEWDECISIONLEVEL( )
                ENQUEUE_i(l)
    end loop
```

describe our proof in terms of the unmodified Alg. 1, but it holds equally well for the optimized SAT Modulo SAT solver.

Given a CNF partitioned into two parts, ϕ_A and ϕ_B, with $\phi_A \cup \phi_B$ unsatisfiable, an interpolant between ϕ_A and ϕ_B is any set of constraints I with the following three properties:

1) ϕ_A implies I.
2) $I \cup \phi_B$ (i.e., the conjunction of the constraints) is unsatisfiable.
3) I contains only variables common to ϕ_A and ϕ_B.

First, consider Alg. 1 with only two modules. On an unsatisfiable instance, Alg. 1 terminates only when the top-most module ϕ_1, combined with the interface clauses L_{ϕ_1} it has learned from module ϕ_0, does not have any satisfying solutions. So at termination (on an unsatisfiable instance), $\phi_1 \cup L_{\phi_1}$ must be unsatisfiable. We also have that $\phi_0 \Rightarrow L_{\phi_1}$, because L_{ϕ_1} consists only of clauses implied by ϕ_0. Finally, the incremental SAT solver interface guarantees that each clause in L_{ϕ_1} contains only variables that are common to ϕ_1 and ϕ_0. These three conditions together satisfy the definition of an interpolant between ϕ_1 and ϕ_0.

Next, consider an unsatisfiable chain of three modules, ϕ_2, ϕ_1, and ϕ_0. There are *two* interpolants that are constructed by Alg. 1: An interpolant L_{ϕ_2} between ϕ_2 and $(\phi_1 \wedge \phi_0)$, and an interpolant L_{ϕ_1} between $(\phi_2 \wedge \phi_1)$ and ϕ_0.

In this three module chain, the argument that L_{ϕ_1} forms an interpolant is the same as above. The argument that L_{ϕ_2} forms an interpolant is similar, except that the clauses collected in L_{ϕ_2} are implied by the conjunction $\phi_1 \wedge \phi_0$, rather than by ϕ_0 alone. This is the case even though in Alg. 1 module ϕ_2 is only ever passed interface clauses constructed by ϕ_1 (and never by ϕ_0), because module ϕ_1 may itself have been passed interface clauses from module ϕ_0, and may then have derived new constraints based on those facts that are subsequently passed to module ϕ_2.

In general, at termination on an unsatisfiable instance, it must either be the case that $\phi_n \wedge \phi_{n-1} \wedge \ldots \wedge \phi_i$ is by itself already unsatisfiable (in which case L_{ϕ_i} is the empty set, and a trivial interpolant), or that $\phi_n \wedge \phi_{n-1} \wedge \ldots \wedge \phi_i \wedge L_{\phi_i}$ is unsatisfiable, in which case L_{ϕ_i} is a valid interpolant between the conjunctions $\phi_n \wedge \ldots \wedge \phi_i$ and $\phi_{i-1} \wedge \ldots \wedge \phi_0$.

III. IC3 USING SAT MODULO SAT

The modular SAT solver we have described here operates on an ordered sequence of CNF modules; a natural use case would be to apply it to bounded model checking [3] by constructing one module per time step, and incrementally adding new modules as time steps are added. Unfortunately, performance is roughly competitive with, but not better than, an (unsophisticated) incremental bounded model checker. However, simple bounded model checking does not take advantage of the sequence interpolants that our solver naturally produces.

Sequence interpolants are not typically generated by themselves as an end goal. Instead, the primary place that sequence interpolants are used is as a component of model checking algorithms (*e.g.*, [5], [21]), most prominently in the current state-of-the-art SAT-based unbounded model checker, IC3 [4]. In IC3, sequence interpolants are created implicitly, through an incremental refinement process that is closely related to the unoptimized modular SAT solver from Alg. 1.

We now demonstrate that the SAT Modulo SAT solver we presented above is useful in practice by creating a version of IC3 based on it and the sequence interpolants it produces.

Our implementation closely follows the PDR [11] variant of IC3, which we do not have space to recount in full. We will assume the reader is familiar with PDR, and describe only our changes here. Modifying PDR's algorithms to use the modular SAT solver will entail some non-trivial changes, which we describe below. As well, while building our solver, we developed some minor improvements to the general IC3 algorithm; we will show below that these minor changes are indeed improvements, but that the most important performance improvement is due to our SAT Modulo SAT solver.

Algorithm 4 The cube-blocking procedure for the stack-based variant of IC3, using a modular SAT solver. Notice that the stack is actually completely eliminated; recursively blocking the cube is directly handled by the modular SAT solver (MODULARSOLVE calls either Alg. 1 or Alg. 3 above). In contrast to IC3, all the newly generated blocking clauses are collected and generalized at the end.

```
function ModularBlockCube(TCube s0)
    i ← s0.frame−1
    if ModularSolve(s0.cube, φ_i) then
        return False // Counter-example found
    else
        CollectAllClauses(i)
        return True

function CollectAllClauses(t)
    // Collect new interface clauses from the first t solvers
    // We assume these are stored in vectors
    // newInterfaceClauses_i for each frame
    for i ← 1 . . . t − 1 do
        for all Clause c ∈ newInterfaceClauses_i do
            markSolver(i) // Needs clause propagation
            c ← generalize(c)
            // Attempt to propagate c forward until it fails
            j ← EagerPropagateClause(c, i)
            F[j].add(c)
        newInterfaceClauses_i ← ∅

function EagerPropagateClause(Clause c, from)
    // Propagate clause c forward as far as we can
    for i ← from . . . F.size()−1 do
        if not propagateClause(c, i) then
            return i
        return i
```

The central part of IC3 is the cube-blocking procedure (in PDR, "RECBLOCKCUBE"). There are two major variants of this procedure. The simpler, 'stack-based' version takes an assignment to the flops (a cube) that is known to lead to the negated property, and incrementally strengthens the interpolants between each time frame until they are sufficient to block that cube in the last time frame. In Alg. 4, we show how we can use a modular SAT solver (either Alg. 1 or Alg.3) to replace RECBLOCKCUBE . Intuitively, cube-blocking in the stack-based variant of IC3 is performing *almost* the same function as the simple recursive modular SAT solver of Alg. 1, with a few extra steps added. By re-arranging this code to separate out the part that closely matches Alg. 1 we then make it possible to replace it with the more complicated SAT Modulo SAT solver in Alg. 3 as well.

The match is not exact. The most obvious difference is that IC3 applies inductive generalization [4] to drop literals from conflict clauses as they are added to the interpolants. Unfortunately, the solvers for each time step are maintaining state between calls in the modular SAT solver, which would be overwritten during inductive generalization. One way to

Algorithm 5 The cube-blocking procedure for the priority-queue based version of IC3 using a modular SAT solver. This function is a replacement for the RECBLOCKCUBE procedure of PDR. We show here the *keepCubes* option discussed below. With *keepCubes* set, we keep the last time frame's TCubes in the priority queue for the next iteration rather than discarding them (as PDR does).

```
function ModularBlockCubePriority(TCube s0)
    Q.add(s0)
    while Q.size( ) > 0 &&
            Q.peek( ).frame < F.size( )  do
        TCube s ← Q.popMin( )
        if not isBlocked(s) then
            if not ModularBlockCube(s) then
                return False // Counter-example found
            else
                Q.add(CollectAllCubes(s.frame))
                if keepCubes || s.frame < F.size() -1 then
                    Q.add(TCube(s.cube, s.frame + 1))
        else if keepCubes && s.frame < F.size() -1 then
            Q.add(TCube(s.cube, s.frame + 1))
    return True

function CollectAllCubes( t)
    // Collect all satisfying assignments to the flops
    // found during ModularSolve. We assume these
    // were stored for frame i in vector flopAssignments_i.
    for i ← 1 . . . t − 1 do
        for each assignment a ∈ flopAssignments_i do
            Q.add(TCube(a, i + 1))
        flopAssignments_i ← ∅
```

resolve this would be to keep an extra SAT solver, not part of the SAT modulo SAT solver, and use that to apply inductive generalization as conflict clauses are learned. This would allow us to apply inductive generalization at the same point as IC3, at the cost of extra memory usage. A second option, which we take in Alg. 4, is to delay inductive generalization until after the complete call to the modular SAT solver (during which many interface clauses may have been learned), and then subsequently apply inductive generalization to each new clause. This allows us to re-use the solvers from our modular SAT solver for generalization.[2]

Another difference is that one of the original selling points of IC3 was that it does not require the transition function to be unrolled; instead, a growing set of sequence interpolants (with some special properties discussed below) are maintained by re-using a single transition function between consecutive interpolants in the sequence. By instantiating a separate copy

[2] Another subtlety is that when we apply inductive generalization to a clause from module ϕ_i, we re-use the SAT solver for ϕ_i from our modular solver, but call its normal, non-modular SOLVE method (which does not recursively solve the other modules in the chain). An alternative option would be to use the entire modular SAT solver chain during generalization, which would increase the chance of dropping literals from the conflict clauses, but at the cost of introducing an additional linear time factor (in the number of modules) into generalization.

of the transition function for each module ϕ_i in our modular SAT solver, we are giving up this near-constant memory usage. However, recent versions of PDR have made the same time-space trade-off, to avoid the cost of tracking which learned clauses correspond to which time frame.

A more substantial difference between our SAT modulo SAT solver and IC3 is that IC3 requires the interpolants for each time frame in the sequence to be constructed from a subset of the clauses that make up the interpolant for the previous time frame. We cannot combine the trick IC3 usually applies for this with Alg. 3, and must instead add a non-deterministic self-loop to the transition function (by adding an extra input to the circuit that, when true, forces the flops to their reset state). This extra non-determinism might be expected to slow down the SAT solver.[3] However, because our solver (like IC3) always solves its time frames in reverse order, the self-loop will always be disabled by simple unit propagation before any decisions must be made in a given time frame. This makes such a self-loop in the transition function almost cost-free.

Having made these changes, we can directly use a modular SAT solver (either the simple recursive Alg. 1 or the more complex SAT Modulo SAT solver Alg.3) to implement the stack-based cube-blocking procedure from IC3 (see Alg. 4).

Efficient versions of IC3, including PDR, maintain a priority queue of cubes to block rather than a stack. In this variant, when IC3 blocks a cube, it generates a new cube with the same flop assignment, but at the next time frame. This allows IC3 to discover counter-example traces that are longer than the number of time frames currently being examined [4], while at the same time improving the overall performance of IC3 [11]. In order to support this, we need to make our implementation slightly more complicated (see Alg. 5), as well as change the modular SAT solver slightly, so that it records each complete satisfying assignment of the flops in each time frame. This is a trivial one line change to the SAT modulo SAT solver. In Alg. 5, we assume that the flop assignments found for time frame i have been stored in the vector *flopAssignments*$_i$.

The priority queue version of IC3 then proceeds by repeatedly popping the lowest TCube s off the queue (a TCube is a tuple of a cube and the time frame it corresponds to), solving ϕ_0 under $\phi_1 \ldots$ under $\phi_{s.frame}$ under $s.cube$, and then adding all the cubes that were found during that process into the queue (see COLLECTALLCUBES). Effectively, this results in a combination of the priority-queue with the modular SAT solver's natural stack-based order for exploring cubes. As we will show below, this re-ordering appears to have a negative impact on performance, but one that is more than made up for by the use of the modular SAT solver.

With these changes, and otherwise following PDR's implementation (including applying ternary simulation, which we apply to α_i just before the loops in Alg. 1 and 3), we used our modular solver to implement a competitive version of the PDR variant of IC3. As we will show below, in addition to solving as many or more instances as either PDR or IC3 on three major benchmark sets, this procedure solves many *different* instances that were not previously solved by either PDR or IC3.

A. Additional Changes to IC3

We also introduce two additional alterations to IC3 to further improve our solver. The first change is fairly minor. In the priority queue variant of IC3, when a cube is blocked at time frame i, it is re-enqueued at frame $i+1$. However, if i is the last currently expanded time frame, the cube is simply discarded. Instead, we keep these cubes and enqueue them into the priority queue at frame $i+1$, and keep them in the queue for the next iteration (at which point time frame $i+1$ will have been explored). This is shown in Alg. 5, when the *keepCubes* flag is set. We only ever discard cubes from the last time frame if they are syntactically blocked. We have also explored keeping all such clauses even if they are blocked syntactically in the last time frame, and it seems to lead to only a slight decrease in performance to do so.

A more significant change addresses a drawback of IC3 (including the PDR variant). IC3 always attempts to propagate clauses from the first to the last time frame at each iteration. As a result, IC3 requires at least quadratic time in the number of frames, and that by itself can lead to unacceptable slow-downs on instances that require many iterations to be explored, even if the instance is otherwise trivial. Just such an example has been encountered in practice by users of the Z3 [8] implementation of PDR.[4]

We observe that it is not typically necessary to try propagating clauses all the way from the lowest time frame at each iteration. Instead, we have found it sufficient in practice to propagate only from the lowest *strengthened* time frame to the last, at each iteration (see Alg. 6). This is a very simple change that improves performance when an instance is explored to a very deep time bound. Informal testing on Z3's PDR variant [14] has shown that this change improves performance on the example referenced earlier.

In principle, failing to propagate clauses from the first time frame may lead to a loss of IC3's convergence guarantees. If this were a concern, it would be sufficient to force clause propagation to periodically start from the first time frame — something we have tried and found not to lead to substantial performance improvements in practice.

[3] IC3 enforces this property by ensuring that all clauses in each interpolant hold at the reset state. In cases where it would learn such a clause that does not hold at the reset state, it weakens the clause by appending a literal from the reset state that does not already appear in the clause. Such a literal is guaranteed to exist, because if no literals in the cube were opposite the polarity of the reset state, then IC3 would have found a counter-example (and exited). That literal can be used to weaken the conflict clause so that it is satisfiable at the reset state, while still blocking the cube.

We cannot combine this trick with unit propagation across modules as in Alg. 3, because such propagation may occur when an arbitrary partial assignment has been made to the flops. This partial assignment might not contain any literals opposite the reset state, in which case we would not be able to weaken the clause as IC3 does while still blocking the assignment.

[4] See, *e.g.*, http://stackoverflow.com/q/15946304

Algorithm 6 Faster clause propagation, by not attempting to propagate clauses from time frames that did not require strengthening in the current iteration.

```
function PROPAGATECLAUSES
    lowest ← 0
    for i ← (F.size()-1) ... 0 do
        if not SolverIsMarked(i) then
            lowest ← i + 1
            break
        clearSolverMark(i)
    for k ← lowest ... F.size()−1 do
        for all clauses c ∈ F[k] do
            if PROPAGATECLAUSE(c,k + 1) then
                F[k].remove(c)
                F[k + 1].add(c)
        if F[k].size()=0 then
            return 'Invariant Found'
```

IV. Experimental Results

Our implementations of both Alg. 1 and the SAT Modulo SAT solver described in Section II-A are based on MiniSat 2.2 [10], a prominent incremental SAT solver that has served as a basis for many successful SAT solvers. We implemented IC3 using this solver as described above in Alg. 5.[5]

We compare to both the publicly released IC3, and also to the current implementation of PDR in ABC (version 1.01). This implementation of PDR is also part of the SuperProve model checker that won the Hardware Model Checking Competition in 2010, 2011, and 2012.

Experiments on the 2008 instances were conducted on 32-bit 3.2GHz Intel Xeon machines with 2 MB cache under openSUSE 11.1, using 15 minute timeouts and 1500 MB memory limits. Experiments for the 2010 and 2012 instances were conducted on 64-bit, 6-core, 2.6GHz Intel Xeon machines with 12 MB cache running Red Hat Linux 5.5, using 15 minute timeouts and 7000 MB memory limits. These conditions closely match those of the 2008 and 2012 competitions, respectively. When testing each model checker (including PDR and IC3), we first used ABC to apply DAG-aware rewriting for preprocessing the circuit (using the 'rewrite' command).[6]

Using our PDR implementation with the SAT Modulo SAT solver, but without the last two improvements to IC3 from Section III-A, performance is comparable to both IC3 and PDR (see the column, 'SMS', of Table I). If we substitute the unoptimized Alg. 1 for the SAT Modulo SAT solver, performance drops substantially on all benchmarks (see column 'No SMS'). This gives us confidence that our SAT Modulo SAT solver is indeed a major improvement to Alg. 1, at least in this context.[7] This model checker is also clearly competitive with IC3 and PDR — especially on the 2010 instances. Moreover, on closer inspection, we also observed that this version of our model checker solved 17 *new* instances that were solved by neither IC3 nor PDR from the 2012 benchmarks, and 13 and 9, respectively, from the 2010 and 2008 sets.

	No SMS	SMS	**SMS-PDR**	PDR	IC3	VBS
HWMCC'08	504	587	**596**	581	586	597
HWMCC'10	684	742	**749**	733	712	781
HWMCC'12	69	84	**92**	84	48	233

TABLE I: Total instances solved within 900 seconds from the 2008, 2010, and 2012 Hardware Model Checking Competitions (single property track). Including all improvements, our final implementation ('SMS-PDR') beats both IC3 and PDR on each benchmark. Notice how for the 2008 benchmarks we solve just 1 fewer instance than the virtual best solver ('VBS' — the virtual best solver counts all instances solved by *any* solver in the competition).

	SMS-PDR		PDR		IC3	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
HWMCC'08	**245**	**351**	242	339	240	346
HWMCC'10	**322**	**427**	317	416	308	404
HWMCC'12	**25**	**67**	21	63	14	34

TABLE II: Breakdown of SAT and UNSAT instances from Table I. SMS-PDR solves more SAT and more UNSAT instances than both PDR and IC3 on all three benchmarks.

We can then ask whether we can improve our solver to solve more of the instances that IC3 and PDR solve, without giving up these new instances. We accomplish exactly this, by adding the last two improvements discussed in Section III-A. These improvements allow us to solve several additional instances (all but 3 of which IC3 or PDR could already solve), without giving up *any* of the newly solved instances of our initial implementation (see column 'SMS-PDR' in Tables I and II).

From Table I, we note that on the 2008 instances, our final model checker solves just one instance fewer than the corresponding virtual best solver — the virtual best solver counts any instance solved by any solver running in that competition — under roughly the same conditions. In Table II, we split the results for each model checker into SAT and UNSAT instances, to show that for all three competitions, we always solve more SAT *and* more UNSAT instances than both IC3 and PDR. In contrast, notice that while PDR improved hugely upon IC3's performance on the 2012 instances, it actually performed slightly worse on the 2008 instances.

The improvements introduced in Section III-A, as well as the use of our SAT Modulo SAT solver, both increased

[5]We have made the source code for the modular SAT solver available online, at www.cs.ubc.ca/labs/isd/Projects/ModularSAT.

[6]Results for each competition's virtual best solver is as reported in the respective competitions.

[7]At the same time, we can ask why it is the case that when using Alg. 1 instead of Alg. 3, our performance is so much worse than IC3's. As discussed in Sec. III, there are effectively just a few differences between PDR's recBlockCube and Alg. 5, if the unoptimized modular SAT solver of Alg. 1 is used and if our last two changes to IC3 are not implemented. The performance drop *relative to IC3* in this case is likely either due to our delaying inductive generalization until later in the process, or a consequence of using a self-loop in the transition function (though we argue why this is not likely in Sec. III).

the total number of solved instances. However, the SAT modulo SAT solver by itself contributes most of the *newly* solved instances — that is, instances that we solved, but that neither IC3 nor PDR could solve. Our model checker using just the SAT modulo SAT solver solved 9, 13, and 17 *new* instances in the 2008, 1010, and 2012 benchmarks, while combining the SAT Modulo SAT solver with the changes from Section III-A solved 9, 15, and 21 such instances. On this basis, we argue that the SAT Modulo SAT solver is critical to the overall performance improvement achieved by our final model checker.[8]

We can also look at the respective memory usage of our solvers. As we remarked earlier, like current versions of ABC's PDR, we instantiate a solver for each time step, which results in roughly linear memory usage in the number of time steps. This gives up one of the original advantages of IC3, which is that it expands only one time frame at a time, which requires roughly constant memory. Both of these bounds ignore the theoretically exponential memory of the learned clauses and interpolants.

We found that our solver ran out of memory on 13, 1, and 7 instances for the 2008, 2010, and 2012 benchmarks (recall that the 2008 competition was limited to just 1.5 GB, vs 7 GB for the others). In contrast, IC3 ran out of memory on just two instances in our experiments, both from the 2012 benchmarks. However, there was only one case in which our solver ran out of memory on an instance that IC3 was able to solve - and that particular instance, from the 2008 benchmark set, was one that our solver *was* able to solve in the 2010 benchmark set (which had a higher memory limit). So, as we would expect, IC3's near-constant memory usage is an advantage on some instances.

V. Conclusion

We have introduced a novel approach for modular SAT solving, which naturally computes sequence interpolants without proofs. We have made this efficient through the use of standard techniques borrowed from lazy SMT solvers, and we have shown that this can form the basis of an efficient model checker. We have also introduced additional improvements to IC3 that should generalize to other implementations, including PDR, whether or not they utilize our SAT Modulo SAT solver. The resulting state-of-the-art model checker performs better than both PDR and IC3, for both SAT and UNSAT instances, on three competitive sets of benchmarks.

VI. Acknowledgments

We thank Armin Biere for his insights about the connection between other CNF partitioning solvers and modular solvers. We thank Nikolaj Bjorner for pointing us to the loop example cited in Section III, and for testing our faster clause propagation in Z3. We also thank the anonymous reviewers of this paper, as well as of a previous manuscript which led to this work.

This research has been supported by the use of computing resources provided by WestGrid and Compute/Calcul Canada, and by funding provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

[8] We can also ask how the changes from Section III-A fare on their own. Implementing them in the non-SMS version of our solver does not improve performance at all, while implementing them in ABC's PDR led to 2 and 3 additional instances solved on the 2008 and 2010 benchmarks, and 3 fewer solved on the 2012 benchmarks.

References

[1] F. Aloul, I. Markov, and K. Sakallah, "MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation," *Journal of Universal Computer Science*, vol. 10, no. 12, pp. 1562–1596, 2004.

[2] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, no. 2-4, pp. 75–97, 2008.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs*. Springer, 1999.

[4] A. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2011, pp. 70–87.

[5] G. Cabodi, S. Nocco, and S. Quer, "Interpolation sequences revisited," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[6] H. Chockler, A. Ivrii, and A. Matsliah, "Computing interpolants without proofs," in *Proceedings of the Eighth Haifa Verification Conference*, 2012.

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

[8] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[9] V. Durairaj and P. Kalla, "Guiding CNF-SAT search via efficient constraint partitioning," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society, 2004, pp. 498–501.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*. Springer, 2004, pp. 333–336.

[11] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Formal Methods in Computer-Aided Design (FMCAD), 2011*. IEEE, 2011, pp. 125–134.

[12] O. Grumberg, A. Schuster, and A. Yadgar, "Memory efficient all-solutions SAT solver and its application for reachability analysis," in *Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 275–289.

[13] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-based decision heuristics for image computation using SAT and BDDs," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2001, pp. 286–292.

[14] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing–SAT 2012*. Springer, 2012, pp. 157–171.

[15] A. Hyvärinen, T. Junttila, and I. Niemelä, "Partitioning SAT instances for distributed solving," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2010, pp. 372–386.

[16] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*. Springer, 2003, pp. 1–13.

[17] T. Park and A. Van Gelder, "Partitioning methods for satisfiability testing on large formulas," *Automated Deduction âĂŤ CADE-13*, pp. 748–762, 1996.

[18] D. Ranjan, D. Tang, and S. Malik, "A comparative study of 2QBF algorithms," in *The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004, pp. 292–305.

[19] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 3, pp. 141–224, 2007.

[20] Y. Vizel, V. Ryvchin, and A. Nadel, "Efficient generation of small interpolants in CNF," in *Computer Aided Verification*, 2013, pp. 330 – 346.

[21] Y. Vizel and O. Grumberg, "Interpolation-sequence based model checking," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. IEEE, 2009, pp. 1–8.